TYPE-SAFE MODELING FOR OPTIMIZATION

TYPE-SAFE MODELING FOR OPTIMIZATION

By NHAN THAI, B.Eng.

A Thesis Submitted to the School of Graduate Studies in Partial
Fulfilment of the Requirements for
the degree Master of Science

McMaster University MASTER OF SCIENCE (2021) Hamilton, Ontario (Computer Science)

TITLE: Type-safe modeling for optimization AUTHOR: Nhan Thai, B.Eng (Hanoi University of Science and Technology) SUPERVISORS: Dr. Christopher Anand and Dr. Wolfram Kahl NUMBER OF PAGES: xi, 67

Abstract

Mathematical optimization has many applications in operations research, image processing, and machine learning, demanding not only computational efficiency but also convenience and correctness in constructing complex models. In this work, we introduce HashedExpression, an open-source algebraic modeling language (AML) that allows users to express unconstrained, box-constrained, and scalar-expressions-constrained optimization problems, aimed at embeddability, type-safety, and high-performance through symbolic transformation and code generation. Written in Haskell, a statically-typed, purely functional programming language, HashedExpression places a great emphasis on modeling correctness by providing users with a type-safe, correct-by-construction interface that uses Haskell type-level programming to express constraints on correctness which the compiler uses to flag many modelling errors as type errors (at compile time). We show how type-safety can be added in steps, first matching expressions' shape and then associated physical units. The library implements symbolic expressions with a hashed indexing scheme to implement common subexpression elimination (CSE). It abstracts away details of the underlying lookup table via monadic type class instances. We explain how using symbolic expressions with CSE enables performance-enhance transformations and automatic computation of derivatives without the issue of "expression swelling". For high-performance purposes, we generate low-level C/C++ code for symbolic expressions and provide bindings to open-source optimization solvers such as Ipopt or L-BFGS-B. We explain how this architecture lays the groundwork for future work on parallelization including SIMDization and targetting multi-core CPUs and GPUs, and other hardware acceleration.

Acknowledgements

First and foremost, I would like to thank Dr. Christopher Anand for his continuous support, patience, and being a great role model throughout my MSc study. I would also like to extend my sincere thanks to Dr. Wolfram Kahl for his guidance and for being an enthusiastic instructor. Their immense knowledge and experience have inspired me beyond words.

I would also like to acknowledge my fellow students, especially Curtis D'Alves, who have helped me with this research and writing this dissertation.

Thanks to my friends and for your great ideas, conversations, the pat on the back, and awesome soccer games.

Thanks to my family, my dad Thành and my mom Thanh, my father-in-law Thuận, my mother-in-law Nguyệt, for always looking after me and putting my wellness above all.

And thanks to my wife, Thu Hải, for always being there.

Contents

List of Figures

List of Listings

List of Abbreviations

COCONUT	Code Construction Tools
AML	Algebraic Modeling Language
AMPL	A Mathematical Programming Language
GAMS	General Algebraic Modeling System
AD	Automatic differentiation
YALMIP	Yet Another LMI Parser
MRI	Magnetic Resonance Imaging
L-BFGS	Limited-memory Broyden–Fletcher–Goldfarb–Shanno
L-BFGS-B	Limited-memory Broyden–Fletcher–Goldfarb–Shanno Bounded
DAG	Direct Acyclic Graph
CSE	Common Subexpression Elimination
API	Application Programming Interface
GHC	Glaslow Haskell Compiler
SI	International System of Units
SIMD	Single Instruction Multiple Data
GPU	Graphics Processing Unit
SEM	Scanning Electron Microscope
BLAS	Basic Linear Algebra Subprograms

Declaration of Academic Achievement

I , Nhan Q. D. Thai, declare this thesis to be my own scholar work, and I have appropriately acknowledged and cited all material from the work of others (in articles, books, dissertations, on the internet, etc.).

No part of this work has been published or submitted for a degree at another institution, and, to the best of my knowledge, this thesis work does not violate anyone's copyright.

This thesis is the continuation of the work of Jessica Pavlin's thesis [56]. My supervisors, Dr. Christopher Anand and Dr. Wolfram Kahl, have provided advice and support throughout the course of this research project. I completed all the research work.

Chapter 1

Introduction

Operations Research (including mathematical optimization) become an established discipline during World War II, when the British government recruited scientists to solve problems in critical military operations [5]. The use of mathematical optimization has now become critical in the use of many industries, including: Airline Transportation (routing and flight plans, crew scheduling, revenue management); Telecommunications (network routing, queue control); Manufacturing Industry (system throughput and bottleneck analysis, inventory control, production scheduling, capacity planning); Health care (hospital management, facility design); and Transportation (traffic control, logistics, network flow, airport terminal layout, location planning). And this is a rapidly growing field. According to the U.S. Bureau of Labor Statistics there is an expected 25% job increase anticipated between 2019-2029 [38]. The demand to solve large multi-variate optimization problems requiring a high degree of expertise is clearly growing. Tackling optimization problems demands not only expertise and high performance computational resources but also correctness in modeling them. In this thesis, we introduce HashedExpression, an embedded algebraic modeling language for expressing and solving large scale optimization problems. We show that by taking advantage of generative programming techniques in strongly typed languages, our tool enables special functionality not found in other modeling languages and is powerful to use in many settings.

1.1 Problem Definition

Solving mathematical optimization is a process involving multiple steps often assisted by the use of algebraic modeling languages (AMLs). However, many problems exist with the current major AMLs:

1. **Embeddability** Some of them are standalone and commercial software, making it difficult to embed them in modern software systems.

2. **Type-safety** Lacking type-safety, users are prone to making mistakes when translating their optimization model to the modeling language.

3. **Computation Optimization** No clear separation between model and data, which limits the uses of mathematical properties to reduce and optimize computations.

4. **Performance** AMLs built within high-level programming languages often compromise performance.

With such problems, we seek to use Haskell, a purely functional programming language with a unique set of features, to build an algebraic modeling language that resolves all of the abovementioned issues.

1.2 Literature Review

In optimization applications, we map the domain-specific problem target to a real-valued function whose domain is encoded as real vectors together with some constraints, and bring it to the standard form (1.1). Our task is to translate the mathematical formulas into machine representation and compute the solution by an optimization solver. Then, the computed solution is mapped back to the original domain.

$$\begin{array}{ll} \text{minimize:} & f(x) \\ \text{subject to:} & g_i(x) \leq 0, \quad i = 1, 2, ..., m_g \\ & h_i(x) = 0, \quad i = 1, 2, ..., m_h \\ & (f, g_i, h_i : \mathbb{R}^n \longrightarrow \mathbb{R}) \end{array} \tag{1.1}$$

This application pipeline relies not only on the power of the optimization solvers, but also a facility to provide input to such optimizers from the original formula (1.1). Traditionally, optimization solvers are written at low-level to squeeze performance, and manually input the requisite format for them is tedious, error-prone and does not scale well with problems' complexity. This is the motivation of algebraic modeling languages (AMLs), software that provide a high level interface between the optimization model and solvers. In this sense, the need for AMLs is analogous to the need for higher-level programming languages on top of assembly languages. AMLs allow domain experts to express optimization problems in algebraic forms that resemble the original mathematical expressions. Implementing an AML typically involves translating user algebraic models, performing necessary transformations, and providing requisite input to optimization solvers, as shown in Figure 1.1.

The earliest (commercially used) tools for algebraic modeling of optimization problems were developed in the late 1970s and early 1980s. AMPL [25] and GAMS [9], which are still widely used nowadays, are among this list. AMLs soon proved to be superior to IBM's matrix generators MPS in handling linear programming (LP) models [34]. They also became the standard for formulating general derivative-based nonlinear optimization problems, whose solution relies on the availability of first-order derivatives ∇f, ∇g_i and ∇h_i, with support for automatic differentiation (AD) [30]. These successes set in motion the ongoing

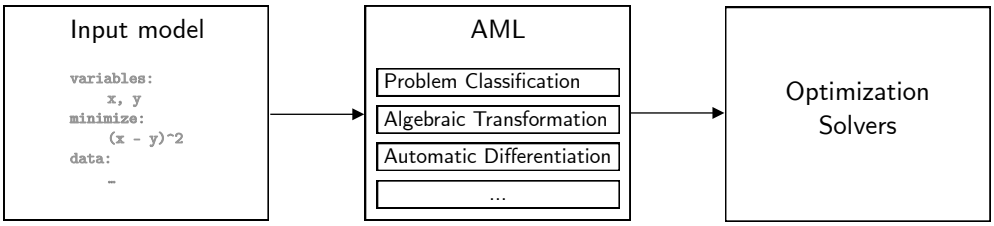


Figure 1.1: AML accepts user-provided models and generates required input to mathematical optimization solvers

development of many algebraic modeling systems. Understandably, the development of AMLs, especially embedded ones, has been going on in parallel with programming languages in general.

The following is a list of popular AMLs – incomplete but covering the wide diversity in functionality – being used in both industry and academia to date:

- AMPL and GAMS are among the most popular commercial modeling software. They provide a standalone modeling language with integrated tools and control commands, together with support for a growing list of supported solvers.
- Pyomo [31] is a Python library that provides similar capabilities to AMPL and GAMS.
- YALMIP [43] is a MATLAB library initially developed for modeling and solving semidefinite programming (SDP) [62], but has since evolved to support a wider range of optimization problems.
- The CVX* family [27, 18, 61] targets disciplined convex optimization, which ensures convexity by establishing a set of rules when constructing models.
- JuMP [20] targeting high performance and a nice syntax rendered possible by Julia's syntactic macros.
- Tensorflow [1] and Pytorch [55], two well-known Python libraries in the machine learning community, can also largely be used as AMLs for unconstrained optimization.

AMPL and GAMS provide first-class support for mathematical optimization with an established user base in both academia and industry. However, being standalone systems, they do not compose naturally with modern application pipelines, which usually involve connecting different software components and reusing related problem instances [23, 20]. Moreover, learning how to use a monolithic modeling system like AMPL and GAMS is considered more challenging than an embedded AML built in a host language users are familiar with.

This is why most modern AMLs, such as YALMIP, Pyomo, or Pytorch, are embedded in Python or MATLAB. With an abundance of packages for

data processing and visualization plus the popularity of such languages among the scientific computing community, these solutions are highly approachable and are easier to compose. However, libraries built on MATLAB and Python come with performance compromises. A mitigation of this is to use numerical libraries like NumPy [65] whose implementation is a thinly layered wrapper around C/C++ kernels, or use domain-specific compilers to accelerate mathematical computation (like Tensorflow's Accelerated Linear Algebra and JAX [6]). However, it is still confusing for inexperienced users to distinguish between fast and slow codes.

But while progress has been made in developing user-friendly modeling languages that don't compromise performance, another issue has been neglected. Surveys of bugs in numerical computation with popular libraries (including NumPy, SciPy, and LAPACK) have found that correctness bugs are the most frequently occurring and challenging to detect [17]. We hypothesize the inherent issue with Python and MATLAB modeling libraries that leads to correctness bugs is that they are built on languages that are mutable and dynamically typed. A variable can have any type and can point to anything at different times. Because of this, there are many ways models go wrong, and users have to run the program to find type errors. However, the reported errors do not always trace to the line that contains the mistake, as illustrated in Figure 1.2. We argue that this largely goes against the premise of a modeling language itself.

```
# internal library representation, hidden to users
class Plus:
  def __init__(self, arg1, arg2):
    self.arg1 = arg1
    self.arg2 = arg2
  def eval(self):
    return self.arg1 + self.arg2

# modeling
x         = 1
y         = [2, 3]
objective = Plus(x, y)

# the library calls eval() later, also hidden to users
objective.eval()
```

```
Traceback (most recent call last):
  File "…", line 15, in <module>
    objective.eval()
  File "…", line 7, in eval
    return self.arg1 + self.arg2
TypeError: unsupported operand
type(s) for +: 'int' and 'list'
```

Figure 1.2: Mistakes in modeling steps but error messages show traces to internal library codes

JuMP [20] (in Julia) broadly addresses the performance limitation, with Julia allows explicit type annotation [60] and therefore assists in catching programmer errors to some extent. However, the language is still dynamically typed and does not fully address our type-safety concerns. Moreover, we find JuMP's macros confusing in some cases, e.g., not every function available in normal declarations is usable in macro declarations. Another shortcoming of

JuMP is the lack of support for complex numbers often required in signal processing applications involving the Fourier transform.

As such, we feel that a functional programming language with enforced immutability is intrinsically more suitable concerning the algebraic aspect of our targeted software category. We identify the following set of features as particularly invaluable for building our algebraic modeling library:

- **Type-safety** ensures correctness of constructed optimization models. This helps users capture errors early in the development cycle where invalid models will result in compile errors.
- **Flexible overloading** and custom infix operators for representing mathematical expressions.
- **Powerful abstraction** allows us to manipulate the internal hashed representation of expressions while still keeping the code clean and easy to understand, which is the foundation for us to implement various algebraic modeling language features.
- **Type-level programming** and support for type-level natural numbers allow us to encode various metadata into the type system. Such metadata could include expressions' shapes, units, or sampling steps.

These facts lead us to pick Haskell, a statically typed, purely functional programming language to embed our modeling language. On top of that, Haskell's type system can assist users in constructing their models and provide an interactive development environment, e.g., the Haskell compiler informs users which "type" a to-be-filled value should have (see Figure 1.3). Functions in Haskell with type signatures can also serve as a good source of documentation, especially for mathematicians and physicists alike.

1.3 Goals of HashedExpression

The aforementioned drawbacks of existing AMLs and the special capability offered by the host language Haskell motivated us to develop HashedExpresison, an AML that:

G1. Allows for type-safe modeling of optimization problems.

G2. Uses concrete syntax as close to the syntax used in mathematical and physical models as possible.

G3. Has first-class support for multi-dimensional variables and complex numbers.

G4. Automatic computation of derivatives.

G5. Allows for efficient performance-enhancing expression transformations.

G6. Generates C code for interfacing with optimization solvers.

Figure 1.3: Interactive development in Haskell (VS Code)

HashedExpression evolved from the Coconut Expression Library (CEL) [56] which only supported unconstrained optimization. A complete rewrite with substantial changes has been made compared to the original library. Such improvements include introducing phantom types and type-level programming to enforce a higher level of correctness modeling. Type-safety modeling is added in steps, first through shape and then with physical units layered on top. The rewrite has now included supports constrained optimizations (box and general-form constraints). It also implements reverse-mode automatic differentiation as opposed to the exterior derivative approach taken by CEL.

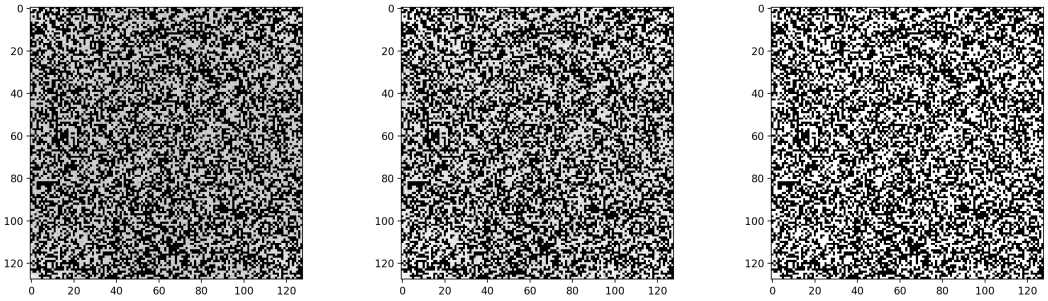
In the following chapters, we will discuss how each goal is accomplished.

1.4 Example: Solving Magnetic Resonance Imaging Reconstruction

In Magnetic Resonance Imaging (MRI) reconstruction problems, we take the raw data taken from MRI machines and try to construct the original measured anatomical image. The details of MRI physics is outside the scope of this thesis, but it suffices to know that the captured data m is a sampled Fourier transform of the original image.

In this case, our collected data $m = a + bi$ has shape 128×128 of complex numbers. This is read from a real part a (1.4a) and a complex part b (1.4b), both with shape 128×128 and are characterized by a 128×128 mask (1.4c) of 1 for sampled kixels[1] and 0 for unsampled kixels.

[1] A kixel is a sample on a grid in Kristallographie space.



(a) Real part (logscale) (b) Imaginary part (c) Mask

Figure 1.4: Collected data

Instead of naively taking the inverse Fourier transform, which only gives us the noisy image (1.5b), we get the reconstruction by solving the following optimization problem which has a regularization term:

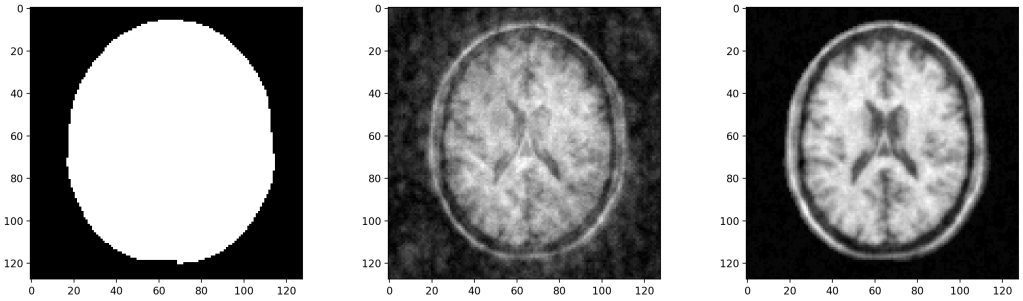
$$\begin{aligned}\hat{x} = \arg\min_{x} \ & \|\pi_{\text{mask}}(\text{FT}(x) - m)\|^2 \\ & +\lambda \sum_{i,j} \left((x_{i+1,j} - x_{i,j})^2 + (x_{i,j+1} - x_{i,j})^2\right)\end{aligned} \tag{1.2}$$

subject to:

$$x_{lb} \preceq x \preceq x_{ub} \tag{1.3}$$

where $\preceq$ denotes the point-wise less than or equal relation.

The loss term $\|\pi_{\text{mask}}(\text{FT}(x)-m)\|^2$ tells us to find the original image x whose Fourier transform is the measured m, but all the differences in unsampled kixels are ignored, which is indicated by the operation π_{mask}. The box constraints 1.3 indicates that all the kixels inside the head are positive, and kixels outside are close to 0 (between allowed noise range). The region of the head is known in the screening process, represented by 1.5a. The regularization penalizes the difference between neighbor kixels so that the solution will favour images with smaller jumps between adjacent kixels.



(a) Boundary of the head.

(b) Naively reconstructed by taking the inverse Fourier transform.

(c) Using optimization problem with regularization.

