Differentiable Functional Program Interpreters

John K. Feser [1] **Marc Brockschmidt** [2] **Alexander L. Gaunt** [2] **Daniel Tarlow** [3]

Abstract

Programming by Example (PBE) is the task of inducing computer programs from input-output examples. It can be seen as a type of machine learning where the hypothesis space is the set of legal programs in some programming language. Recent work on *differentiable interpreters* relaxes the discrete space of programs into a continuous space so that search over programs can be performed using gradient-based optimization. While conceptually powerful, so far differentiable interpreter-based program synthesis has only been capable of solving very simple problems. In this work, we study modeling choices that arise when constructing a differentiable programming language and their impact on the success of synthesis. The main motivation for the modeling choices comes from functional programming: we study the effect of memory allocation schemes, immutable data, type systems, and built-in control-flow structures. Empirically we show that incorporating functional programming ideas into differentiable programming languages allows us to learn much more complex programs than is possible with existing differentiable languages.

1. Introduction

A key decision in supervised machine learning is to choose a hypothesis space, which is the space of possible mappings from inputs to outputs. Common choices for hypothesis spaces are neural networks, decision trees, and nearest neighbor models. The premise underlying this work is that programs,[1] or programs in combination with other models like neural networks, are a useful hypothesis class for machine learning. Programs are interesting models because (1) they can be very expressive—they define complex modern technical infrastructure like operating systems and the internet—and (2) they come with a strong inductive bias. Programming languages are designed to make common programming patterns compact and easy to express, and with appropriate priors this might bias a learning method to favor hypotheses that humans consider to be natural programs. The ultimate goal of this line of work is to build models that generalize strongly from a small amount of data but are expressive enough to fit large, complex data sets. Such models might be particularly applicable to sequential and procedural data.

However, recent results indicate that programs are a difficult hypothesis class to work with. There is significant literature on discrete search-based techniques for program induction (e.g., (Gulwani, 2011; Albarghouthi et al., 2013; Feser et al., 2015; Frankle et al., 2016)), and a small amount of recent work on gradient-based program induction (Bunel et al., 2016; Riedel et al., 2016; Gaunt et al., 2016). (Gaunt et al., 2016) has shown for low-level assembly-like differentiable programming languages, discrete search performs better than gradient-based search, and further that the kinds of problems that can be solved by differentiable interpreters are limited to simple problems like accessing an element of an array.

One may then wonder why it is worth continuing to study differentiable programming languages. We believe there are three main reasons. First, differentiable programming languages allow program-like models to be composed with neural network-like models, which opens up many possibilities. However, the bottleneck in scaling up is that differentiable interpreters cannot currently solve complex problems. Second, differentiable interpreters appear fundamentally different from discrete search, and they may have benefits. For example, differentiable interpreters naturally operate in the stochastic regime with small minibatches of data processed at each step, whereas discrete search methods typically operate in a batch regime.[2] They may also give more natural ways of handling noisy data. The related final point is that differentiable interpreters are a very

[1]Massachusetts Institute of Technology, Cambridge, US [2]Microsoft Research, Cambridge, UK [3]Google Brain, Montréal, Canada (work done while at Microsoft). Correspondence to: John K. Feser <feser@csail.mit.edu>.

[1]In this work, by "program" we mean a program that is represented as source code in a human-readable programming language.

[2]Or in counterexample-guided inductive synthesis (CEGIS) setting, where data instances are added to a monotonically growing active set.

new development, and they have not been studied nearly as extensively as discrete search techniques. We believe it important to study them in different contexts, to better understand where their strengths and weaknesses lie. In this work, we focus on differentiable interpreters for higher level programming languages than have previously been studied.

Specifically, in this work we show that ideas from modern high-level functional programming languages can be used to improve differentiable interpreters. We show how to adapt several key ideas and discuss why they lead to more powerful differentiable interpreters. In total, we develop a functional programming language operating on integers and lists and a corresponding differentiable interpreter. In our empirical evaluation, we show the effects on learning performance of our four modeling recommendations, namely automatic memory management, the use of combinators and if-then-else constructs to structure program control flow, immutability of data, and an application of a simple type system. Our experiments show that each of these features crucially improves program learning over existing baselines.

2. Related Work

Inductive Program Synthesis There has been significant recent interest in synthesizing functional programs from input-output examples in the programming languages community. Synthesis systems generally operate by searching for a program which is correct on the examples, using types or custom deduction rules to eliminate parts of the search space. Among the notable systems: MYTH (Osera & Zdancewic, 2015; Frankle et al., 2016) synthesizes recursive functional programs from examples using types to guide the search for a correct program, λ^2 (Feser et al., 2015) synthesizes data structure manipulating programs structured using combinators using types and deduction rules in its search, ESCHER (Albarghouthi et al., 2013) synthesizes recursive programs using search and a specialized method for learning conditional expressions, and FlashFill (Gulwani, 2011) structures programs as compositions of functions and uses custom deduction rules to prune candidate programs. Our decision to learn functional programs was strongly inspired by this previous work. In particular, the use of combinators to structure control flow was drawn from (Feser et al., 2015). The key difference in our work is the use of differentiable interpreters and gradient-based optimization instead of the discrete search employed in the above works.

Neural Networks Learning Algorithms A number of recent models aim to learn algorithms from input/output data. Many of these augment standard (recurrent) neural network architectures with differentiable memory and simple computation components (e.g. (Graves et al., 2014; Kurach et al., 2016; Joulin & Mikolov, 2015; Neelakantan et al., 2016a; Graves et al., 2016)). The main commonality between differentiable interpreters and these works is the smoothing technique that is used to convert discrete operations into a continuous parameterization. The main difference is that these models use a neural network controller to decide which operation to perform next, whereas differentiable interpreter-based models store the decision of what operations to perform in the source code itself, in conjunction with some kind of *instruction pointer* that denotes where in the source code the current execution is. Zaremba et al. (2016) induce algorithms using a reinforcement learning setup, which avoids the need for the smoothing operations. Like the other above works, it uses a neural network controller to decide the order in which to perform operations. Reed & de Freitas (2016) learn algorithms from strong supervision specifying which operation to perform at each step. Li et al. (2017) weakens the supervision requirement somewhat but still requires supervision in the form of sequences of basic actions and some strong supervision.

None of these works focus on producing source code, and thus we do not expect them to achieve the same strong generalization properties that come from using a source code representation. In experiments, we will show that the source code-based models can generalize from 5 examples, whereas a model using a neural network-based controller fails to do so.

Finally, the most related work is that on differentiable interpreters. Bunel et al. (2016) and Riedel et al. (2016) have used program models similar to assembly (resp. Forth) source code to initialize solutions, and either optimize or complete them. Gaunt et al. (2016) develops a framework that allows comparing differentiable interpreters to several alternative synthesis systems, focusing on low-level programming models including Turing machines, Boolean circuits, and an assembly-like language. Our work differs from these in that we focus on the question of how to design a programming model to improve the performance of differentiable interpreter-based synthesis.

3. Differentiable Interpreters

We begin by reviewing differentiable interpreters and defining a basic program and data representation to introduce the general concepts.

Programs operate on states consisting of an instruction pointer indicating the next instruction to execute, a number of registers holding inputs and intermediate results of executed instructions, and a heap containing memory allocated by the program. We focus on list-manipulating programs,

so we use a heap represented as an array of (data, pointer) value pairs, where pointers are indices of this array, or the special 0 value. To represent a linked list, each cell points to the next cell in the list, except for the last cell, which points to 0. The elements of the list are stored in the data part of each heap location.

Program Representation. All of our models share a basic instruction set, namely the constructor `cons` which stores a (data, pointer) value pair on the heap and returns a pointer to the newly created heap cell, the heap accessors (`head` & `tail`) which return the data (resp. pointer) element of a heap cell, integer addition, increment and decrement (`add`, `inc`, `dec`), integer equality and greater-than comparison (`eq` & `gt`), Boolean conjunction and disjunction (`and` & `or`), common constants (`zero` & `one`), and finally a `noop` instruction. These all have the expected semantics, although we will discuss the behavior of `cons` in detail later.

We pick a maximal integer value M, a number of instructions I, and a number of registers R. In this introductory setting, the size of the heap memory H has to be equal to the maximal integer value M, but we will relax this later.

We limit the length of programs to some value P, and can then encode programs as a sequence of tuples $(o^{(p)}, i^{(p)}, a_1^{(p)}, a_2^{(p)})$, where $i^{(p)} \in [1, I]$ identifies the p-th instruction and $o^{(p)}, a_1^{(p)}, a_2^{(p)} \in [1, R]$ its output and argument registers respectively.

Interpreter. An interpreter takes a program description and executes it. In our setting we limit the number of execution timesteps T and keep a program state $s^{(t)} = (p^{(t)}, r_1^{(t)} \dots r_R^{(t)}, h_1^{(t)} \dots h_H^{(t)})$ for each timestep t, where $p^{(t)} \in [1, P]$ is an instruction pointer indicating which instruction to execute next, $r_*^{(t)}$ are the values of registers, and $h_*^{(t)}$ are the values of the (pointer, data) cells in the heap. The interpreter works by iteratively updating the program state by executing the next instruction, which is determined by the instruction pointer. For example, executing `(`o`, add,` a_1`,` a_2`)` on a state at timestep t yields the following registers at the next timestep:

$$r_u^{(t+1)} = \begin{cases} r_{a_1}^{(t)} + r_{a_2}^{(t)} \quad mod\ M & \text{if } u = o \\ r_u^{(t)} & \text{otherwise.} \end{cases} \qquad \forall u \in [1, R]$$

Differentiable Interpreter. To make an interpreter differentiable, we follow earlier work (e.g. (Graves et al., 2014; Kurach et al., 2016; Bunel et al., 2016; Riedel et al., 2016; Gaunt et al., 2016)) and replace all discrete values with probability distributions over their values and lift all operations to operate on probability distributions instead of discrete values by averaging over all the possibilities with weights given by the probability distributions. For example, if $\eta(s^{(t)}, (o, i, a_1, a_2))$ computes the state obtained by executing the instruction (o, i, a_1, a_2), we can compute the next state $s^{(t+1)}$ as follows, where $[\![x = n]\!]$ denotes the probability that a variable x encoding a discrete probability distribution assigns to the value n.

$$s^{(t+1)} = \sum_{\substack{p \in [1,P], i \in [1,I], \\ o, a_1, a_2 \in [1,R]}} \begin{array}{l} [\![p^{(t)} = p]\!] \cdot [\![o^{(p)} = o]\!] \cdot [\![i^{(p)} = i]\!] \\ \cdot [\![a_1^{(p)} = a_1]\!] \cdot [\![a_2^{(p)} = a_2]\!] \\ \cdot \eta(s^{(t)}, (o, i, a_1, a_2)) \end{array} \tag{1}$$

We can then differentiate with respect to the parameters of these probability distributions. To do program synthesis, we randomly initialize the representation of the program (a collection of probability distributions over discrete values) and then use gradient ascent to find distributions over program variables that (locally) maximize the log probability of observed (discrete) output values given observed (discrete) input values. In a bit more detail, our aim is to learn the program parameters $(o^{(p)}, i^{(p)}, a_1^{(p)}, a_2^{(p)})$ such that program "evaluation" according to (1) starting on a state $s^{(0)}$ initialized to an example input yields the target output in $s^{(T)}$. For scalar outputs such as a sum of values, our objective is simply to minimize the cross-entropy between the distribution in the *output* register $r_R^{(T)}$ and a point distribution with all probability mass on the correct output value. In practice, we developed our models in Terpret. See Gaunt et al. (2016) for more details.

4. Differentiable Functional Program Interpreters

In the following, we will discuss how and why to add functional programming features to differentiable interpreters. We start with the simple assembly-like language from the previous section and progress to a differentiable version of a simple functional programming language. We begin by developing an observation model for list-structured data, and then we make four modeling recommendations inspired by functional programming. Empirically, we will demonstrate the benefit of these recommendations in Sect. 5.