Figure 1.5: Result reconstructed images

The next step is to express our optimization problem in HashedExpression. First, we declare our target optimization variable x:

```
-- x has type TypedExpr '[128, 128] R
x = variable2D @128 @128 "x"
```

Then, we declare the parameters of the problem. In this case, they are the real part, the imaginary part, and the sampled mask of the MRI signal:

```
-- these have type TypedExpr '[128, 128] R
a = param2D @128 @128 "a"
b = param2D @128 @128 "b"
mask = param2D @128 @128 "mask"
```

As x is subjected to box constraints, we also need to declare the bounds for it as following:

```
-- these have type Bound '[128, 128]
xLowerBound = bound2D @128 @128 "x_lb"
xUpperBound = bound2D @128 @128 "x_ub"
```

The regularization term can be expressed by the norm square of the difference of x and its (both horizontal and vertical) 1-off-rotations (This works because we know that the edge kixels should all be close to 0):

```
-- regularization has type TypedExpr Scalar R
regularization = norm2square (rotate (0, 1) x - x)
  + norm2square (rotate (1, 0) x - x)
```

Then, we construct an `OptimizationProblem` by providing its objective, constraints and values for the parameters (which is read from HDF5 [24] files in this case):

```
reconstructionMRI :: OptimizationProblem
reconstructionMRI =
```

```
OptimizationProblem
  { objective =
      norm2square ((mask +: 0) * (ft (x +: 0) - (a +: b))) + 3000
      ↪  * regularization,
    constraints =
      [ x .<= xUpperBound,
        x .>= xLowerBound
      ],
    values =
      [ a :-> VFile (HDF5 "kspace.h5" "a"),
        b :-> VFile (HDF5 "kspace.h5" "b"),
        mask :-> VFile (HDF5 "mask.h5" "mask"),
        xLowerBound :-> VFile (HDF5 "bound.h5" "lb"),
        xUpperBound :-> VFile (HDF5 "bound.h5" "ub")
      ]
  }
```

We can see a clear resemblance between the original mathematical expressions and corresponding ones[2] in HashedExpression. Moreover, expressions are typed with information of their shape and number type ($\mathbb{R}$ or $\mathbb{C}$), which help in preventing us from constructing invalid expressions.

Finally, we use HashedExpression to generate C code which is then combined with an optimizer to produce the final result (1.5c). In this example, our optimization problem is convex with box constraints, so we use L-BFGS-B-C [59], a C implementation of the L-BFGS-B [69, 48] algorithm.

This image reconstruction example shows how our Haskell-embedded AML can be used to solve a real-world medical imaging problem. Further details of this example can be found at `https://github.com/McMasterU/HashedExpression/blob/master/app/Examples`. The example brain image is from [8, 15].

[2]The `(+:)` operation constructs a complex expression from a real part and an imaginary part; `a +: b` is equivalent to $a + bi$

Chapter 2

Expression Graph

Expressions are the basic building blocks of AMLs and automatic differentiation libraries. Generally, operator overloading is used so user constructed expressions will result in some form of dependency-based data structure, usually directed acyclic graphs (DAGs). This can be symbolically constructed ahead of time (and then run many times later) like Pyomo [31], Sympy [46] and Tensorflow 1.x [1], or eagerly executed and implicitly establish the relationship between nodes as in Pytorch [55] [54]. The expression graphs are useful for the computation of derivatives and serve as an efficient implementation for performance optimization.

In HashedExpression, expressions are represented as immutable, pure symbolic directed acyclic graphs, without numerical data associated. This provides a useful justification for many kinds of algebraic graph operations. Such expression graphs can be duplicated, merged, or transformed into equivalent ones by graph rewriting, the technique we will use to accelerate expressions computations. Automatic computing of derivatives works in the same manner: derive the expression graph for the derivatives from the original graph.

What makes HashedExpression truly unique however, and where it derives it's name, is that expressions are encoded in an indexing-based data structure that allows for common subexpression elimination (CSE), including subexpressions common between different expressions, such as the objective, constraints and the derivatives. In the next sections, we will go into details on the implementation of expression graph, as well as how CSE is enforced by hashing.

2.1 Implementation

Expressions are represented by a graph structure encoded in a node lookup table. Each node represents a subexpression. Nodes are indexed by their identifier `NodeID` :

```
-- NodeID is a zero-overhead wrapper of Int
newtype NodeID = NodeID Int
-- IntMap Node is used instead of Map NodeID Node
-- for performance purpose
type ExpressionMap = IntMap Node
```

Here, the `NodeID` is a wrapper around `Int` for type-safety purpose. For `ExpressionMap`, we use the strict version of the performant immutable integer map `IntMap` [53].

The associated Haskell data types for representing nodes are as follow:

```
-- []     => scalar
-- [n]    => 1 dimensional n
-- [m, n] => 2 dimensional mxn
type Shape = [Int]

data ElementType = R | C

data Op
  = Var String             -- variable with an identifier
  | Param String           -- parameter with an identifier
  | Const Double           -- constant
  | Sum [NodeID]           -- sum
  | Prod [NodeID]          -- product
  | Scale NodeID NodeID    -- scale
  | RealImag NodeID NodeID -- complex from real and imaginary
  | ..

type Node = (Shape, ElementType, Op)
```

Each node is a tuple of:

- `Shape` that contains dimensional data of the expression. An empty list represents a scalar expression, a list of one element `[n]` represent an one-dimensional expression of size `n`, and so on.
- `ElementType` tells us if the expression has values of real or complex numbers.
- `Op`, an algebraic data type that indicates if the expression is a variable, a parameter, a constant, or an operator application (together with indices of its operands).

Each constructor of `Op` is named after the corresponding mathematical operation. The order of operands tagged in each constructor matters, and the constructor may also contain additional information beside its operands. For example, the element-wise power constructor `Power Int NodeID` contains its integer exponent, or the piecewise constructor `Piecewise [Double] NodeID [NodeID]` contains the list of endpoints that make up the piecewise intervals.

Illustrated in Figure 2.1 is an example of an expression graph. It contains six subexpressions, all of which are scalar and real. Edges are directed from operands to operators similar to dataflow graphs with the final expression $x(x+y)\log(x+y)$. The lookup table structure on the right resembles the adjacency list representation of graphs: each node has a list of its adjacent indices.

An expression is therefore identified by a tuple: the underlying expression lookup table, and its node index:

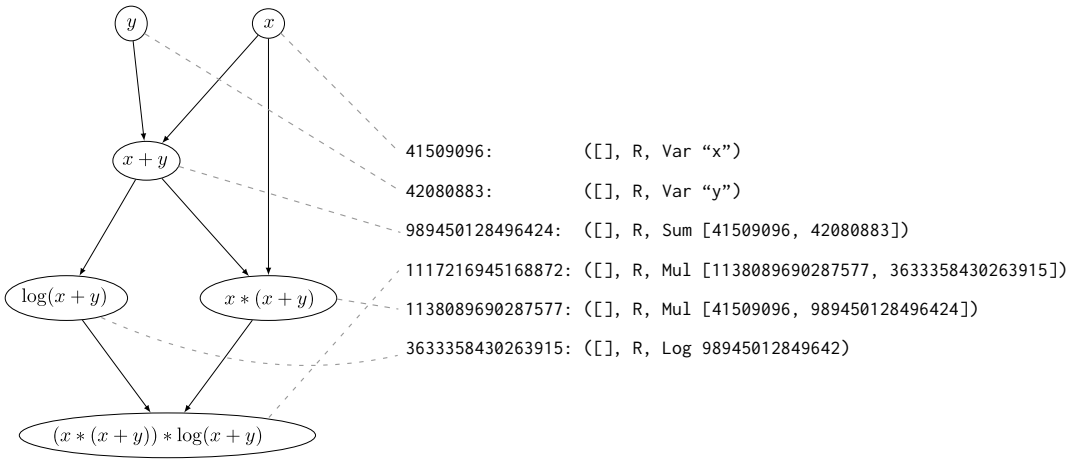


Figure 2.1: Each entry in the lookup table corresponds to a subexpression

```
type RawExpr = (ExpressionMap, NodeID)
```

Expressions can share the same lookup table. Moreover, expression graphs are not limited to a sink expression, i.e., nodes without outgoing edges.

The representation relies on `NodeID` lookup key to implement a pointer base structure required for expression graphs. As such, there is a data structure invariant we must maintain: operand indices in node entries must also exist in the expression look up table. All of these details must be abstracted away from users when constructing expressions, and to library developers when developing algebraic modeling features.

2.2 Common Subexpression Elimination by Hashing

Common subexpression elimination (CSE) is a well-known problem in compiler optimization theory. The task is to find identical expressions and replace them with a single variable holding the common value, aimed at minimizing the number of computational steps and memory requirements in the evaluation of mathematical expressions.

In general programming languages, this problem can be challenging in the presence of control flow and computational side effects. Fortunately, in our symbolic and pure representation of mathematical expressions, things are less tricky. The problem comes down to finding equivalent expression nodes in the code graph and replacing them with a single entry. As we are dealing with mathematical expressions, CSE problems arise at different levels:

- Nominal common subexpression elimination: identical symbolic terms share the same node in the expression graph. An example is shown in Figure 2.2. The symbolic term $x * (y + z)$ is contained in both A and B but appears only once in the final expression graph of f.

- Elimination up to algebraic equivalence of expressions. For instances, due to commutativity and associativity of addition, $x + (y + z)$ is the same as $(x + z) + y$. This will be dealt with using expression rewriting introduced in Chapter 5.

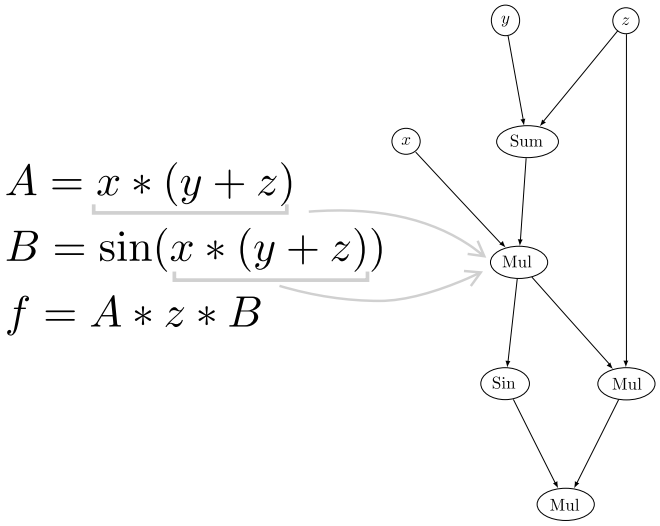


Figure 2.2: Identical terms sharing the same node

Fortunately, with the way expression graphs are structured, we have a straightforward way to achieve nominal common subexpression elimination: index nodes by the hash value of their `Node` contents, i.e., the tuple `(Shape, ElementType, Op)`.

This ensures no expression nodes with identical content are stored and indexed more than once in the lookup table, but this fact alone does not prove that we have achieved nominal common subexpression elimination because node content does not necessarily correspond one-to-one with the symbolic expression it represents. For instances, in Figure 2.1, the node with index `989450128496424` represents the symbolic term $x + y$, but its content is `Sum [41509096, 42080883]` under the context that x is at `41509096` and y is at `42080883`.

However, we can prove this by using a simple inductive argument. First, we know that the base constructors:

```
| Var String
| Param String
| Const Double
```

correspond one-to-one with the symbolic expressions they represent, so nominal common subexpression elimination holds with symbolic terms built on such constructors. For the induction step, let f and g be two identical symbolic expressions, and assume that the property holds for all subterms of f and g. Then, we know that identical corresponding subterms of the two are indexed as the same node in the lookup table. Because of this, the constructed nodes representing f and g are identical (same operator and same operand indices).

Therefore, they are indexed by the same key. Thus, nominal common subexpression elimination holds for all input symbolic expressions.

Our hashing scheme is based on polynomial string hashing introduced in the Rabin-Karp algorithm [35]. Strings are treated as numbers of a given radix, and a hash is performed by taking the remainder of this number on a modulo. The radix and modulo are often prime numbers, and the modulo should be as big as possible, as long as it fits within the range represented by `Int`. In Haskell, `Int` stores 64-bit integers, so the number space should be enough for our practical uses.

```haskell
modulo :: Int
modulo = 253931039382791

radix :: Int
radix = 137

hashString :: String -> Int
hashString [] = 0
hashString (x : xs) = (ord x + radix * hashString xs) `mod` modulo
```

Listing 1: Polynominal string hashing

Nodes are mapped to corresponding ASCII strings. Then, we hash the string and offset the hash outcome by a multiple of `modulo`. This offset multiple differs between `Op` constructors, so nodes of different operators always have different hash outcomes. Moreover, for handling hash collisions, the hash function has an extra argument: the number of rehashing. The following code snippet gives an outline of our hashing process:

```haskell
offsetHash :: Int -> Int -> Int
offsetHash offset hash = offset * modulo + hash

hash :: Node -> Int -> Int
hash (shape, et, op) rehashNum =
  let -- makeString is injective
      str = makeString shape et op rehashNum
      -- based on operator constructor
      -- e.g. Var -> 0, Param -> 1, Const -> 2
      offset = getOpOffset op
   in offsetHash offset $ hashString str
```

When forming a new expression on top of an existing lookup table, we will use the smallest rehash number (starting from 0) that results in no hash collisions (i.e., the result index is not already associated with a node with different content). This function is elegantly expressed thanks to Haskell's lazy evaluation and infinite lists, as shown in Listing 2.

```haskell
index :: ExpressionMap -> Node -> Int
index expressionMap node =
  head . filter notConflict . map (hash node) $ [0 ..]
  where
    notConflict id = case lookup expressionMap id of
      Just existingNode -> node == existingNode
      Nothing -> True
```

Listing 2: Produce the index for a new expression node

Figure 2.3 illustrates how expressions (expressed in high-level modeling API) are hashed and indexed in the node lookup table internally. Note that expression map argument is not explicitly provided to `hashIndex`; readers should assume that it starts with an empty lookup table, and the lookup table result of the previous construction is bound to the next one.

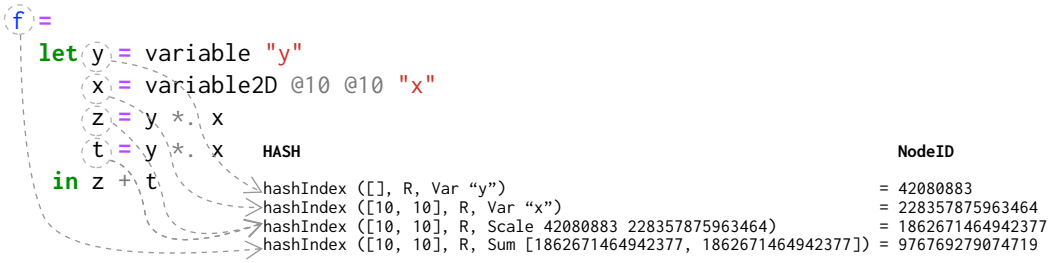


Figure 2.3: Expressions are indexed with their hash values under the hood

This index-by-content scheme also serves as a good basis for us to implement common subexpression elimination up to algebraic equivalence. The problem comes down to rewriting algebraically equivalent terms to the same symbolic expression, and they'll automatically get indexed as a single common node. Moreover, by indexing with hash, we can confirm identical symbolic (sub)expressions in O(1) complexity. This allows for an efficient matching algorithm for term rewriting which we will discuss in Chapter 5.

Chapter 3

Expression Composition

In this chapter, we will discuss possible methods of composing expressions. In a declarative, symbolic modeling software, this is the foundation underlying almost every task: from constructing objective functions and constraints to expression rewriting and computing derivatives. Representing expression graphs in a hash lookup table gives us fine control over common subexpressions but also brings with it the challenge of handling possible hash collisions. This requires additional bookkeeping for the direct method as it involves merging lookup tables. Moreover, the efficiency of the direct method is questionable when dealing with expressions sharing the same lookup table. As such, we will discuss a better computational alternative made possible by the power of Haskell abstraction, namely monadic structure [64].

3.1 Direct Method

3.1.1 Expression as a Tuple

The direct method deals with a concrete representation of expressions. This is expressed as a tuple consisting of the underlying expression map (lookup table) and its node index.

```
type RawExpr = (ExpressionMap, NodeID)
```

Treating this encoding as the base type, composing expressions becomes an algebra on the set of `RawExpr`. For example, adding two expressions has the type signature:

```
add :: RawExpr -> RawExpr -> RawExpr
```

Then, we can combine expressions with an operator by following this outline:

1. Merge all the operands' lookup tables into one single lookup table.
2. Create a new node from the operator and operand nodes
3. Find the hash index for the node, as illustrated in Listing 2.

4. Return the result tuple: the merged lookup table inserted with the created node (with the key is the hash index), and the hash index itself.

Based on the fact that `ExpressionMap` is an `IntMap` which can be merged using the built-in function `IntMap.union`, a naive implementation of this algorithm would look like:

```
add :: RawExpr -> RawExpr -> RawExpr
add (mp1, xID) (mp2, yID) =
  let mp   = IntMap.union mp1 mp2
      node = createSumNode [(mp1, xID), (mp2, yID)]
      nID  = index mp node
   in (IntMap.insert nID mp, NodeID nID)
```

This implementation is straightforward and simple. It has a linear time complexity of $O(m+n)$ where m and n are the number of entries in each lookup table (time complexity of `IntMap.union`)). However, it fails to account for possible hash collisions. The two expressions could be constructed separately, so there is no collision resolving scheme guaranteed between the two lookup tables. Different nodes representing different symbolic expressions maybe have the same index in separate lookup tables (see Figure 3.1). Thus, when we merge the lookup tables this way, one of them will be discarded, therefore yielding an invalid expressions graph.

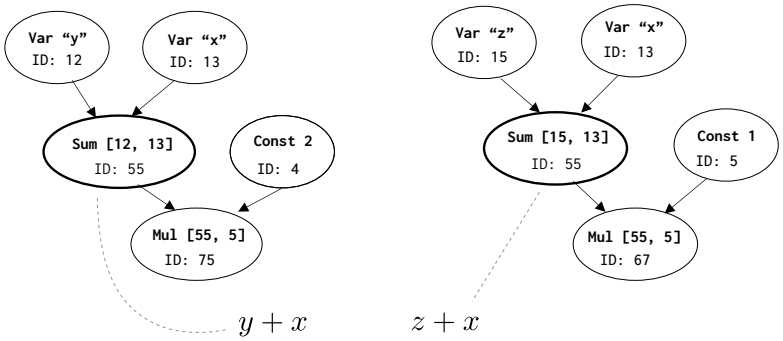


Figure 3.1: A hash conflict between two expressions graphs. The same index represents different symbolic expressions: $y + x$ on the left and $z + x$ on the right.

3.1.2 Handling Hash Collisions

Nodes must be added from one lookup table to the other lookup table one by one so that we can resolve hash collisions. Moreover, the order in which nodes are inserted matters. As node indices are subject to changes (because of rehashing), these changes must be reflected to all the nodes that depend on it. Failing to do so may result in operator nodes point to wrong operands, as shown in Figure 3.2.