4.1. Observation Model for List Data

We first discuss a new observation model for list-structured data. Handling list outputs is more complex than scalar values, as there are many ways for a target list to be represented in memory. Intuitively, we will traverse the heap from the returned heap address until reaching the end of a linked list, recording the list elements as we go, and then we will observe the sequence of elements that was recorded. To formalize this intuition, let ${}^d h_k^{(T)}$ (resp. ${}^p h_k^{(T)}$) denote the data (resp. pointer) information in the

heap cell at address k at the final state of the evaluation. We then compute the traversal sequences of list element values $v_1, \ldots, v_H$ and addresses $a_1, \ldots, a_H$ as follows.

$$a_i = \begin{cases} r_R^{(T)} & \text{if } i = 1 \\ \sum_{a \in [1,H]} [\![a_{i-1} = a]\!] \cdot {}^{p}h_a^{(T)} & \text{otherwise} \end{cases}$$

$$v_i = \sum_{a \in [1,H]} [\![a_i = a]\!] \cdot {}^{d}h_a^{(T)}$$

The probability that the computed output list is equal to an expected output list $[\bar{v}_1, \ldots, \bar{v}_k]$ is then $[\![a_{k+1} = 0]\!] \cdot \sum_{i=1}^{k} [\![v_i = \bar{v}_i]\!]$.

4.2. Structured Control Flow

Our baseline program model corresponds closely to an assembly language as used in earlier work (Bunel et al., 2016), resulting in a program model as shown in Fig. 1, where boxes correspond to learnable parameters. We extend our instruction set with jump-if-zero (`jz`), jump-if-not-zero (`jnz`) and `return` instructions. Our assembly program representation also includes a "branch" parameter b specifying the new value of the instruction pointer for a successful conditional branch. To learn programs in this language, the model must learn how to create the control flow that it needs using these simple conditional jumps. Note that the instruction pointer suffers from the same problems as the stack pointer above, i.e., uncertainty about its value blurs together the effects of many possible program executions.

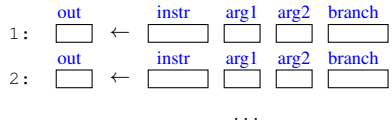

Figure 1

A key difference between hardware-level assembly languages and higher-level programming languages is that higher-level languages structure control flow using loops, conditional statements, and procedures, as raw `gotos` are famously considered harmful (Dijkstra, 1968). Functional languages go a step further and leverage higher-order functions to abstract over common control flow patterns such as iteration over a recursive data structure. In an imperative language, such specialized control flow is often repeated and mixed with other code. In the differentiable interpreter setting, structured control flow gives an additional benefit, which is that it reduces uncertainty in the instruction pointer.

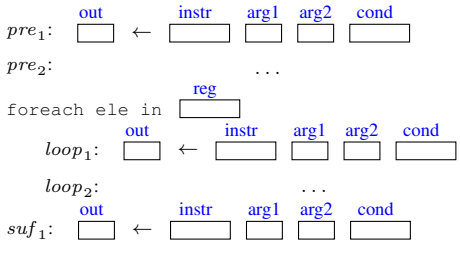

Figure 2

To introduce structured control flow, we replace raw jumps with an `if-then-else` instruction and an explicit `foreach` loop that is suited for processing lists. We restrict our model to a prefix of instructions, a loop which iterates over a list, and a suffix of instructions. The parameters for instructions in the loop can access an additional register that contains the value of the current list element. An outline of such a program is shown in Fig. 2. This removes uncertainty about the value of the instruction pointer, as each time step corresponds to exactly one "line" in the program template. To implement this behavior in practice, we unroll the loop for a fixed number of iterations derived from the bound on the size of the input, which ensures that every input list can be processed.

For the `if-then-else` instruction, we extend the instruction representation with a "condition" parameter $c \in [1, R]$ and let the evaluation of `if-then-else` yield its first argument when the register c is non-zero and the second argument otherwise. An overview of the structure of such programs is displayed above.

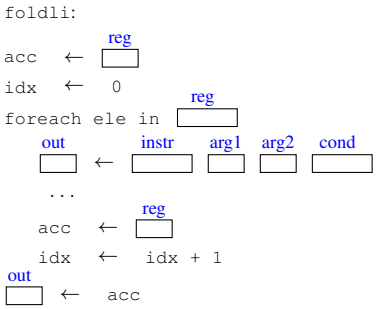

Figure 3

We note that while fixing the iteration over the list elements is already helpful, learning most list-processing programs requires the model to repeatedly infer the concepts of creating a new list, aggregating results and keeping track of the current list index. In functional programming languages, such regular patterns are encapsulated in combinators. Thus, in a second model, we replace the simple `foreach` loop with three combinators: `mapi` creates a new list by applying a function to each element of the input list, `zipWithi` creates a new list by iterating over two lists in parallel and applying a function to both elements, and `foldli` computes a value by iterating over all list elements and applying a function to the current list element and the value computed so far. A program model using the `foldli` combinator is shown in Fig. 3. The `i` suffix indicates that these combinators additionally provide the index of the current list element (the precise semantics of the combinators are presented in Sect. A.2).

Recommendation (L): Instead of raw jumps, use loop and if-then-else templates.

4.3. Memory Management

Most modern programming languages eschew manual memory management and pointer manipulation where possible. Instead, creation of heap objects automatically gen-

erates an appropriate pointer to fresh memory. Similarly, built-in constructs allow access to fields of objects, instead of requiring pointer arithmetic. Both of these choices move program complexity into the fixed implementation of a programming language, making it easier to write correct programs.

As the programs we want to learn need to construct new lists, we explored two memory allocation mechanism that provides fresh cells. First, we attempted to use a *stack pointer* sp which always points to the next free memory cell, and fixing a maximum stack size H. Whenever a memory cell is allocated (i.e., a `cons` instruction is executed), the stack pointer is incremented.