Thus, we have to maintain two things when merging two expressions graphs:

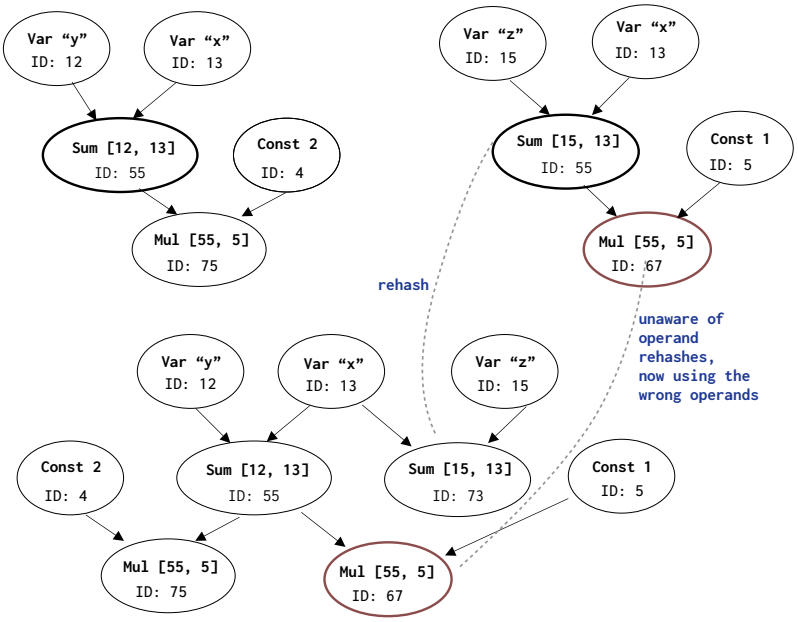


Figure 3.2: Node with index 67 ($(z+x)*1$) being inserted before node with index 55 ($z+x$), thus unware of rehashes and end up pointing to wrong operands

1. Nodes must be inserted in a topological order: from independent to dependent.

2. As node indices can be changed, we must maintain additional bookkeeping: a mapping from old node indices to new node indices (if there is a rehash unfolding) so that subsequent dependent nodes can update their content accordingly before being inserted into the lookup table.

The following code snippet shows this algorithm:

```
toTotal :: Map a a -> (a -> a)
toTotal sub x = case lookup sub x of
  Just v -> v
  _ -> x

mergeExpressionMaps :: ExpressionMap -> ExpressionMap ->
↪  (ExpressionMap, NodeID -> NodeID)
mergeExpressionMaps mp1 mp2 =
  let mp2TopoOrderEntries :: [(NodeID, Node)]
      mp2TopoOrderEntries = byTopoOrder mp2
      f ::
        (ExpressionMap, Map NodeID NodeID) ->
        (NodeID, Node) ->
        (ExpressionMap, Map NodeID NodeID)
      f (mp, sub) (oldID, oldNode) =
```

```
        let newNode = mapOperands (toTotal sub) oldNode -- update
         ↪  operands due to possible changes
            newID = index mp newNode
         in ( IntMap.insert newID newNode mp,
              if newID == oldID
                then sub
                else Map.insert oldID newID sub
            )
      (resultMp, sub) = foldl f (mp1, Map.empty)
       ↪  mp2TopoOrderEntries
    in (resultMp, toTotal sub)
```

With this, we can resolve the situation seen in Figure 3.2 resulting in a correct merge as illustrated in Figure 3.3.

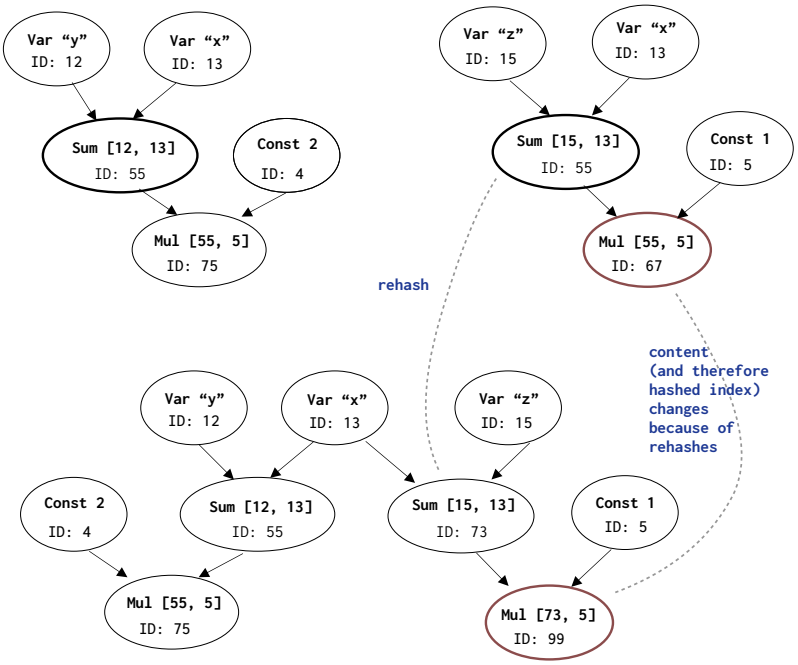


Figure 3.3: The correct merge: insert nodes in a topological order and keep track of rehashes

The time complexity of this function is $O(\min(m, n) \log(m + n))$. We can easily extend this function to take a list of `ExpressionMap`. In this case, the overall time complexity will be $O(n \log(n))$ where n is the total number of nodes in all lookup tables. So this has an extra logarithm factor compared to the naive approach.

Then, we can re-implement the addition function as:

```
add :: RawExpr -> RawExpr -> RawExpr
add (mp1, xID) (mp2, yID) =
  let (mp, sub) = mergeExpressionMaps mp1 mp2
      xIDupdated = sub xID
```

```
      yIDupdated = sub yID
      node = createSumNode [(mp, xIDupdated), (mp, yIDupdated)]
      nID  = index mp node
  in (IntMap.insert nID mp, NodeID nID)
```

3.1.3 Problems with Direct Method

While the discussed method provides a safe way for us to merge different expression graphs into one and, from there, construct new expressions, it fails to take into account the case when expressions share the same lookup table. Using the direct method, we will end up merging the same lookup table to itself unnecessarily. Although we can perform a comparison and only merge the two if they are different, this equality check still takes linear time and therefore does not solve the issue of redundant computation. This does not scale well, especially for term rewriting and computing derivatives where we'll mostly operate on the same expression graph.

3.2 Composing with Monad

3.2.1 An Alternative Type

Our goal is to find an alternative data structure not just for forming new expressions, but also for rewriting existing expressions and computing the derivatives. We observe the following things:

1. There should be a single underlying common lookup table. In an immutable and functional programming environment like Haskell, this means there must be a notion of binding the result of previous computation to the next computation.

2. Nodes and their hash indices must be introduced to this lookup table sequentially, so that any hash collision is resolved on the spot, not requiring any extra processing later. Thus, the data structure should be the *computation* to yield the node index, not the node index itself.

This suggests: the data structure is best embedded in a Monadic structure [64]. Instead of representing expressions concretely by:

```
type RawExpr = (ExpressionMap, NodeID)
```

We use:

```
type ExprBuilder = ExpressionMap -> (ExpressionMap, NodeID)
```

That is, instead of composing concrete instances of expressions, we compose the builders which would yield such expressions. The argument `ExpressionMap` denotes the underlying lookup table upon which this expression is built, and the result `ExpressionMap` denotes the result lookup table after the expression is built. This duo helps us address concern (1). For constructing a new expression,

the argument would start as an empty map. For rewriting or generating partial derivatives from an expression, the argument would be the existing lookup table.

For composing expressions (or builders of such in this case) and to address concern (2), we have the `bind` function:

```
-- bind ::
--    (ExpressionMap -> (ExpressionMap, NodeID))
-- -> (NodeID -> (ExpressionMap -> (ExpressionMap, NodeID)))
-- -> (ExpressionMap -> (ExpressionMap, NodeID))
bind :: ExprBuilder -> (NodeID -> ExprBuilder) -> ExprBuilder
bind builder1 mkBuilder2 mp =
  let (afterExpr1Mp, nID) = builder1 mp
      builder2 = mkBuilder2 nID
   in builder2 afterExpr1Mp
```

As such, when composing expressions we can be certain that expressions are built (nodes are added) in a given order: first builder (first argument) then the second builder, where the second builder can be constructed from the result (`NodeID`) of the first.

3.2.2 The State Monad

In fact, the types we are using are special cases of the `State` monad with state of type `ExpressionMap` and result of type `NodeID`, and our `bind` is actually the monadic bind `(>>=)`. Thus, we can alias our `ExprBuilder` to an underlying monad wrapper of the `State` monad:

```
-- the existing State monad
newtype State s a = State {runState :: s -> (a, s)}

instance Monad (State s) where
  (>>=) :: State s a -> (a -> State s b) -> State s b
  (>>=) = ...

-- our Builder monad
newtype Builder a
  = Builder (State ExpressionMap a)
  deriving (Functor, Applicative, Monad)

type ExprBuilder = Builder NodeID

buildExpr :: ExprBuilder -> RawExpr
buildExpr (Builder exB) = swap $ runState exB IM.empty
```

We will have other monads similar to `Builder` for term rewriting and computing derivatives. Although the monads wrap around the `State` monad, we don't want to expose the full range of operations allowed by the type class `MonadState` - which would permit arbitrary changes of state. Instead, what we can do with our wrapper monads is expressed by the type class `MonadExpression`:

Within the monad, we can:

```haskell
class (Monad m) => MonadExpression m where
  introduceNode :: Node -> m NodeID
  getContextMap :: m ExpressionMap
```

Listing 3: The `MonadExpression` type class

1. Introduce a new expression to the lookup table using `introduceNode`.
2. Get the information of the underlying lookup table using `getContextMap`.
3. Composing expression builders (or manipulators) sequentially with the monadic bind `(>>=)`.

A `MonadExpression` instance for our monads can be easily provided, as they are just a wrapper around the `State` monad, for example with `Builder`:

```haskell
instance MonadExpression Builder where
  introduceNode node = Builder $ do
    mp <- get
    let nID = index mp mp
    modify' $ IM.insert nID node
    return (NodeID nID)

  getContextMap = Builder $ get
```

Finally, we can elegantly write the addition function discussed earlier using the do notation:

```haskell
add :: MonadExpression m => m NodeID -> m NodeID -> m NodeID
add operand1 operand2 = do
  xID <- operand1
  yID <- operand2
  mp <- getContextMap
  let node = createSumNode [(mp, xID), (mp, yID)]
  introduceNode node
```

And we can use addition for any instance of `MonadExpression`, e.g., `Builder`:

```haskell
-- ExprBuilder = Builder NodeID
(+) :: ExprBuilder -> ExprBuilder -> ExprBuilder
(+) = add
```

The monad abstraction gives developers a solid framework for manipulating expressions. Expressions are not composed directly, but the computations yielding them are composed. Only the final composition is executed to form an actual expression graph, avoiding the need for dealing with separate lookup tables and hash collisions between them. Nodes are introduced sequentially, and each use of `introduceNode` costs $O(\log(n))$ time. Thus, the overall complexity of constructing a new expression is $O(n \log(n))$ — the same as the direct method.

Chapter 4

Modeling APIs

Statically typed programming languages usually come with support for generic programming [50] allowing us to manipulate values at the type-level that are evaluated and verified at compile time. This functionality ranges from simple uses of polymorphic containers in C++, Java to more powerful type and constraint combinators found in Typescript or Rust. In Haskell, the type system is even more powerful: we can program on the Haskell type-level much the same as we would program on the term-level. Type-level programming in Haskell provides some functionality used for certified programming in dependently typed languages like Agda [52] or Idris [7], while at the same time keeping the features simple enough to be embedded in a general-purpose programming environment.

Haskell's support for type-level programming is one of the main reasons why we chose it as the host language for our algebraic modeling language. Our goal is to implement a type-safe interface for expressing optimization problems. It is similar to how Servant [45] introduced type-safety to Web API definitions, or how the Mezza [19] music description library enforces the rules of music composition on the type level. In this chapter, we will discuss how Haskell helps us accomplish this by allowing us to lift expression metadata into the type-level upon which we can express operator specifications and data constraints.

4.1 Type-Level Programming in Haskell

This section gives a brief overview of type-level programming in Haskell.

4.1.1 Kinds

The Haskell type system uses the denotational semantics of a simply typed lambda calculus system with "one order up", allowing for the denotation of the "type of a type" (colloquially referred to as a "kind" in the Haskell community). So similar to how everything has a type on the term-level, such as `True :: Bool` or `(+1) :: Int -> Int`, all the entities that exists on the type-level (we will refer to these as "type-level values") have a kind. We can inspect the kind of a type in GHCi using the `:k` command:

```
>:k Bool
Bool :: Type

>:k Maybe
Maybe :: Type -> Type

>:k Maybe Int
Maybe Int :: Type

>:k Monad
Monad :: (Type -> Type) -> Constraint
```

As with types, kinds can be composed with the `(->)` operator. Most conventional kinds will be nothing more than compositions of the built-in kind `Type`. A more interesting kind may be composed with another built-in kind known as `Constraint`, which is a special kind used for expressing conditions for functions and types in general. Generally, a type level value yielding a `Constraint` kind is declared in a type class definition. For example:

```
class Ord a where
  compare :: a -> a -> Ordering

sort :: Ord a => [a] -> [a]
```

`Ord` is said to have kind `Type -> Constraint`. When an invocation of `compare` or `sort` on a type-level value `a` is type-checked, an instance of `Ord a` must exist for the type-check to succeed.

In addition to built-in kinds like `Type` and `Constraint`, the Haskell compiler GHC also comes with an extension `DataKinds` [68] that, when enabled, allows us to *promote* term-level declarations to the type level. That is, with the extension enabled if we define:

```
data Direction = Forward | Backward
```

besides creating the type `Direction` inhabited by 2 values `Forward` and `Backward`, we also get the kind `Direction` inhabited by 2 type-level values `'Forward` and `'Backward` (the prefix tick is not required as long as no type with the same name exists in the name space).

```
>:t Forward
Forward :: Direction -- Direction as type

>:k 'Forward
'Forward :: Direction -- Direction is also a kind thanks to DataKinds
```

GHC with `DataKinds` also has built-in literal support for natural numbers (kind `Nat`) and string (kind `Symbol`):

```
>:k 42
42 :: Nat

>:k "hello world"
"hello world" :: Symbol
```

4.1.2 Type Families

Kinds give us a means for classifying type-level values in the same way that types are used to classify term-level values. For expressive type-level programming, we also need the ability to operate on kinds. In GHC, we do so by using type families [21] - which can be thought of as functions at the type-level. For example, the reverse direction function normally expressed at term-level:

```
reverseDirection :: Direction -> Direction
reverseDirection Forward = Backward
reverseDirection Backward = Forward
```

can be expressed at the type-level as a type family:

```
type family ReverseDirection (d :: Direction) :: Direction where
  ReverseDirection 'Forward = 'Backward
  ReverseDirection 'Backward  = 'Forward
```

With kinds and type families, the type system of Haskell is as computationally expressive as a mini programming language[1]. Even better, GHC exports a number of type families to deal with natural numbers such as comparison `CmpNat :: Nat -> Nat -> ORD`, addition `(+) :: Nat -> Nat -> Nat` and modulus (`(Mod :: Nat -> Nat -> Nat)`) which are necessary when dealing with expression shapes. Type families also allows *kind polymorphism* so we can have definitions abstracted in their kind parameters, for example:

```
type family Length (list :: [a]) :: Nat where
  Length '[] = 0
  Length (x ': xs) = 1 + Length xs
```

There is also support for unsaturated type family application [36] scheduled for future GHC versions.

4.1.3 Custom Type Errors

Kinds and type families can be used to encode constraints on the type level so that invalid programs would result in a compile error. However, the error content matters. We want the error to be an informative message that reflects the nature of program domains rather than a generic GHC error. This is even more desirable for our use case: an embedded modeling language. To resolve this, GHC provides a special built-in, type-level function:

```
type family TypeError (msg :: ErrorMessage) :: k
-- ErrorMessage is intended to be used as a kind
data ErrorMessage =
      Text Symbol                       -- Show this text as is
    | forall t. ShowType t              -- Pretty print a type
```

[1] The type system with the `UndecidableInstances` extension enabled is Turing-complete, see `https://mail.haskell.org/pipermail/haskell/2006-August/018355.html`

```
  | ErrorMessage :<>: ErrorMessage      -- Put two chunks of error
  ↪  message next to each other
  | ErrorMessage :$$: ErrorMessage      -- Put two chunks of error
  ↪  message above each other
```

That is, the type family `TypeError` takes a value of kind `ErrorMessage` and produces a bottom type-level value that can be used as any kind. This can be put in error cases in our type-level functions, and GHC will reflect the error message if the program falls into such cases. In the next sections, we will see how custom compile errors can be used to provide meaningful error messages.

4.2 A Type-safe Modeling API

This section will detail how we exploit Haskell's type system to provide type-safety in our embedded algebraic modeling language. Type-safety will help catch "would-be" correctness bugs at compile time by constraining how expressions are used and constructed in a given context.

4.2.1 Lifting Information To The Type-level

Recall that our data type to represent an expression node is defined as:

```
data ElementType = R | C

type Shape = [Int]

type Node = (Shape, ElementType, Op)
```

and the type for constructing expressions is:

```
type ExprBuilder = Builder NodeID
```

`Node`s contain `Shape` and `ElementType` metadata. Because each operation has a specification of dimension and element type of its inputs and output, this metadata is important to constrain expression composition. However, the type for building expressions does not distinguish shape and element type. For the type system to help with governing constraints and ensure the correctness of expression constructions, we must lift these pieces of information to the type-level. To do so, we wrap the existing type inside a new type parameterized by two phantom type parameters [13]:

```
newtype TypedExpr (sh :: [Nat]) (et :: ElementType)
  = TypedExpr ExprBuilder
```

- The first parameter (kind `[Nat]`) holds shape information. This kind is inhabited by type-level values, e.g. `'[]` represents a scalar, `'[128, 128]` represents a 2D shape of 128×128, etc.

- The second parameter - `ElementType` is the kind promoted from the same type we are using in the term-level and is inhabited by 2 type-level values `'R` and `'C` (we can omit the prefix tick).

`TypedExpr` has kind `[Nat] -> ElementType -> Type`. A saturated type-level value of `TypedExpr` denotes the type of expressions with given shape and element type. For instance:

```
TypedExpr '[15, 15, 15] C -- or TypedExpr '[15, 15, 15] 'C
```

is a 3D expression of shape $15 \times 15 \times 15$, and its elements are complex numbers.

Note that such parameters are called *phantom types* because they only appear on the left side of the type definition. Using phantom types is a common practice to decorate existing non-parametric types, allowing us to keep a somewhat low-level representation of expressions built on graphs, indices, and monads, while at the same time having metadata on the type-level. The data constructor `TypedExpr` is hidden from users to prevent arbitrarily coercing between expression types. Instead, users can create primitive expressions, i.e., variables, parameters, and constants, via predefined functions such as:

```
variable2D :: forall m n. (KnownNat m, KnownNat n) => String ->
↪  TypedExpr '[m, n] R
variable2D name = TypedExpr $ introduceNode (toShape @'[m, n], R,
↪  Var name)

-- define a variable
x = variable2D @20 @15 "x" -- TypedExpr '[20, 15] R
```

where the `@` sign denotes a type application, allowing us to provide type-level natural number argument to kind-polymorphic functions like `variable2D`.