There are two problems with this allocation mechanism. 1) We must maintain a copy of the heap and stack pointer for each timestep t. For large values of T or H, this significantly increases the size of the model. 2) Uncertainty about whether an instruction is `cons` translates into uncertainty about the precise value of the stack pointer, as each call to `cons` changes sp. This uncertainty causes cells holding results from different instructions in the stack to blur together, despite the fact that cells are immutable once created. As an example, consider the execution of two instructions, where the first is `cons 1 0` with probability 0.5 and `noop` otherwise, and the second is `cons 2 0` with probability 0.5 and `noop` otherwise. After executing starting with $sp = 1$, the value of sp will be blurred across three values 1, 2 and 3 with probabilities 0.25, 0.5 and 0.25. Similarly, the value of the first heap cell will be 0 (the default) with probability 0.25, 1 with probability 0.5 and 2 with probability 0.25. This blurring effect becomes stronger with longer programs, and we found that it substantially impacted learning.

Both of these problems can be solved by transitioning to a fully immutable representation of the heap. In this variant, we allocate and initialize one heap cell per timestep, i.e., we set $H = T$. If the current instruction is a `cons`, the appropriate values are filled in, otherwise both data and pointer value are set to a default value (in our case, 0). This eliminates the issue of blurring between outputs of different instructions. The values of a heap cell may still be uncertain as they inherit uncertainty about the executed instructions and the values of arguments, but depend only on the operations at one timestep. Because there is no uncertainty about whether to fill in a heap cell, we keep a single copy of the heap and use the current timestep t as the stack pointer. While this modification requires a larger domain to store pointers, we found not copying the stack significantly reduces memory usage during training of our models.

Recommendation (F): Use fixed heap memory allocation deterministically controlled by the model.

4.4. Immutable Data

In functional programming, functions are expected to not have side effects, and all data is immutable. This helps programmers reason about their code, as it eliminates the possibility that a variable might be left uninitialized or accessed in an inconsistent state. Moreover, no data is ever "lost" by being overwritten or mutated.

In training initial models, we observed that many random initializations of the program parameters would overwrite input data or important intermediate results. In models with combinators that provide a way to accumulate result values, we can sidestep this issue by making registers immutable. To do so, we create one register per timestep, and fix the output of each instruction to the register for its timestep. Parameters for arguments then range over all registers initialized in prior timesteps, with an exception for the closures executed by a combinator. Here, each instruction only gets access to the inputs to the closure, values computed in the prefix, and registers initialized by preceding instructions in the same loop iteration. As in the heap allocation case, we can avoid keeping a copy of all registers for every timestep, and instead share these values over all steps. A somewhat unintuitive consequence is that this strategy reduces memory footprint of the model.

Recommendation (I): Use immutable registers by deterministically choosing where to store outputs.

4.5. Types

In programming languages, expressive type systems are used to protect programmers from writing programs that will fail. Practically, a type checker is able to rule out many syntactically correct programs that are certain to fail at runtime, and thus restricts the space of valid programs. When training initial models, we found that for many initializations, training would fall into local minima corresponding to ill-typed programs, e.g., where references to the heap would be used in integer additions. We expect the learned program to be well-typed, so we introduce a simple type system. We explored two approaches to adding a type system.

Our first attempt extended the objective with a penalty for type errors. In our programs, we use three simple types of data—integers, pointers and booleans—as well as a special type, $\perp$, which represents type errors. We extended the program state to contain an additional element ${}^t r$ for each register, encoding its type. Each instruction then not only computes a value that is assigned to the target register, but also a type for the target register. Most significantly, if one of the arguments has an unsuitable type (e.g., an integer in place of a pointer), the resulting type is $\perp$. We then extended our objective function to add a penalty for values

with type $\perp$. Unfortunately, this changed objective function had neither a positive nor negative effect on our experiments, so it seems that optimizing for the correct type is redundant when we are already optimizing for the correct return value.

In our second attempt, rather than penalizing ill-typed programs, we prevent programs from accessing ill-typed data by construction. We augment our register representation by adding an integer, pointer, and Boolean slot to each register, so each register can hold a separate value of each type. Instructions which read from registers now read from the slot corresponding to the type of the argument. When writing to a register, we write to the slot corresponding to the instruction's return type, and set the other slots to a default value 0. This prevents any ill-typed sequence of instructions, i.e., it is now impossible to, for example, increment a pointer value or to fill the pointer part of a heap cell with a non-pointer value. Furthermore, this modification allows us to set the heap size H to a value different from the maximal integer M because it allows pointers and integers to have different maximum values.

Recommendation (T): Use different storage for data of different types.

5. Experiments

We have empirically evaluated our modeling recommendations on a selection of program induction tasks of increasing complexity, ranging from straight-line programs to problems with loops and conditional expressions. All of our models are implemented in Terpret (Gaunt et al., 2016) and we learn using Terpret's TENSORFLOW (Abadi et al., 2015) backend.

For all tasks, three groups of five input/output example pairs were sampled as training data and another 25 input/output pairs as test data. For each group of five examples, training was started from 100 random initializations of the model parameters. After training for 3500 epochs (tests with longer training runs showed no significant changes in the outcomes), the learned programs were tested by discretizing all parameters and comparing program outputs on test inputs with the expected values. We perform 300 runs per model and task, and report only the ratio of successful runs. A run is successful if the discretized program returns the correct result on all five training and 25 test examples.[3] The ratio of runs converging to zero loss on the training examples was within 1% of the number of successful runs, i.e., very few found solutions failed to generalize.

[3]We inspected samples of the obtained programs as well and verified that they were indeed correct solutions. See Sect. A.3 for some of the learned programs.

We performed a cursory exploration of hyperparameter choices, sampling 100 hyperparameter settings (choosing optimization algorithm, learning rate, gradient noise, (decay of) entropy bonus, and gradient clipping) and tested their effect on two simple tasks. We ran the remaining experiments with the best configuration obtained by this process: the RMSProp optimization algorithm, a learning rate of 0.1, clipped gradients at 1, and no gradient noise.

We consider the ratio of successful runs as earlier work has identified this as a significant problem. For example, (Neelakantan et al., 2016b) reports that even after a (task-specific) "large grid search" of hyperparameters, the Neural Random Access Machine converged only in 5%, 7% and 22% of random restarts on the considered tasks. Similar observations were made in (Kaiser & Sutskever, 2016; Bunel et al., 2016; Gaunt et al., 2016) for related program learning models.

In our experiments we evaluate the effect of the choices discussed in Sect. 4, comparing seven model variants in total. We call our initial assembly model A and its variation with a fixed memory allocation scheme A+F. All other models use the fixed memory allocation scheme. The extension of the assembly model with a built-in `foreach` loop is called A+L. The model including predefined combinators is called C, where C+I (resp. C+T) are its extensions with immutable registers (resp. typed registers). Finally, C+T+I combines all of these, making it a simple end-to-end differentiable functional programming language.

As baselines, we consider λ^2 (Feser et al., 2015), a strong program synthesis baseline from programming languages research, and an implementation of the Neural Random Access Machine (NRAM) (Kurach et al., 2016). We chose λ^2 because of its built-in support for list-processing programs. As λ^2 is deterministic, we only report a success ratio of either 1 (if a program matching all input-output examples was generated) or 0 (if no so such program was generated) for all experiments. We ran λ^2 with a timeout of 600 seconds. We give a detailed description of our NRAM model and experimental results separately in Sect. 5.4.

5.1. Straight-line programs

In our first experiment, we consider two families of simple problems—solvable with straight-line programs—to study the interaction of our modeling choices with program length. Our first benchmark task is to duplicate a scalar input a fixed number k times to create a list of length k. Our second benchmark is to retrieve the k-th element of a list, again fixing k beforehand (we will consider a generalization of this task where k is a program input later). We set the hyperparameters for all models to allow 11 statements, i.e., for A and A+F we have set the program length to 11, and for the A+L and C* models we have set the prefix and

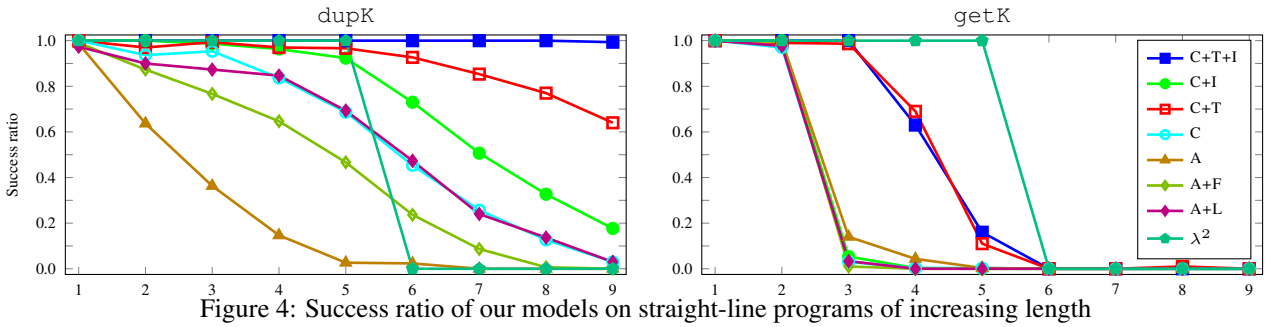

Figure 4: Success ratio of our models on straight-line programs of increasing length

loop length to 0 and the suffix length to 11. For models where the number of registers does not depend on the number of timesteps, we use 3 registers, with one initialized to the input. This allows for $\sim 10^{39}$ programs in the A, A+F, C+I, and C+T+I models and for $\sim 10^{28}$ programs in the remaining models. These parameters were chosen to be slightly larger than required by the largest program to be learned. For all of our experiments, the maximal integer M was set to 20 for models where possible (i.e., for A, C+T+I, C+T), and to H (derived from T, coming to 22) for the others.[4]

We evaluated all of our models following the regime discussed above and present the results in Fig. 4 for k values from 1 to 9. The difference between A and A+F on the `dupK` task illustrates the significance of **Recommendation (F)** to fix the memory allocation scheme. Following **Recommendation (T)** to separate values of different types improves the results on both tasks, as the differences between C+T+I (resp. C+T) and C+I (resp. C) illustrate.

5.2. Simple loop programs

In our second experiment, we compare our models on three simple list algorithms: computing the length of a list, reversing a list and summing a list. Model parameters have been set to allow 6 statements for the A and A+F models, and empty prefixes, empty suffixes, and 2 instructions in the loop for the other models. For models where the number of registers does not depend on the number of timesteps, we use 4 registers, with one initialized to the input.

The results of our evaluation are displayed in Tab. 1, starkly illustrating **Recommendation (L)** to use predefined loop structures. We speculate that learning explicit jump targets is extremely challenging because changes to the parameters controlling jump target instructions have outsized effects on all computed (intermediate and output) values. On the other hand, models that could choose between different list iteration primitives were able to find programs for all tasks. We again note the effect of **Recommendation (T)** to separate values of different types on the success ratios for the `len` and `sum` examples, and the effect of **Recommendation (I)** to avoid mutable data on results for `len` and `rev`.

[4] We also experimented with varying the value of M. Choices over 20 showed no significant differences to smaller values.

5.3. Loop Programs

In our main experiment, we consider a larger set of common list-manipulating tasks (such as checking if all/one element of a list is greater than a bound, retrieving a list element by index, finding the index of a value, and computing the maximum value). Descriptions of all tasks are shown in Sect. A.1 in the appendix. We do not show results for the A and A+F models, which always fail. We set the parameters for the remaining models to $M = 32$ where possible ($M = H = 34$ for the others), the length of the prefix to 1, the length of the closure / loop body to 3 and the length of the suffix to 2. Again, these parameters are slightly larger than required by the largest program to be learned.

Table 2: Success ratios for full set of tasks.