4.2.2 Operator Specification

Using `TypedExpr`, we can now define mathematical operators that constrain the shape and element type of their inputs/output. By providing the same type parameter or explicit type values to the phantom type parameters in `TypedExpr` we can constrain which operands must share the same shape or use a specific element type, i.e.,

```
-- element-wise addition
(+) :: TypedExpr sh et -> TypedExpr sh et -> TypedExpr sh et

-- element-wise complex expression from real and imaginary parts
(+:) :: TypedExpr sh R -> TypedExpr sh R -> TypedExpr sh C

-- element-wise exp
exp :: TypedExpr sh R -> TypedExpr sh R

-- Fourier transform
ft :: TypedExpr sh C -> TypedExpr sh C
```

Listing 4: Example of specifications in type signatures

The examples are self-explanatory: `(+)` takes and returns expressions of the same shape and element type, and the Fourier transform `ft` only accepts a complex expression and returns a complex expression with the same shape.

Note that as `TypedExpr` is a phantom type wrapper around `ExprBuilder`, we only have to focus on type signatures. Implementation of such functions is carried out with the monadic construction previously discussed in 3.2, though some extra wrapping and unwrapping is required.

With the given type signature, adding expressions with mismatched shape will yield a compile error:

```
> x = variable2D @10 @10 "x" -- TypedExpr '[10, 10] R
> y = variable2D @20 @10 "y" -- TypedExpr '[20, 10] R
> z = x + y

• Couldn't match type ‘20’ with ‘10’
Expected type: TypedExpr '[10, 10] R
  Actual type: TypedExpr '[20, 10] R
```

More complicated functions may involve type families. Take the scale operator in vector spaces as an example: the first operand is a scalar, the second operand is a vector, and the operator returns a vector (the same type as the second operand). This can be encoded as:

```
(*.) ::
  (Scalable etScalar etVector) =>
  TypedExpr Scalar etScalar ->
  TypedExpr sh etVector ->
  TypedExpr sh etVector
```

The constraint `Scalable` checks on the compatibility of scalar and vector's type. As we know, both $\mathbb{R}^n$ and $\mathbb{C}^n$ are vector spaces over $\mathbb{R}$, but only $\mathbb{C}^n$ is a vector space over $\mathbb{C}$:

```
type family Scalable (etScalar :: ElementType) (etVector ::
↪  ElementType) :: Constraint where
  Scalable R e = Satisfied  -- Real scalar can scale both real and
  ↪  complex vectors
  Scalable C C = Satisfied  -- Complex scalar can only scale
  ↪  complex vectors
  Scalable C R = TypeError (Text "Scaling a real vector by a
  ↪  complex scalar is not allowed")
```

This is reflected in GHCi:

```
> x = variable2D @10 @10 "x" -- TypedExpr '[10, 10] R
> y = variable2D @10 @10 "y" -- TypedExpr '[10, 10] R
> s1 = variable "s1" -- TypedExpr Scalar R
> s2 = variable "s1" -- TypedExpr Scalar R
> s = s1 +: s2  -- TypedExpr Scalar C
> z = s1 *. (x +: y) -- OK, TypedExpr '[10, 10] C
> z2 = s *. x -- Type Error

• Scaling a real vector by a complex scalar is not allowed
• In the expression: s *. x
  In an equation for ‘z2’: z2 = s *. x
```

For functions that require computation and constraints on expression shapes, more type-level programming is required. For example, the projection operator involves a custom data kind and a type family for computing the resulting shape:

```
data Selector
  = All
  | Range
      Nat -- start
      Nat -- end
      Nat -- step
  | At Nat -- position

project ::
  IsSelectors selectors =>
  TypedExpr inputShape et ->
  TypedExpr (ProjectionShape inputShape selectors) et
```

where `ProjectionShape` with kind `[Nat] -> [Selector] -> [Nat]` calculates the output shape from the input shape and selectors. The type family also verifies the validity of indices, e.g., positions must be within the dimension range. Detailed implementation of `ProjectionShape` can be found in Appendix A. Projection gives us a type-safe construction similar to Python's slicing notation:

```
x = variable2D @20 @30
-- y = x[0:9, 0:5], has type TypedExpr '[10, 6] R
```

```
y = project @'[Range 0 9 1, Range 0 5 1] x
-- z = x[1, 5], has type TypedExpr Scalar R
z = project @'[At 1, At 5] x
-- t = x[1:2, :], has type TypedExpr '[2, 30] R
t = project @'[Range 1 2 1, All] x
```

Invalid selectors will result in compile errors:

```
x = variable2D @20 @30
-- Type Error: end index 30 is out of range in the 1st dimension
y = project @'[Range 10 30 1, At 5] x

• Invalid end index 30, must be in range (0, 20)
```

With operator constraints and specifications encoded at the type-level, expressions are therefore correct by construction. Enabling a type-safe API for constructing optimization problems can be done similarly by specifying the types of problem components:

1. The objective expression must be a scalar real expression.
2. A general constraint (see equation 1.1) must be a scalar real expression and take a `Double` value as lower or upper bound.
3. Box constraints require the bounded variable and the bound to have the same shape, as illustrated in example 1.4.

4.3 Physical Units

Problems of manipulating expressions with incompatible shape or element type are solved by encoding such information at the type-level. However, there is still another source of errors in scientific applications that concerns physicists and mathematicians: incompatible physical units. Users can still, for example, construct different expressions that are both real and have the same shape but were meant to represent different physical signals with disparate units. Adding them, though allowed in this case, is considered invalid. In this view, expressions are not simply grids of numbers but are instances representing actual physical signals. A common interpretation in scientific optimization models, especially in medical imaging, views expressions as the discretization of real-world physical signals [47] which entails not only dimensional data but also information about physical units and sampling.

Our goal is to create another interface with a new set of operations to form expressions that provide discretization metadata. Doing so first requires information to be encoded at the type-level.

Physical units The use of a type system to correctly handle physical units in scientific applications has been around for quite some time [49] with mature packages dedicated for type-level physical units[2]. We take an approach similar

[2]For a list, see `https://wiki.haskell.org/Physical_units`

to the `Dimensional` package of representing physical units as combinations of seven base units in the International System of Units (SI): the meter (m), the kilogram (kg), the second (s), the kelvin (K), the ampere (A), the mole (mol), and the candela (cd).

```
-- unit is intended to be used as a kind
data Unit
  = Unit
      UMeter     -- (length, m - meter)
      UKilogram -- (mass, kg - kilogram)
      USecond    -- (time, s - second)
      UAmpere    -- (electric current, A - ampere)
      UKelvin    -- (thermodynamic temperature, K - kelvin)
      UMole      -- (amount of substance, Mol - mole)
      UCandela  -- (luminous intensity, cd - candela)

-- TypeInt represents type-level integers
newtype UMeter = M TypeInt
newtype UKilogram = Kg TypeInt
...
```

Sampling step size Ideally, the sampling step should be encoded as floating-point numbers, but floating-point type-level literals are not yet supported in GHC. Although we can resort to `TemplateHaskell`, we found that the complication outweighs the benefit. Instead, we use positive fractional numbers formed by a pair of natural numbers to represent the sampling step. This also means we will need to perform some extra type-level computation to bring them to the irreducible form.

```
data SampleStep = Nat :/: Nat

-- (:/) is used instead (:/:) to check for zero division
-- and bring the fraction to canonical form
type family a :/ b where
  a :/ 0 = TypeError (Text "Division to zero: " :<>: ShowType a
  ↪  :<>: Text "/0")
  0 :/ b = TypeError (Text "Sampling step must be positive: " :<>:
  ↪  Text "0/" :<>: ShowType b)
  a :/ b = (a `N.Div` (GCD a b)) :/: (b `N.Div` (GCD a b))
```

We again use the phantom type parameter technique to form a new expression type that represents the discretization of physical signals. Dimensional metadata is not just a list of natural numbers `[Nat]` but a list of sampling information for each dimension: number of samples, the sampling step, and the domain unit along which values are sampled.

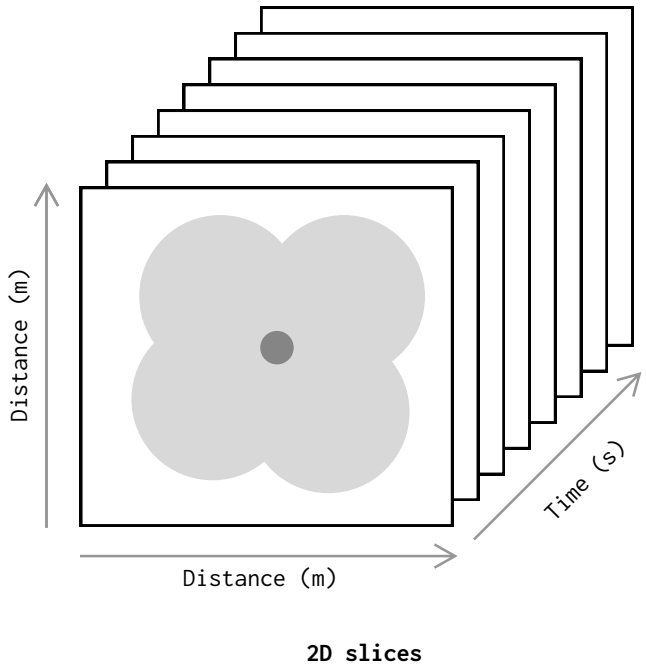


Figure 4.1: Different sampling units along different dimensions in a discretization

```
data Sampling
  = D
      Nat        -- number of samples
      SampleStep -- sampling steps
      Unit       -- domain unit

newtype
  UnitExpr
    (ds :: [Sampling])
    (unit :: Unit)      -- range unit
    (et :: ElementType) -- real or copmlex
  = UnitExpr ExprBuilder
```

As shown in Figure 4.1, it is possible to have different samplings along different dimensions. A variable representing this discretization can be:

```
v =
  variable
    @'[ D 256 (1 :/ 2000) Meter,    -- 1st dimension: distance
        D 256 (1 :/ 2000) Meter,    -- 2nd dimension: distance
        D 100 (1 :/ 60) Second      -- 3rd dimension: time
      ]
    @Candela                        -- range unit: luminance
    "x"
```

With physical units metadata available on the type-level, we can precisely formulate point-wise multiplication: the resulting unit should be the product of its operands' units.

```
-- x has type UnitExpr '[D 10 (1 ':/: 2000) Meter] Kilogram R
x = variable @'[D 10 (1 :/ 2000) Meter] @Kilogram "x"

-- x has type UnitExpr '[D 10 (1 ':/: 2000) Meter] Candela R
y = variable @'[D 10 (1 :/ 2000) Meter] @Candela "y"

-- z has type UnitExpr '[ 'D 10 (1 ':/: 2000) (Meter :*: Candela)]
↪  Kilogram 'R
z = x * y
```

Similarly, the Fourier transform keeps range units but acts on domain units and sampling steps. A detailed analysis can be found in [47].

```
-- x has type UnitExpr '[D 10 (1 :/: 2000) Meter] Ampere R
x = variable @'[D 10 (1 :/ 2000) Meter] @Ampere "x"

-- y has type UnitExpr '[D 10 (1 :/: 2000) Meter] Ampere R
y = variable @'[D 10 (1 :/ 2000) Meter] @Ampere "y"

-- z has type UnitExpr '[D 10 (200 :/: 1) (Meter :^: (Negative 1))]
↪  Ampere R
z = dft $ (x +: y)
```

4.4 Type-level or Term-level

There are trade-offs that must be considered when determining whether to encode metadata at the type-level or not. In this section we will discuss some of the drawbacks to encoding data at the type-level and an instance where we judged term-level encoding to be more appropriate.

4.4.1 Type-level Value Declarations

While having metadata information available in the type level is useful in many settings, it also comes with a drawback that should not be overlooked: type-level values must be known at compile-time, or otherwise functions and computations must be accompanied by corresponding type-level constraints.

For instance, by using projection, we can drop the first/last element of a one dimensional vector to form new expressions. If they have the same shape and element type, they can be added together:

```
problem :: OptimizationProblem
problem =
  let
    x = variable1D @10 "x"        -- TypedExpr 10 R
```

```
    y = project @'[Range 1 9 1] x -- TypedExpr 9 R
    z = project @'[Range 0 8 1] x -- TypedExpr 9 R
    t = y + z .                   -- TypedExpr 9 R
  in
    ...
```

All the type-level natural numbers are manually provided in this case, which can become tedious. Suppose we want to generalize the code on the size of variable x instead of a fixed number 10. Our suggested way to accomplish this is by type-level declarations:

```
type N = 10

problem :: OptimizationProblem
problem =
  let
    x = variable1D @N "x"                -- TypedExpr 10 R
    y = project @'[Range 1 (N - 1) 1] x -- TypedExpr 9 R
    z = project @'[Range 0 (N - 2) 1] x -- TypedExpr 9 R
    t = y + z                            -- TypedExpr 9 R
  in
    ...
```

As long as `N` is known (which is 10), the type system can compute the exact final shape of y and z and decide if we are allowed to take the sum of them.

4.4.2 Unknown Type-level Values

However, if the value of x's size is only an unspecified type-level natural value:

```
problem :: forall n. (KnownNat n) => OptimizationProblem
problem =
  let x = variable1D @n "x" -- TypedExpr n R
      -- y :: TypedExpr (ProjectionShape '[n] '[Range 1 (n - 1) 1])
      ↪  R
      y = project @'[Range 1 (n - 1) 1] x
      -- z :: TypedExpr (ProjectionShape '[n] '[Range 0 (n - 2) 1])
      ↪  R
      z = project @'[Range 0 (n - 2) 1] x
      t = y + z -- COMPILE ERROR
   in
      ....
```

Then the type system cannot verify that the shape of `y` and `z` are in fact the same (as long as `n > 1`), and will yield a compile error indicate that we can not perform the operation `y + z`. In this case, we must accompany the "proof" (in form of constraints) that their shape should be identical in the problem's type signature:

```
problem ::
  forall n.
  ( KnownNat n,
    0 < n,
    (n - 2) < n,
    KnownNat ((n + (n - 2)) `Mod` n + 1),
    ProjectionShape '[n] '[ 'Range 1 (n - 1) 1]
      ~ ProjectionShape '[n] '[ 'Range 0 (n - 2) 1]
  ) =>
  OptimizationProblem
problem =
  let x = variable1D @n "x"
      y = project @'[Range 1 (n - 1) 1] x
      z = project @'[Range 0 (n - 2) 1] x
      t = y + z -- COMPILED
   in
      ...
```

4.4.3 Undecorated API

In such instances, the list of constraints we need to provide can grow very quickly and become too verbose to be expressed, and the more sensible approach to this problem is to use the undecorated type - expressions without metadata on the type-level. In this case, the type `Expr` is just an alias for `ExprBuilder`, and creating variables no longer requires type applications:

```
type Expr = ExprBuilder

variable :: Shape -> String -> Expr
variable shape name = introduceNode (shape, R, Var name)
```

Then, the optimization with unknown-shape variables can be written as:

```
problem :: Int -> OptimizationProblem
problem n =
  let
    x = variable [n] "x"                  -- Expr
    y = project (IntRange 1 (n - 1) 1) x -- Expr
    z = project (IntRange 0 (n - 2) 1) x -- Expr
    t = y + z                             -- Expr
  in
    ...
```

Of course, in this case, the compiler will not be able to spot mismatched shape (or element type) calculations. However, we can verify all of that when we run the program to construct the model. Note that this is still safe because the model is symbolic, and no actual evaluations are performed until the model is verified for code generation. Nevertheless, we will lose the interactivity with the Haskell type system.

Chapter 5

Rewriting and Simplification

Algebraic simplification aims to obtain equivalent but simpler expressions by using algebraic properties of mathematical operators and their combinations [10]. In the context of modeling languages for mathematical optimization, our chief criteria for discerning the effectiveness of a "simplification" is its reduction in evaluation complexity and memory consumption. Performing expression simplification will benefit the resulting expression evaluation by:

1. Reducing evaluation time by eliminating redundant computations. For instance, $1 * x$ can be rewritten to x to eliminate an unnecessary multiplication (modelers generally do not write expressions like $1 * x$ directly, but they can be introduced when computing derivatives or from auxiliary functions).

2. Replacing expressions that are algebraically equivalent to a single, common expression and therefore share the computation. For example, $x + (y + z)$ and $(x + y) + z$ can both be transformed to $sum(x, y, z)$ which will be indexed to a single expression node. This is common subexpression elimination via algebraic equivalence as discussed in Chapter 2.

Some optimization and modeling software performs expression simplification during expression evaluation, which can be considered a form of dynamic simplification. Depending on the value of the input, a code graph is derived; then computation reduction rules and evaluations are applied together, thus requiring simplification logic to take place in low-level code. For instance, TensorFlow [1] delegates algebraic simplification to the C++ graph optimizer Grappler [39]. Dynamic simplification can be a good tradeoff, introducing more overhead but with access to runtime values and hence the possibility of yielding better simplification. However, for a symbolic expression library that performs code generation like HashedExpression, static analysis and simplification are more appropriate. First, it allows us to transform expressions once beforehand and evaluate them multiple times later as opposed to the dynamic simplification where transformation must happen within a limited time constraint. Moreover, with symbolic expressions decoupled from actual numerical data, we can carry out simplifications in the library's host language Haskell. This is the basis for a sophisticated rewriting system, allowing us to look for higher-level patterns which would otherwise require expressive whole-program optimization.

5.1 Types

In this section, we will delve into the implementation of the type system for rewriting and simplification. We will also demonstrate how simplification rules are implemented, one of which is via a match-and-replace pattern language, as well as how simplification rules are composed and generalized to all nodes in the expression graph.

5.1.1 Instance of MonadExpression

Rewriting of expressions involves constructing new expressions, and as a direct consequence of our hashed indexing scheme, we must utilize the framework introduced in Chapter 2 (i.e., the type class `MonadExpression`) to create a convenient interface for performing rewrites. Similar to the `Builder` monad, we have a monad `Rewrite` wrapping around the `State` monad for rewriting and simplification. The only difference is the lookup table we feed to the final computation. For the `Builder` monad, it is the empty lookup table, and for the `Rewrite` monad, it is the original lookup table of the targeted expression.

```
newtype Rewrite a = Rewrite {unRewrite :: State ExpressionMap a}
  deriving (Functor, Applicative, Monad)

-- similar to the Builder monad
instance MonadExpression Rewrite where
  introduceNode node = Rewrite $ do
    mp <- get
    let nID = index mp node
    modify' $ IM.insert nID node
    return (NodeID nID)

  getContextMap = Rewrite $ get
```

Listing 5: The `Rewrite` monad for rewriting expressions.

The `Rewrite` monad shares expression composition logic with the `Builder` monad. The type for rewriting an expression is:

```
type Modification = NodeID -> Rewrite NodeID
```

We can read the type synonym as: take a node index (a pointer to an expression), and return another node index (pointer to the rewritten expression), where the computation happens in the `Rewrite` monad. In fact, because `State s a` is isomorphic to `s -> (s, a)`, the type synonym `Modification` is isomorphic to:

```
NodeID -> ExpressionMap -> (ExpressionMap, NodeID)
```

which is isomorphic to:

```
(ExpressionMap, NodeID) -> (ExpressionMap, NodeID)
```

In other words, `Modification` is isomorphic to a function that takes a type `RawExpr = (ExpressionMap, NodeID)` (an expression as identified by the underlying lookup table and a node index), and "transforms" it into another `RawExpr`.

However, just as we compose expressions by using `ExprBuilder` instead of `RawExpr`, we use `Modification` for rewriting instead of the endomorphism type `RawExpr -> RawExpr`. We make `Rewrite` an instance of `MonadExpression` which allows us to combine rewriting rules without having to deal with hash collisions and achieve nominal common subexpression elimination. This simple analysis again shows the powerful abstraction power of Haskell's type system.

5.1.2 Implementing Rewriting Rules

Now that we have a type to represent rewriting rules, i.e.,

```
type Modification = NodeID -> Rewrite NodeID
```

our task is encode rewrite rules as functions of this type. From this type signature, each rewrite rule will have access to:

1. An argument of type `NodeID`
2. The function `getContextMap :: Rewrite ExpressionMap` that provides the current `ExpressionMap` via the `MonadExpression` monad.

Taking this into consideration, we can develop rewrite rules in a fairly systematic fashion using the following steps:

1. Pattern match on targeted expression, inspecting any data by the given `NodeID` and underlying lookup table `ExpressionMap`
2. Check against any rewriting conditions
3. Construct the resulting expression (in the `Rewrite`) monad.

The example in Listing 6 illustrates how to implement a rewrite rule that simplifies expressions with zeros and ones that appear in a sum or a product. Note that in this example, `sum_` and `product_` are functions that form a sum or product of expressions defined for `MonadExpression` instances:

```
sum_     :: MonadExpression m => [m NodeID] -> m NodeID
product_ :: MonadExpression m => [m NodeID] -> m NodeID
```

```haskell
zeroOneSumProdRule :: Modification
zeroOneSumProdRule nID = withExpressionMap $ \mp ->
  case retrieveOp nID mp of
    Sum ns
      -- remove 0s from sum
      -- sum(x, y, z, 0, t, 0) -> sum(x, y, z, t)
      | (x : _, []) <- partition (isZero mp) ns -> return x
      | (_, nonZeros) <- partition (isZero mp) ns -> sum_ . map
      ↪  return $ nonZeros
    Mul ns
      -- remove 1s from product
      -- product(x, y, z, 1, t, 1) -> product(x, y, z, t)
      | (x : _, []) <- partition (isOne mp) ns -> return x
      | (_, nonOnes) <- partition (isOne mp) ns -> product_ . map
      ↪  return $ nonOnes
      -- if contains a 0, collapse to 0
      -- product(x, y, z, 0, t, u, v) -> 0
      | nId : _ <- filter (isZero mp) ns -> return nId
    _ -> return nID
```

Listing 6: A simplification rule written in function form.

5.2 Matching and replacing

The steps mentioned in the previous section should provide a general outline suitable to implement any simplification and rewriting rules. However, we want to take it one step further by deriving a system for rewriting based on match-and-replace with a syntax that closely resembles that used by mathematicians. Unlike the traditional Haskell pattern matching notation, it allows us to match expressions with symbols that could appear more than once. Such repeated appearances indicate they should be matched by identical expressions.

For instance, the following function expresses the rule that $\Re(a+b)(a-bi)$ should be simplified to $(a*a+b*b)$. (Note that this is regarded as a simplification rule because the right-hand side is less computationally expensive)

```haskell
simpRule :: Modification
simpRule = fm $ xRe ((a +: b) * (a +: negate b)) |.~~> (a * a + b *
↪  b)
```

We implement the match-and-replace rewriting system by having a data type `Pattern` that reflects the structure of the expressions.

```haskell
-- similar to Op
data Pattern
  = PHole Capture
  | PConst Double
  | PSum [Pattern]
```

```
  | PMul [Pattern]
  | PNeg Pattern
  | ...
```

`Pattern` has similar constructor names and is used to match with data type `Op` discussed in chapter 2. However, `Pattern` does not take into account common expression sharing and therefore has a straightforward tree structure. The custom data type also has a special constructor `PHole` representing symbols used to match with underlying subexpressions. The corresponding pattern value of the above simplifications left-hand side is:

```
-- xRe ((a +: b) * (a +: negate b))
PRealPart
  ( PMul
      (PRealImag (PHole "a") (PHole "b"))
      (PRealImag (PHole "a") (PNeg (PHole "b")))
  )
```

Applying a rewriting rule by matching and replacing consists of 2 main steps:

1. First, the targeted expression is matched with the left-hand side of the rewriting rules to produce a substitution. This is aided by the fact that identical expressions can be verified in constant time $O(1)$ (simply by comparing their hashed indices). Figure 5.1 shows an example of a successful substitution.

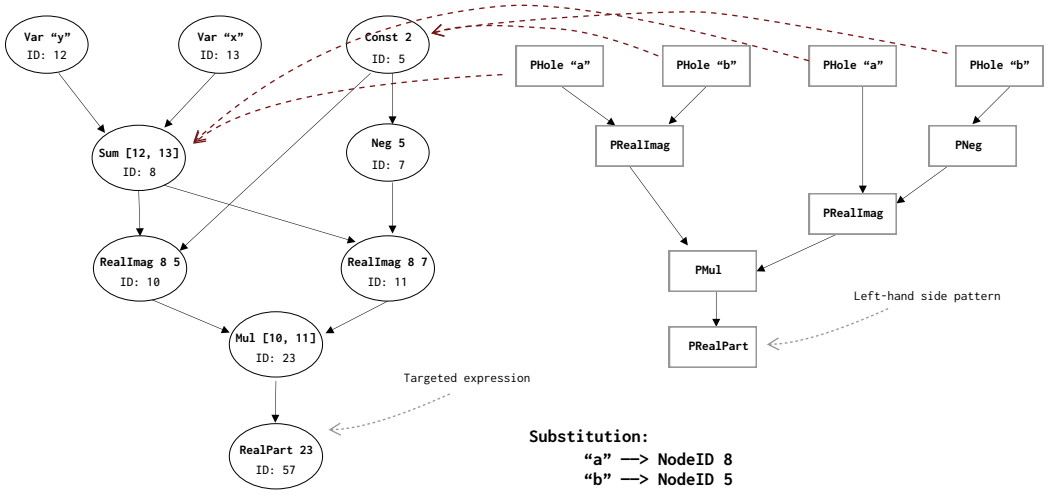


Figure 5.1: A successful match for the targeted expression $\Re((x+y)+2i)((x+y)-2i)$.

2. Then, the substitution is applied to the right-hand side pattern to construct the result expression, as shown in Figure 5.2.

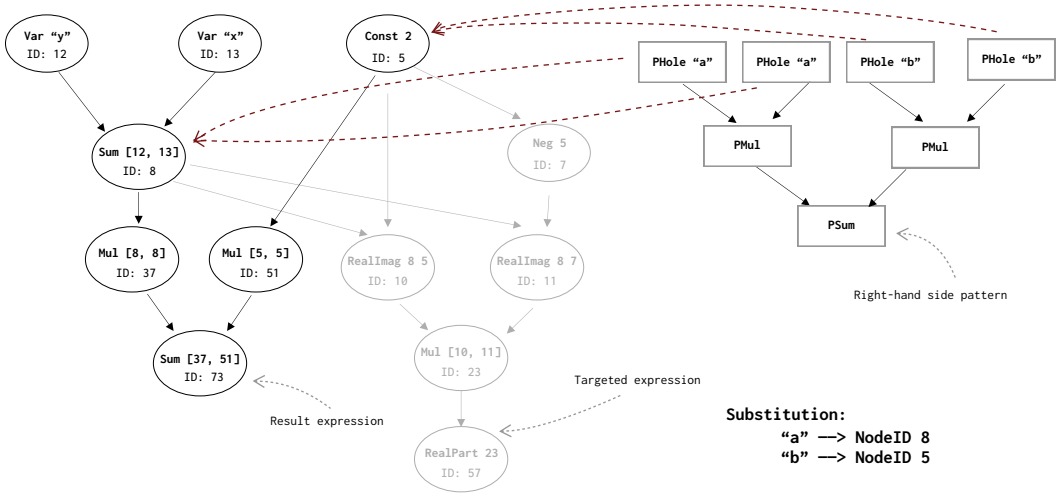


Figure 5.2: Construct resulted expression from the substitution and right-hand side pattern

We also allow a guarded style to construct rewriting rules similar to Haskell guard's to make the system more expressive, e.g:

```
rule :: Modification
rule = fm $ x *. (y *. z) |. sameElementType [x, y] ~~> (x * y) *.
↪  z
```

All in all, we find this extra rewriting subsystem brings several advantages. As library developers, we find it convenient to write algebraic simplification rules. Moreover, by providing a syntax that is similar to mathematical notations, it is possible to allow domain experts to provide such high-level performance-enhancing transformations together with their optimization models.

5.3 Composing and Generalizing

5.3.1 Composing

Composing rewriting rules is relatively straightforward. This is accomplished by a single fold:

```
chainModifications :: [Modification] -> Modification
chainModifications = foldr (>=>) return
```

where

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

is a left-to-right composition of two functions under a monadic context.

5.3.2 Generalizing

We can combine simplification rules using a simple fold and a monadic composition, but we also want such combined rules to apply to all nodes in the expression graph rather than a single individual expression node. As such, we need a function that turns a rewriting rule into a transformation that accepts an input expression graph and returns another expression graph where the rewriting rule has been applied to all nodes. Moreover, due to the dependency-structure of the expression graph, we also need to keep track of the changes that occur. In other words, given a node index in the old graph, we want to know the corresponding node index in the rewritten graph:

```
generalizeModification ::
  Modification ->
  ExpressionMap ->
  (ExpressionMap, NodeID -> NodeID)
```

The traversal order matters, and rewriting changes in any node must be reflected to nodes dependent on it. Because of this, `generalizeModification` is a fold over the nodes sorted in topological order - from independent to dependent. Upon applying a fold on a node:

1. First, operands of the current node are updated if they have been rewritten.
2. Then, apply the rewriting rules to the node with rewritten operands.
3. If the node has changed as a result of these steps, add a new pair to the update list.

The process is illustrated in Figure 5.3, 5.4 and 5.5. First, the node representing the symbolic expression $x * 1$ is rewritten to the node x.

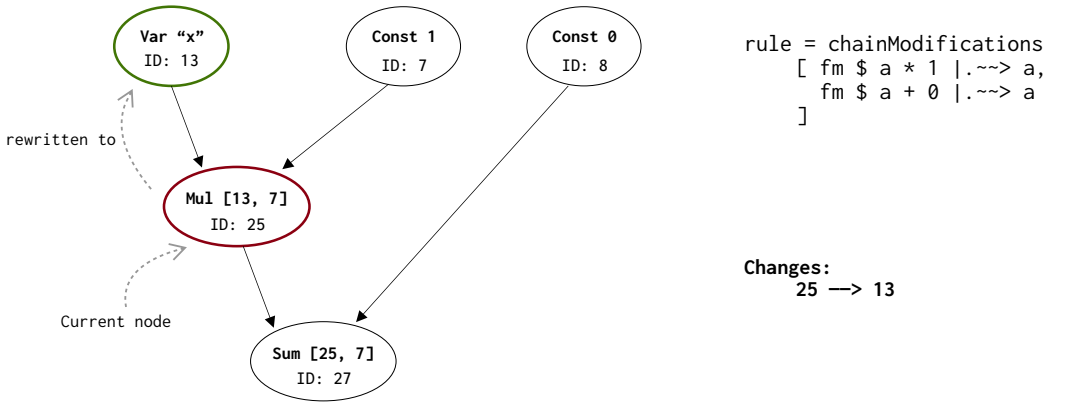


Figure 5.3: Expression node is simplified, and the change `25 -> 13` is added

Moving on to the next node in the topological order, the node representing $(x*1)+0$ finds its operands modified, and updates itself to the new node $x+0$.

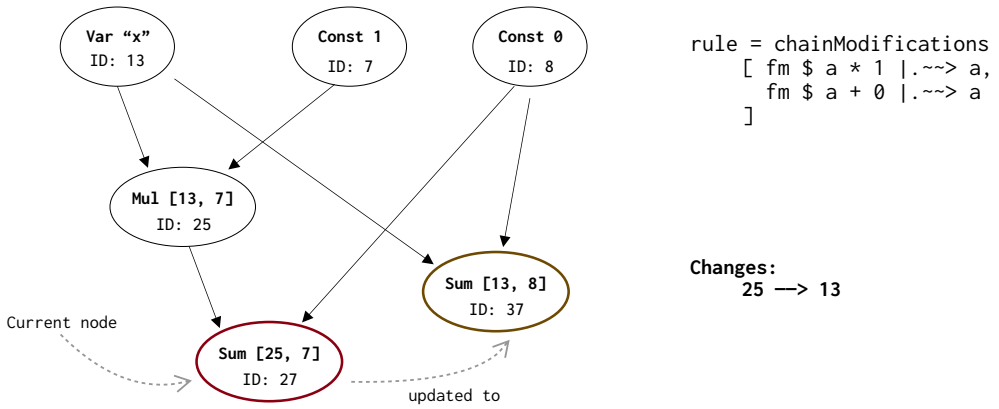


Figure 5.4: Expression node is updated based on the changes of operands

Then, the updated node is rewritten to the node x, and the change is reflected in the accumulated changes.

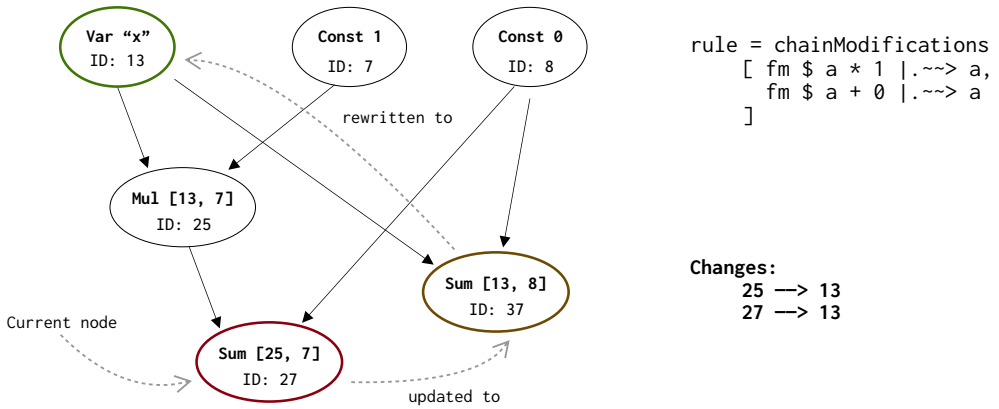


Figure 5.5: Updated node is simplified, and the change `27 -> 13` is added

5.4 Semantics-preserving, Confluence, and Termination

In the context of an algebraic rewriting system, three properties of interest are:

1. Semantics-preserving: whether rewriting rules preserve the mathematical meaning of input expressions, i.e., the rewritten expression should evaluate to the same result as the original expression.
2. Termination: whether the process of applying rewriting rules terminates.
3. Confluence: whether expressions are rewritten to the same result no matter the order of rewriting rules applied to it.

In HashedExpression, semantics-preserving and termination are tested and enforced programmatically. For semantics-preserving, it makes heavy use of QuickCheck [14], a property-based testing library, to verify against simplification rules as well as combinations of them. For each rule or combination of rules, a large number of random expressions and variables-substituted values are generated. Then, using a built-in interpreter, it will verify if the rewritten expression and the original expression evaluate to the same value.

Termination is enforced by applying the set of rewriting rules exhaustively and repeatedly with an additional "fuel" parameter. The process will terminate when none of the rewriting rules can apply, or after the maximum "fuel" times rules are applied, whichever comes first. In the future, when we allow users to provide custom rewriting rules along with their optimization instances, we will need to add a detection system that provides feedback if the rewriting causes cycles or, e.g., if the process diverges with the expression's size increasing after every iteration.

The confluence property is the hardest property to enforce or prove, and this is outside the scope of this thesis and will be left for future work. One direction would be to use the semi-decidable Knuth-Bendix algorithm [37]. However, not all rewriting rules are written using the pattern language (for example, see Listing 6), and with conditional term-rewriting allowed (using the guarded style), more consideration must be taken. Nevertheless, the confluence property is not a must-have in the context of a symbolic AML with code generation. The reason is that expressions' evaluation time varies between code generators, computer architectures, and runtime factors, especially when parallelization is involved. Thus, a single, unique normal form implied by the confluence property does not mean it is the most efficient form in terms of evaluation time.

Chapter 6

Computing Derivatives

6.1 Background

Recall the standard form of optimization problems:

$$\begin{array}{ll} \text{minimize:} & f(x) \\ \text{subject to:} & g_i(x) \leq 0, \quad i = 1, 2, ..., m_g \\ & h_i(x) = 0, \quad i = 1, 2, ..., m_h \\ & (f, g_i, h_i : \mathbb{R}^n \longrightarrow \mathbb{R}) \end{array}$$

Automatic computation of derivatives is another essential part of algebraic modeling software. Optimization algorithms and solvers rely on the computation of the objective function's gradient ∇f and constraints' gradients ∇g_i (and sometimes the Hessians of the objective function). Lacking the ability to automatically derive gradients, users are left with the error-prone and time-consuming task of manually calculating and inputting derivatives. Thus, it is usually mandatory for algebraic modeling software to include support for the automatic computation of derivatives.

Methods for computing derivatives are usually classified into 3 categories: (1) numerical differentiation, (2) symbolic differentiation, and (3) automatic differentiation. Numerical differentiation uses finite difference approximations and is the easiest method to implement. However, this method has the least accuracy due to floating-point round-off and truncation errors [57]. Alternatively, symbolic differentiation, available in computer algebra systems such as Mathematica [67] or Maple [12], operates on the symbolic expressions and produces symbolic derivative expressions using the chain rule. On the other hand, automatic differentiation also uses the chain rule but produces the numerical derivatives during evaluation time by accumulation of values instead of dealing with symbolic expressions [4], and often comes in 2 modes: forward and reverse. Automatic differentiation is probably the most common method and is available in most AMLs and machine learning libraries like JuMP or PyTorch.

There is a common belief that automatic differentiation is by nature more efficient than symbolic differentiation because the latter has the problem of *expression swelling* [29, 4, 16]. However, the claim is only true if expression

representation does not allow for common expressions sharing. This is addressed in the paper [40] and it is indicated that the two types of differentiation use the same set of techniques and only differ in output representation, i.e., symbolic differentiation targets derivative expressions while automatic differentiation targets their numeric computed values.