Program	C+T+I	C+T	C+I	C	A+L	λ^2
len	**98.67**	96.33	0.67	0.33	0.00	100.00
rev	**18.00**	10.33	2.67	8.33	9.67	100.00
sum	38.00	**38.33**	1.00	0.00	10.00	100.00
allGtK	0.00	0.00	0.00	**0.33**	0.00	100.00
exGtK	**3.00**	1.00	0.67	0.00	0.67	100.00
findLastIdx	**0.33**	0.00	0.00	0.00	0.00	0.00
getIdx	**1.00**	0.00	0.00	0.00	0.00	0.00
last2	0.00	8.00	0.00	2.00	**23.00**	0.00
mapAddK	**100.00**	98.00	**100.00**	95.67	0.00	100.00
mapInc	**99.67**	98.00	99.33	97.00	0.00	100.00
max	2.33	**5.67**	0.00	0.00	0.33	100.00
pairwiseSum	43.33	32.33	**43.67**	33.67	0.00	100.00
revMapInc	0.00	0.67	0.00	0.00	**6.33**	100.00

The results for our experiments on these tasks are shown in Tab. 2. Note the changed results of the examples from Sect. 5.2, as the change in model parameters has increased the size of the program space from $\sim 10^7$ to $\sim 10^{20}$. The comparison to the A+L model show the value of built-in iteration and aggregation patterns. The choice between immutable and mutable registers is less clear here, seemingly dampened by other influences. An inspection of the generated programs (eg. Fig. 10 in the appendix) reveals that

Table 1: Success ratio for experiments on simple loop-requiring tasks.

Program	C+T+I	C+T	C+I	C	A	A+F	A+L	λ^2
len	**100.00**	75.00	**100.00**	43.67	0.00	0.00	15.67	100.00
rev	48.33	32.67	46.33	41.33	0.00	0.00	**86.33**	100.00
sum	**91.67**	41.00	88.33	30.67	0.00	0.00	32.67	100.00

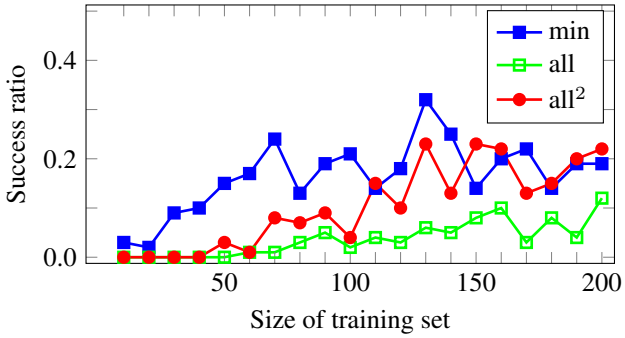

Figure 5: Success ratio of NRAM on len

mutability of registers can sometimes be exploited to find elegant solutions. Overall, it may be effective to combine both approaches, using a few mutable "scratch value" registers *and* immutable default output registers for each statement.

5.4. Comparison with NRAM

Our hypothesis was that the NRAM controller would fail to generalize when trained on a small set of input-output examples. As we believe that programming by example use cases usually operate on small numbers of examples, we explicitly tested this hypothesis on the most simple list-processing task len. While not considered in (Kurach et al., 2016), it is slightly simpler than the **ListK** and **ListSearch** tasks that are classified as "Hard Tasks" there. We note that while very different, the NRAM model implements some of our recommendations: The RNN-like structure imposes a basic loop structure, and the output of each module (i.e., instruction in our setting) is stored in a fixed register that is immutable during a loop iteration.

For our experiments, we simplified the NRAM model significantly and only provided the modules `READ`, `ZERO`, `ONE`, `INC`, `ADD`, and `DEC` operating on integers. Most notably, the absence of `WRITE` means that the heap remains unchanged during program execution. We considered three related models: "min", in which *only* the modules required in each iteration of a perfect solution are available,[5] "all" in which all modules are available once,[6] and "all^2", in which all modules are available twice. We fixed the maximal length of input lists to 5, and the maximal integer to 11.

For each model, we performed an extensive random hyperparameter search (choosing the optimizer, learning rate, momentum, size of the LSTM cell in the NRAM controller, gradient noise, gradient clipping parameters, entropy bonus, dropout probability), using 20 random restarts on 200 input-output pairs, with validation and test sets of (disjoint) 50 examples each. We stopped training after 200 epochs, or if the accuracy on the validation set reached 100% (most successful runs stopped after few epochs). A run was counted as successful if the accuracy of the discretized model on the test was 100%, i.e., if the trained model successfully generalized to unseen data of the same size. For the best hyperparameter settings, we then varied the size of the training set from 200 down to 10 in increments of 10, keeping the size of the validation set at a quarter of the size of the training set. The results are displayed in Fig. 5. We note that only the "min" model, where the NRAM controller chooses between only 600 syntactically different programs in each iteration, has any success with training sets smaller than 50. Thus, our experiments confirm our hypothesis that NRAM-like models fail to generalize from little data.

6. Discussion and Future Work

We have discussed a range of modeling choices for end-to-end differentiable programming languages and made four design recommendations. Empirically, we have shown these recommendations to significantly improve the success ratio of learning programs from input/output examples, and we expect these results to generalize to similar models attempting to learn programs.

In this paper, we only consider list-manipulating programs, but we are interested in supporting more data structures, such as arrays (which should be a straightforward extension) and associative maps. We also only support loops over lists at this time, but are interested in extending our models to also have built-in support for loops counting up to (and down from) integer values. A generalization of this concept would be an extension allowing the learning and use of recursive functions. Recursion is still more structured than raw goto calls, but more flexible than the combinators that we currently employ. An efficient implementation of recursion is a challenging research problem,

[5] For len, this was `INC`, `INC`, `READ`.

[6] Note that this is the most challenging setting, as this requires the controller to choose different instructions in alternating iterations: One setting to advance the list pointer, and one to increment the length counter.

but it could allow significantly more complex programs to be learned. Modeling recursion in an end-to-end differentiable language could allow us to build libraries of (learned) differentiable functions that can be used in later synthesis problems.

However, we note that with few exceptions on long straight-line code, λ^2 performs better than all of our considered models, and is able to synthesize programs in milliseconds. We see the future of differentiable programming languages in areas in which deterministic tools are known to perform poorly, such as the integration of perceptual data, priors and "soft" side information such as natural language hints about the desired functionality.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mané, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viégas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Albarghouthi, Aws, Gulwani, Sumit, and Kincaid, Zachary. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pp. 934–950, 2013.

Bunel, Rudy, Desmaison, Alban, Kohli, Pushmeet, Torr, Philip H. S., and Kumar, M. Pawan. Adaptive neural compilation. In *Proceedings of the 29th Conference on Advances in Neural Information Processing Systems (NIPS)*, 2016. To appear.

Dijkstra, Edsger W. Letters to the Editor: Go to Statement Considered Harmful. 11(3):147–148, 1968. ISSN 0001-0782. doi: 10.1145/362929.362947.

Feser, John K., Chaudhuri, Swarat, and Dillig, Isil. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 229–239, 2015.

Frankle, Jonathan, Osera, Peter-Michael, Walker, David, and Zdancewic, Steve. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 802–815. ACM, 2016. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837629.

Gaunt, Alexander L., Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016. URL `http://arxiv.org/abs/1608.04428`.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL `http://arxiv.org/abs/1410.5401`.

Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pp. 317–330. ACM, 2011.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Proceedings of the 28th Conference on Advances in Neural Information Processing Systems (NIPS)*, pp. 190–198, 2015.

Kaiser, Łukasz and Sutskever, Ilya. Neural GPUs learn algorithms. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. URL http://arxiv.org/abs/1511.08228.

Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. URL http://arxiv.org/abs/1511.06392.

Li, Chengtao, Tarlow, Daniel, Gaunt, Alexander L., Brockschmidt, Marc, and Kushman, Nate. Neural program lattices. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Neelakantan, Arvind, Le, Quoc V., and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016a.

Neelakantan, Arvind, Vilnis, Luke, Le, Quoc V., Sutskever, Ilya, Kaiser, Lukasz, Kurach, Karol, and Martens, James. Adding gradient noise improves learning for very deep networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016b. URL http://arxiv.org/abs/1511.06807.

Osera, Peter-Michael and Zdancewic, Steve. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pp. 619–630. ACM, 2015. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738007.

Reed, Scott E. and de Freitas, Nando. Neural programmer-interpreters. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. URL http://arxiv.org/abs/1511.06279.

Riedel, Sebastian, Bosnjak, Matko, and Rocktäschel, Tim. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016. URL http://arxiv.org/abs/1605.06640.

Zaremba, Wojciech, Mikolov, Tomas, Joulin, Armand, and Fergus, Rob. Learning simple algorithms from examples. In *Proceedings of the 33nd International Conference on Machine Learning (ICML)*, pp. 421–429, 2016.

A. Appendix

A.1. Experiment Tasks

Name	Description
`len`	Return the length of a list.
`rev`	Reverse a list.
`sum`	Sum all elements of a list.
`allGtK`	Check if all elements of a list are greater than k.
`exGtK`	Check if at least one element of a list is greater k.
`findLastIdx`	Find the index of the last list element which is equal to v.
`getIdx`	Return the kth element of a list.
`last2`	Return the 2nd to last element of a list.
`mapAddK`	Compute list in which k has been added to each element of the input list.
`mapInc`	Compute list in which each element of the input list has been incremented.
`max`	Return the maximum element of a list.
`pairwiseSum`	Compute list where each element is the sum of the corresponding elements of two input lists.
`revMapInc`	Reverse a list and increment each element.

Our example tasks for loop based programs. "Simple" tasks are above the line.

A.2. Combinators

Semantics of `foldli`, `mapi`, `zipwithi` in a Python-like language:

function FOLDLI($list, acc, func$)
  $idx \leftarrow 0$
  for ele **in** $list$ **do**
    $acc \leftarrow func(acc, ele, idx)$
    $idx \leftarrow idx + 1$
  return acc

function ZIPWITHI($list_1, list_2, func$)
  $idx \leftarrow 0$
  $ret \leftarrow [\,]$
  for ele_1, ele_2 **in** $list_1, list_2$ **do**
    $ret \leftarrow append(ret, func(ele_1, ele_2, idx))$
    $idx \leftarrow idx + 1$
  return ret

function MAPI($list, func$)
  $idx \leftarrow 0$
  $ret \leftarrow [\,]$
  for ele **in** $list$ **do**
    $ret \leftarrow append(ret, func(ele, idx))$
    $idx \leftarrow idx + 1$
  return ret

A.3. Selected Solutions

We show example results of our training in Figs. 6-18. Note that these are the actual results produced by our system, and have only been slightly edited for typesetting. Finally, we have colored statements that a simple program analysis can identify as not contributing to the result in gray.

```
r0 ← l
r1 ← k
r2 ← r0 ∨ r0
r1 ← foldli r0 r0 (λ ele acc idx →
      r0 ← ele > r1
      r2 ← head acc
      r2 ← r0 ∧ acc
      r2)
r2 ← r1 ∧ r0
r1 ← r1
return r2
```

Figure 6: A solution to `allGtK` in the C model. Code in gray is dead.

```
let r0 = l in
let r1 = k in
let r2 = (r0 = r1) in
let r3 = foldli r0 r0 (λ ele acc idx →
      let c0 = acc ∨ acc in
      let c1 = ele > r1 in
      let c2 = c0 ∨ c1 in
      c2) in
let r4 = r3 ∨ r3 in
let r5 = r3 ∧ r2 in
return r4
```

```
r0 ← l
r1 ← k
r2 ← r2 = r1
for ele in r0 do
    r0 ← if r2 then ele else r1
    r0 ← ele > r1
    r2 ← r2 ∨ r0
r2 ← r2 ∨ r0
r1 ← r2 ∨ r2
return r2
```