Many Haskell libraries support automatic computation of derivatives, with the ad package [2] being the most established and popular. The ad package implements different automatic differentiation modes and can also produce symbolic derivatives when used with the symbolic numeric type. For our purposes, we want support for multi-dimensional variables, the ability to extend the list of operators and extract the resulting expression graphs. Unfortunately, to the best of our knowledge, none of the current solutions offer that functionality.

HashedExpression implements its own derivative computation module. It takes the hybrid approach by using the reverse accumulation method (the same technique used in reverse-mode automatic differentiation) to work out the symbolic derivative expressions. Then, the result can be combined and simplified together with the original objective function (see Figure 6.1). In this chapter, we will go over the reverse-mode technique, how it is implemented in our current expression data structure. We will show why producing symbolic derivative expressions can be advantageous compared to producing numerical derivative values.

6.2 Method

6.2.1 Reverse mode

The reverse accumulation method (or reverse mode differentiation) to compute first-order derivatives was first published by Seppo Linnainmaa in 1976 [28]. The algorithm gives an outline of how to compute derivatives of functions $f : \mathbb{R}^n \to \mathbb{R}^m$ in m reversal sweeps, and requires a data structure to store information about all intermediate variables, otherwise known as a Wengert list [66]. In our case, we only target scalar functions ($m = 1$) for the both objective function and general constraints, so only a single pass is necessary. For the latter, the requisite data structure is encoded in our expression graph representation.

The algorithm works by computing the partial derivative of the target scalar function f with respect to all intermediate variables $\overline{v_i} = \frac{\partial f}{\partial v_i}$ (corresponding to nodes in our expression graphs) in the reverse order of the computational graph. For each intermediate node, e.g., $v(x, y)$, the derivative of $\overline{v} = \frac{\partial f}{\partial v}$ is fixed. Then, the (cumulative) derivatives of its operands are calculated using the chain rule:

$$\overline{x} = \frac{\partial f}{\partial x} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial x} = \overline{v}\frac{\partial v}{\partial x} \tag{6.1}$$

$$\overline{y} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial y} = \overline{v}\frac{\partial v}{\partial y} \tag{6.2}$$

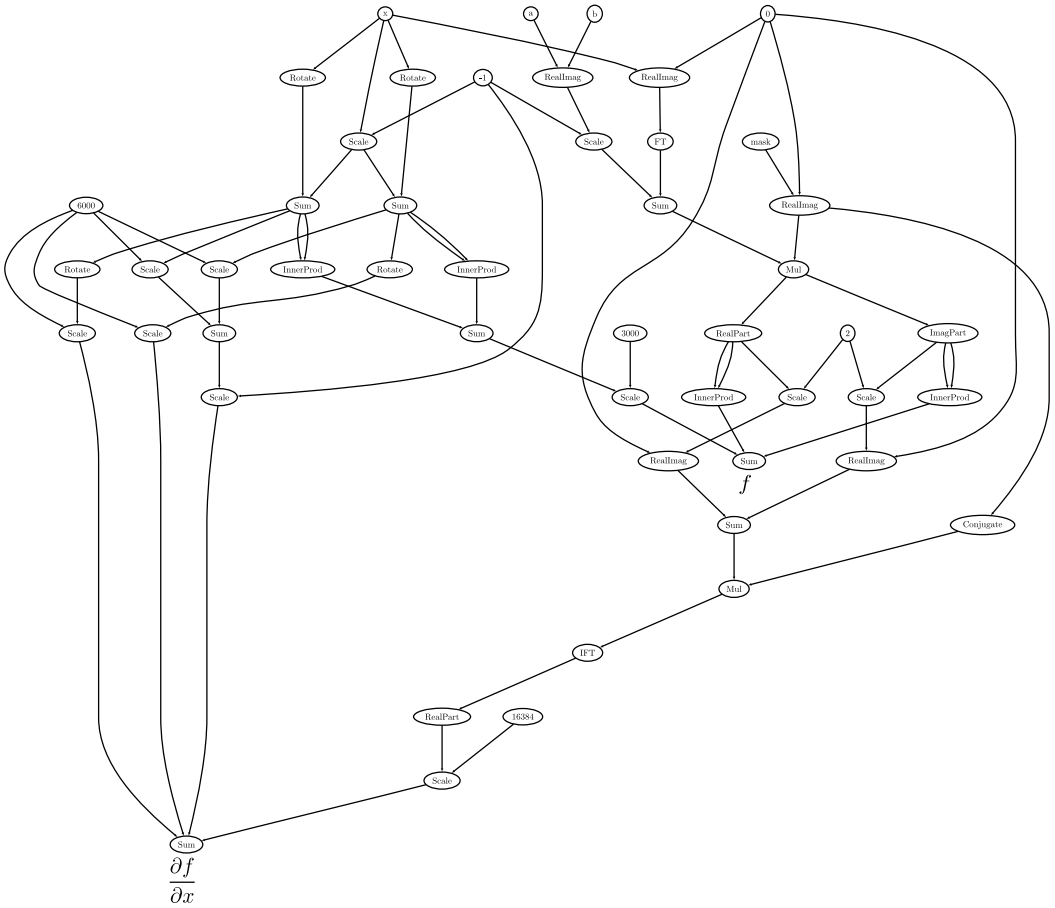


Figure 6.1: Combining the objective function and its derivatives (MRI reconstruction problem in Section 1.4).

The process starts with $\overline{f} = 1$, and upon visiting an intermediate node v, its derivative $\overline{v}$ is finalized by taking the sum of all cumulative derivatives contributed from (previously traversed) nodes dependent on it.

Take the scalar function $f = x(2x+1)+y^2$ for example, its expression graph and corresponding Wengert list is illustrated in Figure 6.2. From the reverse order of the computation graph, we compute derivatives of all expression nodes by the mechanics described in Figure 6.3.

However, most practical optimization problems use higher dimensional variables, and in HashedExpression, we also want to support complex variables. Fortunately, the reverse method works just as well with higher dimensional variables with some modifications. The partial derivative is generalized to the vectorized form:

$$\overline{\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

And the chain rule in equation 6.1 instead takes the Vector-Jacobian product form:

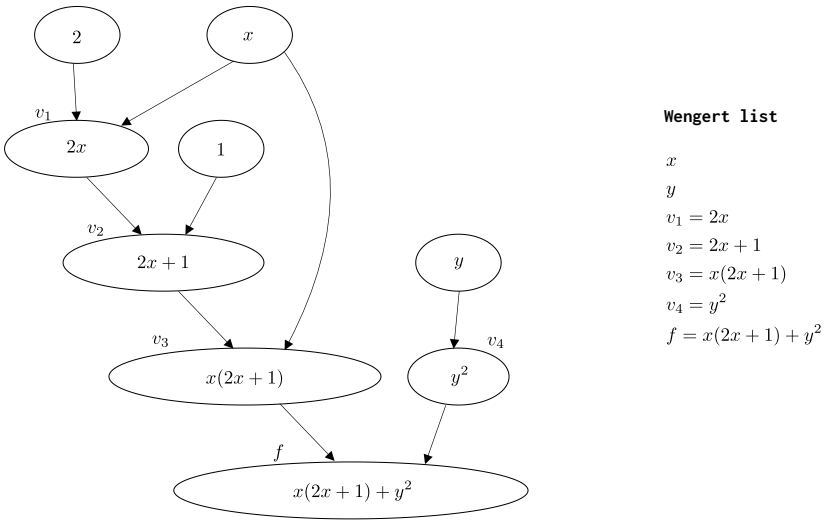


Figure 6.2: Expression graph and Wengert list of $f = x(2x + 1) + y^2$

$$\overline{\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} = \mathbf{J}^T \overline{\mathbf{v}}$$

where

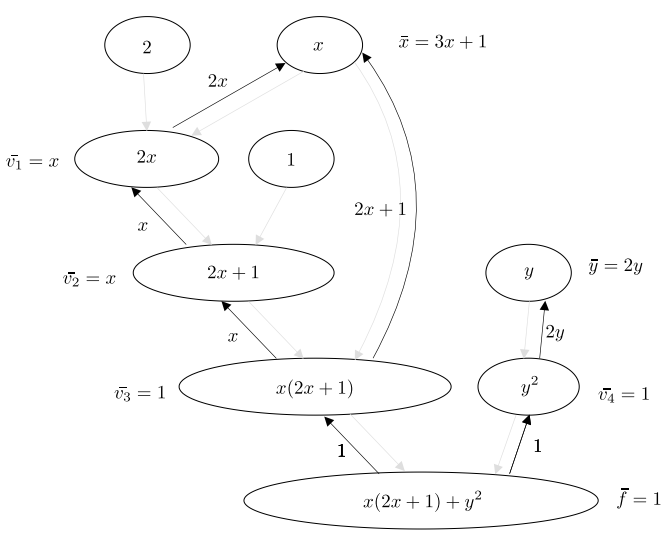
$$\mathbf{J} = \frac{\partial \mathbf{v}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \cdots & \frac{\partial v_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial v_n}{\partial x_1} & \cdots & \frac{\partial v_n}{\partial x_n} \end{bmatrix}$$

The Jacobian $\mathbf{J}$ has size $m * n$ and might be impractical to calculate, especially in image processing problems. Fortunately, most operators do not require explicit computation of the Jacobian, but can instead directly work out a vectorized representation of the Jacobian product $\mathbf{J}^T \overline{\mathbf{v}}$. For example, consider the point-wise sin operator:

$$\mathbf{v} = \sin(\mathbf{x}) = \begin{bmatrix} \sin(x_1) \\ \sin(x_2) \\ \vdots \\ \sin(x_n) \end{bmatrix}$$

the Jacobian is:

$$\mathbf{J} = \begin{bmatrix} \cos(x_1) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \cos(x_n) \end{bmatrix}$$



$\rightarrow \overline{f} = 1$

$\frac{\partial f}{\partial v_4} = 1 \Rightarrow add(\overline{v_4}, 1 * 1)$

$\frac{\partial f}{\partial v_3} = 1 \Rightarrow add(\overline{v_4}, 1 * 1)$

$\rightarrow \overline{v_4} = 1 * 1 = 1$

$\frac{\partial v_4}{\partial y} = 2y \Rightarrow add(\overline{y}, 1 * 2y)$

$\rightarrow \overline{v_3} = 1 * 1 = 1$

$\frac{\partial v_3}{\partial v_2} = x \Rightarrow add(\overline{v_2}, 1 * x)$

$\frac{\partial v_3}{\partial x} = 2x + 1 \Rightarrow add(\overline{x}, 1 * (2x + 1))$

$\rightarrow \overline{v_2} = 1 * x = x$

$\frac{\partial v_2}{\partial v_1} = 1 \Rightarrow add(\overline{v_1}, x * 1)$

$\rightarrow \overline{v_1} = x * 1 = x$

$\frac{\partial v_1}{\partial x} = 2 \Rightarrow add(\overline{x}, x * 2)$

$\rightarrow \overline{y} = 1 * 2y = 2y$

$\rightarrow \overline{x} = 1 * (2x + 1) + x * 2 = 3x + 1$

Figure 6.3: Applying reverse accumulation to compute derivatives

without explicitly computing $\mathbf{J}$, we can obtain the derivative:

$$\overline{\mathbf{x}} = \mathbf{J}^T \overline{\mathbf{v}} = \cos(\mathbf{x}) * \overline{\mathbf{v}}$$

where $(*)$ is the point-wise multiplication operator.

Complex variables are actually a special case of higher dimensional variables that we can treat as 2-parts of an extra dimension. The derivative of a complex expression $\mathbf{z} = \mathbf{a} + \mathbf{b}i$ is defined as:

$$\overline{\mathbf{z}} = \frac{\partial f}{\partial \mathbf{z}} = \frac{\partial f}{\partial \mathbf{a}} + i\frac{\partial f}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial f}{\partial a_1} + i\frac{\partial f}{\partial b_1} \\ \frac{\partial f}{\partial a_2} + i\frac{\partial f}{\partial b_2} \\ \vdots \\ \frac{\partial f}{\partial a_n} + i\frac{\partial f}{\partial b_n} \end{bmatrix}$$

The chain rule applies similarly, although additional conjugations and other complex number operations may be involved.

6.2.2 Implementation

The process involves creating derivative expressions as we traverse over the code graph. To have a convienient interface to do this we create another instance of `MonadExpression`:

```haskell
data ComputeDState = ComputeDState
  { contextMap :: ExpressionMap,
    cumulativeDerivatives :: Map NodeID [NodeID],
    partialDerivativeMap :: Map String NodeID
  }

newtype ComputeReverseM a = ComputeReverseM (State ComputeDState a)
  ↪ deriving Monad

instance MonadExpression (State ComputeDState) where
  -- similar to the Rewrite monad
  introduceNode node = ...
  getContextMap = ComputeReverseM $ gets contextMap
```

Listing 7: The `ComputeReverseM` monad for computing derivative using the reverse mode

The `ComputeReverseM` monad shares expressions composing logic with all the instances of `MonadExpression` we have seen throughout this thesis. We also need to manage the cumulative derivatives contributed from dependent nodes to their operand nodes through the `cumulativeDerivatives` mapping.

The rest is actually quite straightforward. We follow the reverse order topological sort of the original expression graph, and for each node:

1. Finalize the derivative expression of current node, which is the sum of all cumulative derivatives incurred previously from nodes dependent on the current node.

2. Pattern match on the operator, and use the chain rule and contribute derivative expressions to its operands.

This is illustrated in Listing 8. We can see a close resemblance to the written process shown in the right side of Figure 6.3.

In a sense, the algorithm can be thought of as traditional reverse-mode automatic differentiation but executed in the `ComputeReverseM` monad. Instead of producing numerical values, we produce symbolic derivative expressions which are added to the common lookup table. Figure 6.4 shows an (unsimplified) result of computing derivatives for $f = x(2x + 1) + y^2$.

It is worth noting that graphs, tapes, and mutations are not required to implement reverse-mode differentiation as Conal Elliott elegantly shows in the paper "The Simple Essence of Automatic Differentiation" [22]. The author defines computational categories shared between operators and their derivative formulas. As such, functions written in the defined categorical vocabulary take two forms: one to evaluate itself, and another to evaluate both its original value and derivatives. However, we have yet to find compatible uses of the paper's method in our modeling library, since we need explicit control of the expression graph for simplification and code generation.

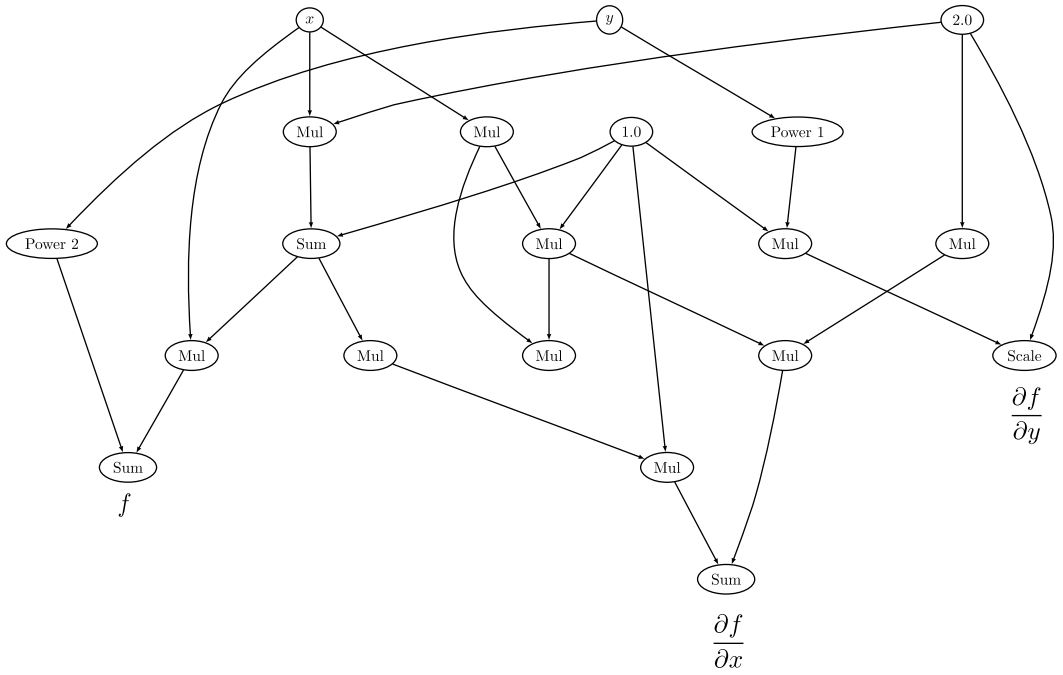


Figure 6.4: Computing derivatives result for $f = x(2x + 1) + y^2$

6.2.3 Sharing Computations and Simplification

As identical expressions are indexed to the same node, we avoid the issue of *expression swelling* that is often considered to be a drawback of symbolic differentiation techniques. Having symbolic derivative expressions also means:

1. We can apply rewriting rules to simplify and remove redundant computations of computed derivative expressions.

2. By combining multiple functions and their derivative expressions, we can share computations of not only the objective function f and its derivatives ∇f, but also the problem's constraints g_i and their derivatives ∇g_i.

Consider the example $f = x(2x+1)+y^2$. Generally, reverse-mode automatic differentiation will run a computation equivalent to the graph in Figure 6.4 (unless algebraic checks are performed, but doing so would incur extra running time). In contrast, if we obtain the symbolic derivative expressions we can apply rewriting rules and simplify them to the graph shown in Figure 6.5. The original has to evaluate 17 operators, whereas the simplified one only requires 8.

As we are dealing with multi-dimensional expressions, the significance of computations sharing is shown not only in the number of operators common between the objective function and its derivatives, but also the number of primitive machine operations shared between them. If we consider the number of primitive machine multiplications, in the MRI reconstruction example:

- The simplified graph for evaluating the objective function is shown in Figure 6.6. It has 21 operators and requires approximately 573441 machine multiplications (how to approximate: for instance, with expressions of shape 128×128, each point-wise multiplication requires $16384 *$

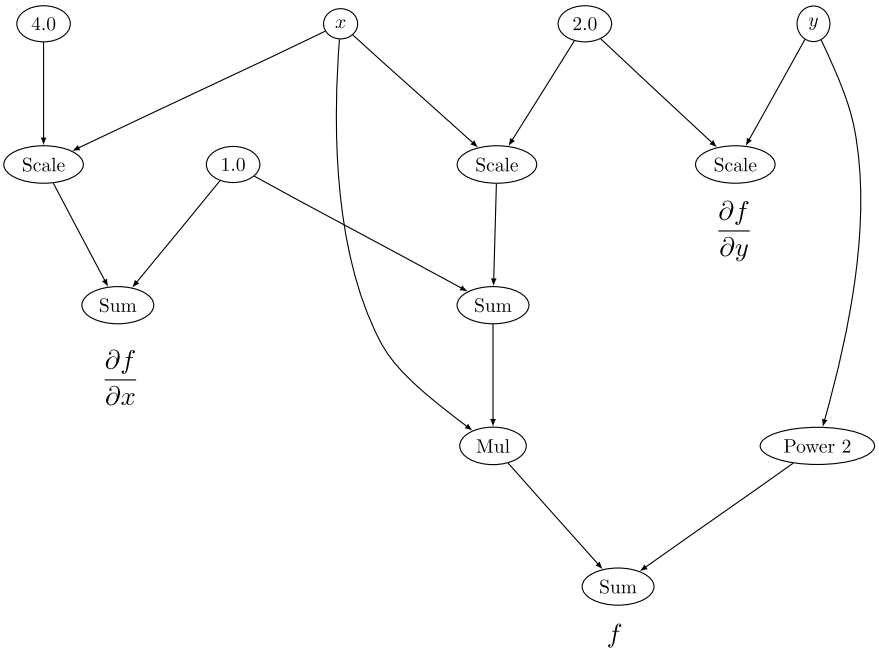


Figure 6.5: Simplified expression graph of $f = x(2x + 1) + y^2$ and its derivatives

(number of operands - 1) machine multiplications, each dot product requires 16384, and the Fourier transform requires $16384 * \log(16384) = 229376$).

- The simplified graph for evaluating the gradient is shown in Figure 6.7. It has 33 operators and requires approximately 1114112 machine multiplications.
- The combined expression graph of both values is illustrated in Figure 6.1. It has 41 operators and requires approximately 1179649 machine multiplications.

Thus, compared with if we were to evaluate the objective function and its gradient separately, evaluating the combined graph reduces the number of operators by 24% and reduces the number of machine multiplications by 30%.

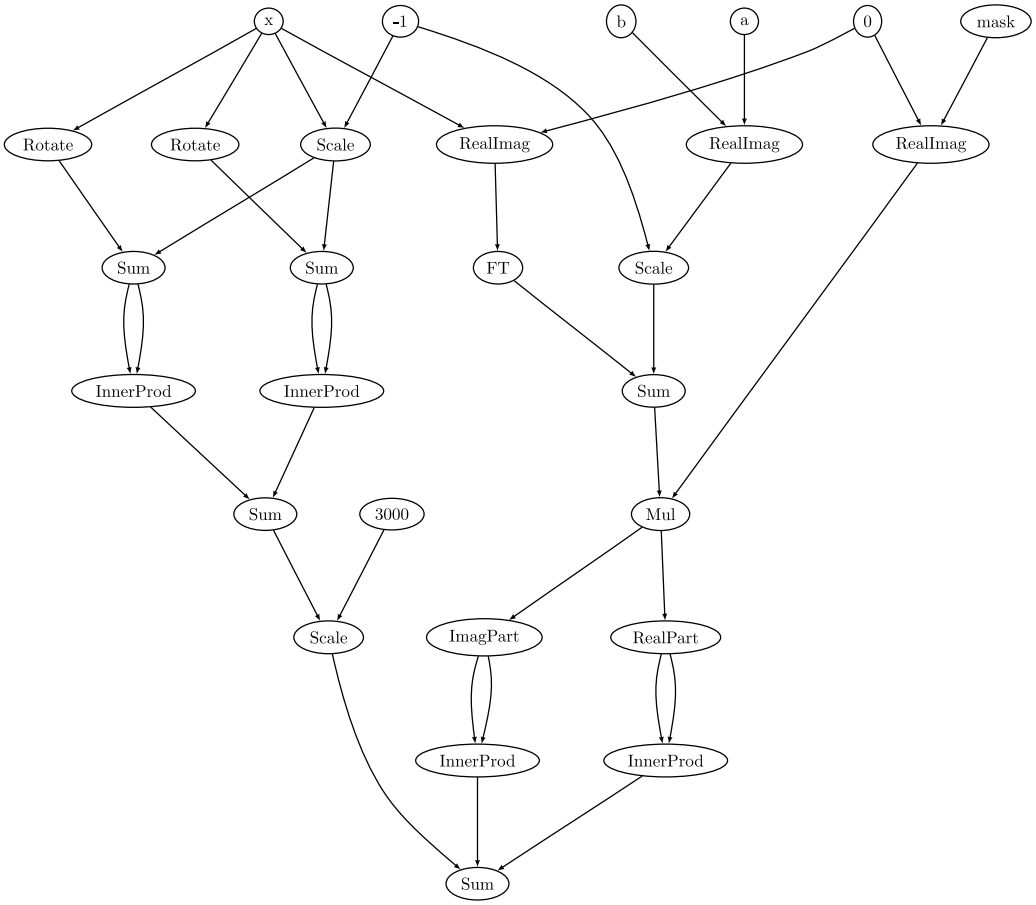


Figure 6.6: Evaluating the objective function (MRI reconstruction problem)

```haskell
process :: NodeID -> ComputeReverseM ()
process n = do
  --- finalize the derivative expression of the current node
  diffN <-
    if nID == rootID
      then one
      else do
        derivativeParts <- Map.lookup nID <$> gets
        ↪  cumulativeDerivatives
        -- Sum all the derivative parts incurred by its parents
        case derivativeParts of
          Just [d] -> from d
          Just ds -> sum_ $ map from ds
          _ -> zero
  ....
  case op of
    Var name -> setPartialDerivative n diffN
    Sum args -> do
      forM_ args $ \x -> do
        let diffX = diffN
        addDerivative x diffX
    Mul args -> do
      forM_ (removeEach args) $ \(x, rest) -> do
        productRest <- product_ $ map from rest
        case et of
          R -> do
            diffX <- from diffN * from productRest
            addDerivative x diffX
          C -> do
            diffX <- from diffN * conjugate (from productRest)
            addDerivative x diffX
    Sin x -> do
      diffX <- from diffN * cos (from x)
      addDerivative x diffX
    Cos x -> do
      diffX <- from diffN * (- sin (from x))
      addDerivative x diffX
    ...
```

Listing 8: Processing a node in reverse mode

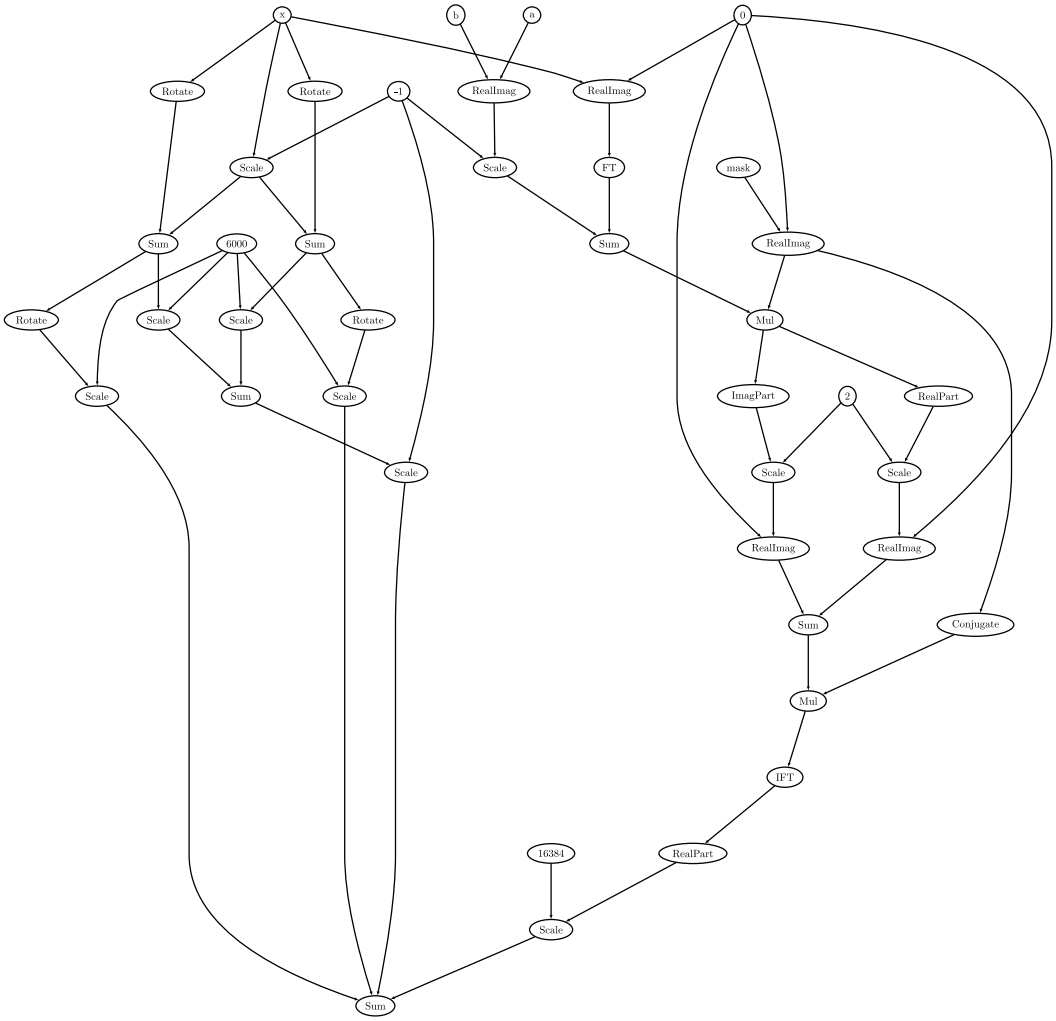


Figure 6.7: Evaluating the gradient (MRI reconstruction problem)

Chapter 7

Code Generation

7.1 Interfacing With Optimization Solvers

The last component of our algebraic modeling language is code generation. The module is responsible for transforming symbolic expressions (encoded as directed acyclic graphs) into source code which can be compiled with third-party optimization solvers. Similar to CVXGEN [44] or Modelica [3], HashedExpression combines a modeling language with code generation because it allows for (1) high-performance gain with low-level C/C++ code and (2) the possibility to combine with other optimization systems (because code generation is backend-agnostic, we can generate evaluation code in any language and/or numerical libraries).

With code generation, we can finally construct a process to solve optimization problems. First, the problem (in the standard form 1.1) is passed through pre-processing and symbolic manipulation steps that:

1. Perform necessary validations.
2. Compute the symbolic derivative expressions ∇f and ∇g_i as described in Chapter 6.
3. Perform simplification and rewriting as described in Chapter 5, and merge everything into a single expression graph to account for node sharing.

The result is an expression graph containing the objective function, constraints and their respective gradients. Then, the code generator turns it into evaluation code which can be integrated with numerical optimization algorithms to produce the final result, as illustrated in Figure 7.1. Code generation can also perform transformations to further speed up evaluation. Such could include rewriting combinations of operations to custom nodes that are evaluated by special hardware instructions, or collapsing a sequence of nodes into one single computation to save memory.

Although code generation is backend-agnostic, for granular control on performance and portability with optimization solver interfaces, we will mainly focus on generating low-level C/C++ code. In the next sections, we will present an implementation of a simple C code generator and discuss opportunities that can be employed to speed up evaluation in future generators.

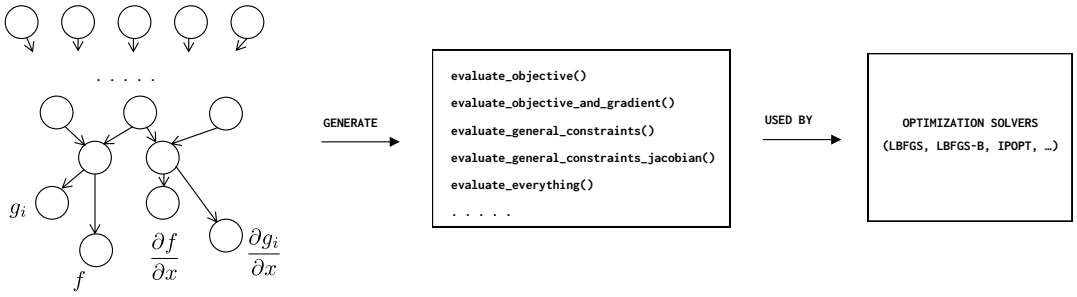


Figure 7.1: Code generation

7.2 Generating C Code

In this first release of HashedExpression, we include a fully working code generator that produces C code in the C99 standard. In this version, computations are sequential, and operations on higher dimensional variables are computed in elementary for-loops. For extra performance gain and convenient data interfacing, the Fourier transform is carried out using the FFTW3 library [26], and dataset input-output is facilitated by the HDF5 [24] format and library. Plain text format is also supported.

7.2.1 Implementation

HashedExpression provides a type class interface for defining different code generators. The first argument (`codeConfig`) is the code generator's configuration, the second argument (`Problem`) is the pre-processed optimization problem (see the steps mentioned at 3) , and the third (`ValMap`) contains parameter values and initial points for variables:

```
class Codegen codegenConfig where
  generateProblemCode ::
    codegenConfig ->
    -- the preprocessed optimization problem
    Problem ->
    -- values of parameters and variables
    ValMap ->
    -- if success, generate necessary files to the given filepath
    Either String (FilePath -> IO ())

instance Codegen CSimpleConfig where
  ....
```

The `CSimple` code generator implements this interface via its configuration type `CSimpleConfig`. The code generation algorithm consists of two parts. First, `initCodegen` bootstraps an optimization problem's metadata and memory allocation as a simple scheme that concatenates storage for every subexpression evaluated into two regions: one for real expressions and one for complex expressions (see Figure 7.2).

```haskell
data Address
  = AddressReal Offset    -- allocated in `double ptr*`
  | AddressComplex Offset -- allocated in `complex double ptr_c*`

data CSimpleCodegen = CSimpleCodegen
  { cExpressionMap :: ExpressionMap,
    cAddress :: NodeID -> Address,
    totalReal :: Int,                -- total number of real
    ↪ numbers
    totalComplex :: Int,             -- total number complex
    ↪ numbers
    (!!) :: NodeID -> Index -> Text, -- utility for accessing code
    config :: CSimpleConfig
  }

initCodegen :: CSimpleConfig -> ExpressionMap -> [NodeID] ->
↪ CSimpleCodegen
initCodegen config mp variableIDs = ...
```

Then, we generate the code to evaluate each node (each represents a subexpression) in the graph by topological order so that dependent nodes are resolved first, as shown in Listing 9. This also utilizes computation sharing because all nodes are evaluated once and their values persist throughout the entire process.

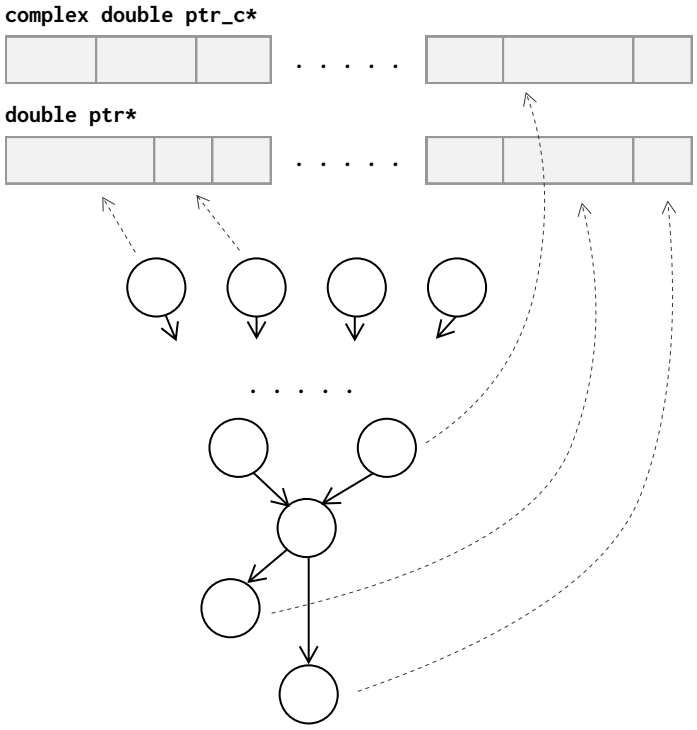


Figure 7.2: Simple memory allocation in the C99 code generator

```haskell
evaluating :: CSimpleCodegen -> [NodeID] -> Text
evaluating CSimpleCodegen {..} rootIDs =
  codeCToText $ Scoped $ map genCode $ topologicalSortManyRoots
  ↪  (cExpressionMap, rootIDs)
  where
    (!!)   :: NodeID -> Index -> Text
    sizeOf :: NodeID -> Int
    for    :: Text -> Int -> [CCode] -> CCode
    ....
    genCode :: NodeID -> CCode
    genCode n =
      let (_, et, op) = retrieveNode n cExpressionMap
       in case op of
            ...
            Sum args ->
              let sumAt i = T.intercalate " + " $ map (!! i) args
               in for i (sizeOf n) [(n !! i) := sumAt i]
            Mul args ->
              let prodAt i = T.intercalate " * " $ map (!! i) args
               in for i (sizeOf n) [(n !! i) := prodAt i]
            Neg arg ->
              for i (sizeOf n) [(n !! i) := ("-" <> (arg !! i))]
            ....
```

Listing 9: Generating C code

7.2.2 Using the generated code

The code generator produces a single, primary C file `problem.c` that contains all the data definitions and functions needed for computing the optimization problem. Optimization solvers can import this file and call these functions as needed. Data definitions include information such as the working space arrays (`ptr` for real expressions and `ptr_c` for complex expressions), variable metadata (e.g., their name, indices, and partial derivatives indices in the working space), and constraint-related information:

```
...
const char* var_name[NUM_HIGH_DIMENSIONAL_VARIABLES] = { "theta1",
↪  "theta0" };
const int var_size[NUM_HIGH_DIMENSIONAL_VARIABLES] = { 1, 1 };

const int var_offset[NUM_HIGH_DIMENSIONAL_VARIABLES] = { 0, 1 };
const int partial_derivative_offset[NUM_HIGH_DIMENSIONAL_VARIABLES]
↪  = { 1075, 1074 };
const int objective_offset = 1076;

double ptr[MEMORY_NUM_DOUBLES];
complex double ptr_c[MEMORY_NUM_COMPLEX_DOUBLES];
....
```

The C file exports functions that evaluate the objective and constraint functions (as well as their gradients), i.e., f, ∇f, g_i, and ∇g_i. Solvers can evaluate each of them in separation, or evaluate them together to take advantage of sharing.

```
void evaluate_partial_derivatives_and_objective() { ... };
void evaluate_objective() { ... };
void evaluate_partial_derivatives() { ... };
void evaluate_scalar_constraints() { ... };
void evaluate_scalar_constraints_jacobian() { ... };
...
```

All of the functions operate on a shared memory space with inputs and outputs implicitly defined. If the optimization solver wants to evaluate the objective function, it will need to write the values of variables into their allocated spot (determined by `ptr` and `var_offset`), run the `evaluate_objective` function, and read the evaluated objective value in `ptr[objective_offset]`.