Figure 7: Solutions to `exGtK` in the C+T+I and A+L models.

```
let r0 = l in
let r1 = e in
let r2 = r0 + 1 in
let r3 = foldli r0 r2 (λ ele acc idx →
      let c0 = if r2 then idx else r1 in
      let c1 = (r1 = ele) in
      let c2 = if c1 then idx else acc in
      c2) in
let r4 = r3 + 1 in
let r5 = r2 in
return r3
```

Figure 8: A solution to `findLastIdx` in the C+T+I model.

```
let r0 = l in
let r1 = k in
let r2 = head r0 in
let r3 = foldli r0 r2 (λ ele acc idx →
      let c0 = (r1 = idx) in
      let c1 = if c0 then ele else acc in
      let c2 = if idx then idx else c0 in
      c1) in
let r4 = head r0 in
let r5 = tail r0 in
return r3
```

Figure 9: A solution to `getIdx` in the C+T+I model.

```
r0 ← l
r1 ← 0
r2 ← nil
r2 ← foldli r0 r1 (λ ele acc idx →
    r0 ← acc
    r2 ← r1
    r1 ← ele
    r2) in
r0 ← r2 + r2
r0 ← r0 + 1
return r2
```

```
r0 ← l
r1 ← 0
r2 ← cons r0 r0
for (ele1, ele2) in (r0, r2) do
    r2 ← if r2 then ele2 else ele2
    r1 ← r2 − 1
    r1 ← head r0
r1 ← if r1 then r2 else r0
r0 ← if r2 then r0 else r1
return r2
```

Figure 10: Solutions to `last2` in the C+T and A+L models.

```
let r0 = l in
let r1 = 0 in
let r2 = tail r0 in
let r3 = foldli r0 r0 (λ ele acc idx →
    let c0 = idx + 1 in
    let c1 = if r1 then r2 else c0 in
    let c2 = c0 = ele in
    c0) in
let r4 = if r2 then r3 else r3 in
let r5 = r3 + r2 in
return r3
```

Figure 11: A solution to `len` in the C+T+I model.

```
let r0 = l in
let r1 = k in
let r2 = if r1 then r0 else r0 in
let r3 = mapi r0 (λ ele idx →
    let c0 = ele − 1 in
    let c1 = c0 − 1 in
    let c2 = r1 + ele in
    c2) in
let r4 = r3 in
let r5 = r3 in
return r3
```

Figure 12: A solution to `mapAddK` in the C+T+I model.

```
let r0 = l in
let r1 = 0 in
let r2 = r0 in
let r3 = mapi r0 (λ ele idx →
    let c0 = if r1 then ele else acc in
    let c1 = ele + 1 in
    let c2 = r1 in
    c1) in
let r4 = cons r3 r0 in
let r5 = r4 in
return r3
```

Figure 13: A solution to `mapInc` in the C+T+I model.

```
let r0 = l in
let r1 = 0 in
let r2 = tail r0 in
let r3 = foldli r0 r0 (λ ele acc idx →
    let c0 = acc > ele in
    let c1 = acc in
    let c2 = if c0 then acc else ele in
    c2) in
let r4 = r2 − 1 in
let r5 = head r2 in
return r3
```

```
r0 ← l
r1 ← 0
r2 ← r0 − 1
for (ele1, ele2) in (r0, r0) do
    r0 ← ele1
    r1 ← ele1 > r2
    r2 ← if r1 then ele1 else r2
r0 ← r0 + r0
r0 ← cons r2 r2
return r2
```

Figure 14: Solutions to `max` in the C+T+I and A+L models.

```
let r0 = l1 in
let r1 = l2 in
let r2 = if r1 then r0 else r1 in
let r3 = zipWithi r1 r0 (λ ele1 ele2 idx →
    let c0 = ele1 + ele2 in
    let c1 = ele2 − 1 in
    let c2 = idx − 1 in
    c0) in
let r4 = if r0 then r3 else r1 in
let r5 = if r4 then r2 else r1 in
return r3
```

Figure 15: A solution to `pairwiseSum` in the C+T+I model.

```
let r0 = l in
let r1 = 0 in
let r2 = cons r0 r0 in
let r3 = foldli r0 r1 (λ ele acc idx →
    let c0 = cons ele acc in
    let c1 = cons acc acc in
    let c2 = cons ele acc in
    c2) in
let r4 = if r2 then r3 else r2 in
let r5 = cons r4 r3 in
return r3
```

```
r0 ← l
r1 ← 0
r2 ← tail r1
for ele1 in r0 do
    r1 ← cons ele2 r0
    r2 ← cons ele1 r2
    r1 ← tail r0
r0 ← tail r0
r0 ← cons r2 r1
return r2
```

Figure 16: Solutions to `rev` in the C+T+I and A+L models.

```
r0 ← l
r1 ← 0
r1 ← tail r0
r2 ← foldli r0 r2 (λ ele acc idx →
    r2 ← ele + 1
    r0 ← cons r2 acc
    r2 ← cons r2 acc
    r2)
r1 ← cons r2 r1
r0 ← cons r2 r0
return r2
```

```
r0 ← l
r1 ← 0
r1 ← 1
for ele1 in r0 do
    r0 ← ele1 + 1
    r1 ← 1
    r2 ← cons r0 r2
r1 ← cons r0 r2
r1 ← cons r0 r2
return r2
```

Figure 17: Solutions to `revMapInc` in the C+T and A+L models.

```
let r0 = l in
let r1 = 0 in
let r2 = r0 in
let r3 = foldli r0 r0 (λ ele acc idx →
    let c0 = acc + r0 in
    let c1 = acc + ele in
    let c2 = if r0 then idx else r1 in
    c1) in
let r4 = r2 + 1 in
let r5 = r3 − 1 in
return r3
```

```
r0 ← l
r1 ← 0
r1 ← if r2 then r1 else r0
for ele1 in r0 do
    r2 ← ele1 + r2
    r1 ← cons r2 r0
    r1 ← ele1 + ele2
r0 ← r1 + r1
r0 ← r2 + 1
return r2
```

Figure 18: Solutions to `sum` in the C+T+I and A+L models.