Once the C file is generated, we can link it with C/C++ optimization solvers to produce the final result. These solvers are often interfaced via callback functions, and our task is to fill in these callback functions with appropriate calls to the functions defined in `problem.c`. For instance, the following code snippet demonstrates how one could interface with the libLBFGS solver:

```
#include <lbfgs.h>
#include "problem.c"
```

```
...

static lbfgsfloatval_t evaluate(void *instance,
    const lbfgsfloatval_t *x,
    lbfgsfloatval_t *g,
    const int n,
    const lbfgsfloatval_t _step) {
  evaluate_partial_derivatives_and_objective();
  int i;
  int cnt = 0;
  for (i = 0; i < NUM_HIGH_DIMENSIONAL_VARIABLES; i++) {
    memcpy(g + cnt, ptr + partial_derivative_offset[i], var_size[i]
    ↪  * sizeof(double));
    cnt += var_size[i];
  }

  return ptr[objective_offset];
}

int main() {
  ..
  lbfgsfloatval_t *x = ptr + VARS_START_OFFSET;.
  ret = lbfgs(N, x, &fx, evaluate, progress, NULL, &param);
  ...
}
```

In the initial release, we provide template files with autotools configuration (i.e., Makefiles) for automated building with the following solvers [1]:

- Our custom implementation of the gradient descent algorithm [58].
- libLBFGS, a C implementation of L-BFGS [51, 42]
- LBFGS-b, a C implementation L-BFGS-B [69, 48]
- Ipopt, an C/C++ solvers using the interior-point method [63].

This allows end-users to "plug-and-play" with their solver of choice, by simply dropping the generated `problem.c` into one of our pre-made templates and running `make`. And since these templates are simple, well commented C/C++ code with free and open source licenses users can easily customize or integrate them to suit their applications needs.

7.3 Speed up evaluation

The current C generator implementation should be considered an "-O0" level of optimization (optimize for compile-time over execution-time), and many aspects

[1] Source code is available at `https://github.com/McMasterU/HashedExpression/tree/master/solvers`

of it can be improved to speed up evaluation and make solving optimization problems more efficient overall.

High dimensional variables present an opportunity to implement "embarrassing parallelism". For certain operators, especially point-wise operators, computations among high dimensional values have no interdependency. Thus, we can handle these computations in parallel (see Figure 7.3). For this kind of data parallelism, we can accelerate computation with either single instruction, multiple data (SIMD) instructions or GPU programming.

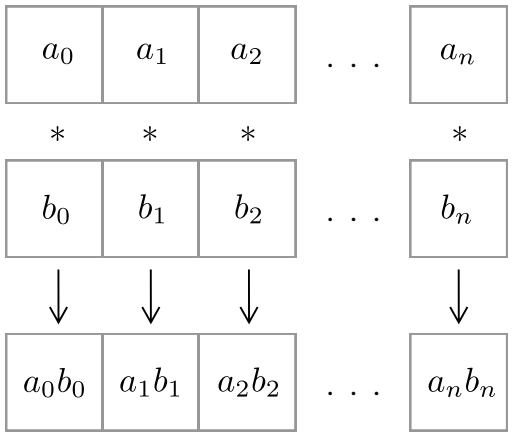


Figure 7.3: Point-wise operators executed in parallel

In addition to data parallelism with SIMD and GPU, we can introduce multithreading to evaluate different parts of computations concurrently. This relies on the fact that the produced acyclic graph (DAG) structure has a dependency order that can be used for concurrency control. From the expression graph, we can perform a graph analysis and split the work into multiple threads (see Figure 7.4) which can be run in different processing units.

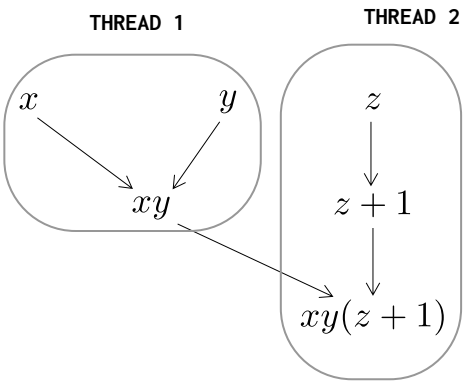


Figure 7.4: Evaluate expression graphs concurrently across multiple threads.

Another performance factor is memory allocation. Currently, all subexpressions are allocated in a shared memory pool that is far from optimal size since

the lifetimes of many intermediate values in a computation do not overlap. Although not implemented, we have a proposed solution to overcome this. First, we schedule the order of evaluation to determine lifetimes of subexpressions (each of which is an interval that begins when the subexpression is evaluated and ends when the subexpression is last used to compute another subexpression). Then, we can run the disjoint interval partitioning algorithm [11] to find the minimum number of partitions of mutually-disjointed intervals, and allocate the same memory space (with the size of the biggest subexpression) for all subexpressions in a partition. Furthermore, we can skip memory for intermediate expressions that are only used once, and their evaluation can instead be performed in-registers.

Chapter 8

Conclusion and Discussion

In this thesis, we have developed a full-scale algebraic modeling software for solving optimization problems. The library is built on the purely functional programming language Haskell that is endowed with facilities for powerful abstraction, flexible overloading, and type-level programming. Such features have allowed us to deliver advances in modeling and solving optimization problems, accomplishing the goals we had for HashedExpression as stated 1.3 in the introduction:

G1. We achieve type-safety by embedding the modeling language in Haskell. Moreover, we also developed type systems for expressions that capture metadata on the type-level preventing the construction of invalid expressions. We included 3 versions of expression type systems for modeling: one version without type-level metadata, one with shape & element type information, and the third adding physical units. Users can choose to model optimization problems using any of them, depending on the source of the problem and importance of correctness.

G2. Haskell operator overloading is used, and with support for custom infix operators, the syntax closely resembles conventional notation used by mathematicians and physicists.

G3. We have built a system that supports multi-dimensional variables and complex numbers, expressions are always accompanied with their shape and element type information.

G4. The reverse accumulation method is implemented to produce symbolic derivative expressions, and thanks to automatic common subexpression elimination, we avoid the problem of "expression swelling".

G5. We have developed a module for rewriting and simplifying symbolic expressions and extended it with a system of rewriting using matching and replacing.

G6. We included a C code generator and showed how to integrate it with different optimization solvers. The current version, although lacking parallelization and an efficient memory allocation scheme, can be used to test and cross-validate future code generators.

In this thesis, we have dissected the implementation of each component included in HashedExpression, emphasizing approaches available in a pure language like Haskell. We see that all the modules that involve the construction of expressions (modeling APIs, expression rewriting, and computing derivatives) rely on a monadic type class `MonadExpression` and use the `State` monad to control mutation without violating the purity of the language. We have unfolded a solid framework that brings together type-safety for modeling, symbolic manipulation for simplification and automatic computation of derivatives and low-level code generation for performance.

HashedExpression joins the family of the Coconut project written in Haskell and has been being used in other scheduling projects as well as Scanning Electron Microscope (SEM) imaging problems. The project is open-source and can be found at the Github repository `https://github.com/McMasterU/HashedExpression`.

8.1 Future Work

The library is in active development. There is much work to be done and many interesting directions and features to be explored. First, we need to implement parallelized and memory-efficient versions of code generation and perform benchmarking between code generators. These implementations will make use of existing, established BLAS [41] numerical libraries like CBLAS. We also consider generating code for high-level libraries and programming languages that have built-in parallelism and GPU support, e.g CuPy, a NumPy-like library that uses GPU calculations, or Futhark, a purely functional data-parallel array language [32], and benchmark them with low-level C code in the scope of solving optimization problems. Moreover, by targeting code generation to numerical libraries that support sparse linear algebra, we can allow users to work with sparse multi-dimensional expressions.

For symbolic manipulation, currently, we have a long list of predefined simplification rules. In the future, we want to allow users to add new simplification rules specific to their problem domain to further optimize computations, similar to GHC's `RULES` pragma [33]. However, as opposed to GHC that makes no attempt to verify that programmer-specified rules are valid, we can use QuickCheck property-based testing to generate random expressions, and verify them against the custom simplification rules.

Another consideration is the mechanism for operational extensibility. Currently, all mathematical functions are built-in and library developers are the ones to implement and extend the list of supported operators. We want to give this capability to users as well. In another words, we will need to re-parameterize the type system so that users can define custom operators and provide the rule to compute their derivatives. However, as our system is symbolic, it also means they will need to provide code generation rules to evaluate them. Therefore, this feature should be used only by advanced users, and, in the meantime, users can introduce new operators by contributing to the open-source project.

Appendix A

Type family implementation of projection

```
data Selector
  = All
  | Range Nat Nat Nat
  | At Nat

type family IsSelectors (selectors :: [Selector]) where
  IsSelectors _ = Satisfied


type family Require (conditions :: [(Bool, ErrorMessage)]) (val ::
↪  k) :: k where
  Require '[] val = val
  Require ('(condition, errorMessage) : cs) val =
    IfThenElse condition (Require cs val) (TypeError errorMessage)

type family ProjectionShape (inputShape :: [Nat]) (selectors ::
↪  [Selector]) :: [Nat] where
  ProjectionShape '[] '[] = '[]
  ProjectionShape '[] _ = TypeError (Text "Shape and selector
  ↪  dimensions mismatch")
  ProjectionShape _ '[] = TypeError (Text "Shape and selector
  ↪  dimensions mismatch")
  ProjectionShape (n : ns) (All : ss) = n : ProjectionShape ns ss
  ProjectionShape (n : ns) (At pos : ss) =
    Require
      '[ '( CmpNat pos n == LT,
            Text "Invalid index " :<>: ShowType pos
              :<>: Text ", must be in range (0, "
              :<>: ShowType n
              :<>: Text ")"
          )
       ]
```

```
    (ProjectionShape ns ss)
ProjectionShape (n : ns) (Range start end step : ss) =
  Require
    '[ '( CmpNat start n == LT,
          Text "Invalid start index "
            :<>: ShowType start
            :<>: Text ", must be in range (0, "
            :<>: ShowType n
            :<>: Text ")"
       ),
       '( CmpNat end n == LT,
          Text "Invalid end index "
            :<>: ShowType end
            :<>: Text ", must be in range (0, "
            :<>: ShowType n
            :<>: Text ")"
       ),
       '(CmpNat 0 step == LT, Text "Step must be positive")
     ]
    ((((n + end - start) `Mod` n) `Div` step + 1) :
    ↪  ProjectionShape ns ss)
```

Bibliography

[1] Martín Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] *ad: Automatic Differentiation*. `https://hackage.haskell.org/package/ad`. (Accessed on 04/01/2021).

[3] Johan Åkesson et al. "Modeling and optimization with Optimica and JModelica. org—Languages and tools for solving large-scale dynamic optimization problems". In: *Computers & Chemical Engineering* 34.11 (2010), pp. 1737–1749.

[4] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *Journal of machine learning research* 18 (2018).

[5] Dennis E Blumenfeld, Debra A Elkins, and Jeffrey M Alden. "Mathematics and operations research in industry". In: *Focus* 24.2 (2004), pp. 10–12.

[6] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: `http://github.com/google/jax`.

[7] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation." In: *J. Funct. Program.* 23.5 (2013), pp. 552–593.

[8] *BrainWeb: Simulated Brain Database*. URL: `http://www.bic.mni.mcgill.ca/brainweb/`.

[9] Anthony Brook, David Kendrick, and Alexander Meeraus. "GAMS, a user's guide". In: *ACM Signum Newsletter* 23.3-4 (1988), pp. 10–11.

[10] Bruno Buchberger and Rüdiger Loos. "Algebraic simplification". In: *Computer algebra*. Springer, 1982, pp. 11–43.

[11] Francesco Cafagna and Michael H Böhlen. "Disjoint interval partitioning". In: *The VLDB Journal* 26.3 (2017), pp. 447–466.

[12] Bruce W Char et al. *Maple V library reference manual*. Springer Science & Business Media, 2013.

[13] James Cheney and Ralf Hinze. *First-class phantom types*. Tech. rep. Cornell University, 2003.

[14] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Acm sigplan notices* 46.4 (2011), pp. 53–64.

[15] Chris A Cocosco et al. “Brainweb: Online interface to a 3D MRI simulated brain database”. In: *NeuroImage*. Citeseer. 1997.

[16] George F Corliss. “Applications of differentiation arithmetic”. In: *Reliability in Computing*. Elsevier, 1988, pp. 127–148.

[17] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. “A comprehensive study of real-world numerical bug characteristics”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 509–519.

[18] Steven Diamond and Stephen Boyd. “CVXPY: A Python-embedded modeling language for convex optimization”. In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2909–2913.

[19] *DimaSamoz/mezzo: A Haskell library for typesafe music composition*. `https://github.com/DimaSamoz/mezzo`. (Accessed on 01/26/2021).

[20] Iain Dunning, Joey Huchette, and Miles Lubin. “JuMP: A modeling language for mathematical optimization”. In: *SIAM Review* 59.2 (2017), pp. 295–320.

[21] Richard A Eisenberg and Jan Stolarek. “Promoting functions to type families in Haskell”. In: *ACM SIGPLAN Notices* 49.12 (2014), pp. 95–106.

[22] Conal Elliott. “The simple essence of automatic differentiation”. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–29.

[23] Hans J Ferreau et al. “Embedded optimization methods for industrial automatic control”. In: *IFAC-PapersOnLine* 50.1 (2017), pp. 13194–13209.

[24] Mike Folk et al. “An overview of the HDF5 technology suite and its applications”. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. 2011, pp. 36–47.

[25] Robert Fourer, David M Gay, and Brian W Kernighan. “AMPL. A modeling language for mathematical programming”. In: (2003).

[26] Matteo Frigo and Steven G Johnson. “The design and implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[27] Michael Grant and Stephen Boyd. *CVX: Matlab software for disciplined convex programming, version 2.1*. 2014.

[28] Andreas Griewank. “Who invented the reverse mode of differentiation”. In: *Documenta Mathematica, Extra Volume ISMP* (2012), pp. 389–400.

[29] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[30] Andreas Griewank et al. “On automatic differentiation”. In: *Mathematical Programming: recent developments and applications* 6.6 (1989), pp. 83–107.

[31] William E Hart, Jean-Paul Watson, and David L Woodruff. “Pyomo: modeling and solving mathematical programs in Python”. In: *Mathematical Programming Computation* 3.3 (2011), p. 219.

[32] Troels Henriksen et al. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 556–571.

[33] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. “Playing by the rules: rewriting as a practical optimisation technique in GHC”. In: *Haskell workshop*. Vol. 1. 2001, pp. 203–233.

[34] Josef Kallrath. *Algebraic Modeling Systems: Modeling and Solving Real World Optimization Problems*. Vol. 104. Springer Science & Business Media, 2012.

[35] Richard M Karp and Michael O Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM journal of research and development* 31.2 (1987), pp. 249–260.

[36] Csongor Kiss et al. “Higher-order type-level programming in Haskell”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–26.

[37] Donald E Knuth and Peter B Bendix. “Simple word problems in universal algebras”. In: *Automation of Reasoning*. Springer, 1983, pp. 342–376.

[38] U.S. Bureau of Labor Statistics. *Operations Research Analysts*. 2020. URL: `https://www.bls.gov/ooh/math/operations-research-analysts.htm#tab-6` (visited on 09/01/2020).

[39] Rasmus Munk Larsen and Tatiana Shpeisman. “TensorFlow Graph Optimizations”. In: (2019).

[40] Sören Laue. “On the equivalence of forward mode automatic differentiation and symbolic differentiation”. In: *arXiv preprint arXiv:1904.02990* (2019).

[41] Chuck L Lawson et al. “Basic linear algebra subprograms for Fortran usage”. In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[42] Dong C Liu and Jorge Nocedal. “On the limited memory BFGS method for large scale optimization”. In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.

[43] Johan Lofberg. “YALMIP: A toolbox for modeling and optimization in MATLAB”. In: *2004 IEEE international conference on robotics and automation (IEEE Cat. No. 04CH37508)*. IEEE. 2004, pp. 284–289.

[44] Jacob Mattingley and Stephen Boyd. “CVXGEN: A code generator for embedded convex optimization”. In: *Optimization and Engineering* 13.1 (2012), pp. 1–27.

[45] Alp Mestanogullari et al. “Type-level web APIs with servant: an exercise in domain-specific generic programming”. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. 2015, pp. 1–12.

[46] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (2017), e103.

[47] Maryam Moghadas. "Type-safety for inverse imaging problems". PhD thesis. 2012.

[48] José Luis Morales and Jorge Nocedal. "Remark on "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization"". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–4.

[49] Takayuki Muranushi and Richard A Eisenberg. "Experience report: Type-checking polymorphic units for astrophysics research in Haskell". In: *ACM SIGPLAN Notices* 49.12 (2014), pp. 31–38.

[50] David R Musser and Alexander A Stepanov. "Generic programming". In: *International Symposium on Symbolic and Algebraic Computation*. Springer. 1988, pp. 13–25.

[51] Jorge Nocedal. "Updating quasi-Newton matrices with limited storage". In: *Mathematics of computation* 35.151 (1980), pp. 773–782.

[52] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, 2007.

[53] Chris Okasaki and Andrew Gill. "Fast Mergeable Integer Maps". In: *In Workshop on ML*. 1998, pp. 77–86.

[54] Adam Paszke et al. "Automatic differentiation in pytorch". In: (2017).

[55] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems*. 2019, pp. 8026–8037.

[56] Jessica L. M. Pavlin. "Symbolic Generation of Parallel Solvers for Unconstrained Optimization". MSc Thesis. McMaster University, Department of Computing and Software, 2012.

[57] Alexander Ramm and Alexandra Smirnova. "On stable numerical differentiation". In: *Mathematics of computation* 70.235 (2001), pp. 1131–1153.

[58] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[59] *stephenbeckr/L-BFGS-B-C: L-BFGS-B, converted from Fortran to C, with Matlab wrapper*. `https://github.com/stephenbeckr/L-BFGS-B-C`. (Accessed on 01/06/2021).

[60] *Types · The Julia Language*. `https://docs.julialang.org/en/v1/manual/types/`.

[61] Madeleine Udell et al. "Convex Optimization in Julia". In: *SC14 Workshop on High Performance Technical Computing in Dynamic Languages* (2014). arXiv: `1410.4821 [math-oc]`.

[62] Lieven Vandenberghe and Stephen Boyd. "Semidefinite programming". In: *SIAM review* 38.1 (1996), pp. 49–95.

[63] Andreas Wächter and Lorenz T Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical programming* 106.1 (2006), pp. 25–57.

[64] Philip Wadler. "Monads for functional programming". In: *International School on Advanced Functional Programming*. Springer. 1995, pp. 24–52.

[65] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in science & engineering* 13.2 (2011), pp. 22–30.

[66] Robert Edwin Wengert. "A simple automatic derivative evaluation program". In: *Communications of the ACM* 7.8 (1964), pp. 463–464.

[67] Stephen Wolfram et al. *The MATHEMATICA® book, version 4*. Cambridge university press, 1999.

[68] Brent A Yorgey et al. "Giving Haskell a promotion". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. 2012, pp. 53–66.

[69] Ciyou Zhu et al. "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization". In: *ACM Transactions on Mathematical Software (TOMS)* 23.4 (1997), pp. 550–560.