TerpreT: A Probabilistic Programming Language for Program Induction

Alexander L. Gaunt
Microsoft Research
t-algaun@microsoft.com

Marc Brockschmidt
Microsoft Research
mabrocks@microsoft.com

Rishabh Singh
Microsoft Research
risin@microsoft.com

Nate Kushman
Microsoft Research
nkushman@microsoft.com

Pushmeet Kohli
Microsoft Research
pkohli@microsoft.com

Jonathan Taylor
perceptiveIO*
jtaylor@perceptiveio.com

Daniel Tarlow
Microsoft Research
dtarlow@microsoft.com

arXiv:1608.04428v1 [cs.LG] 15 Aug 2016

Abstract

We study machine learning formulations of inductive program synthesis; that is, given input-output examples, we would like to synthesize source code that maps inputs to corresponding outputs. Our aims in this work are to develop new machine learning approaches to the problem based on neural networks and graphical models, and to understand the capabilities of machine learning techniques relative to traditional alternatives, such as those based on constraint solving from the programming languages community.

Our key contribution is the proposal of TerpreT, a domain-specific language for expressing program synthesis problems. TerpreT is similar to a probabilistic programming language: a model is composed of a specification of a program representation (declarations of random variables) and an interpreter that describes how programs map inputs to outputs (a model connecting unknowns to observations). The inference task is to observe a set of input-output examples and infer the underlying program. TerpreT has two main benefits. First, it enables rapid exploration of a range of domains, program representations, and interpreter models. Second, it separates the model specification from the inference algorithm, allowing proper like-to-like comparisons between different approaches to inference. From a single TerpreT specification we can automatically perform inference using four different back-ends that include machine learning and program synthesis approaches. These are based on gradient descent (thus each specification can be seen as a differentiable interpreter), linear program (LP) relaxations for graphical models, discrete satisfiability solving, and the Sketch program synthesis system.

We illustrate the value of TerpreT by developing several interpreter models and performing an extensive empirical comparison between alternative inference algorithms on a variety of program models. Our key, and perhaps surprising, empirical finding is that constraint solvers dominate the gradient descent and LP-based formulations. We conclude with some suggestions on how the machine learning community can make progress on program synthesis.

1 Introduction

Learning computer programs from input-output examples, or Inductive Program Synthesis (IPS), is a fundamental problem in computer science, dating back at least to Summers (1977) and Biermann (1978). The field has produced many successes, with perhaps the most visible example being the FlashFill system in Microsoft Excel (Gulwani, 2011; Gulwani et al., 2012).

*Work done while author was at Microsoft Research.

Learning from examples is also studied extensively in the statistics and machine learning communities. Trained decision trees and neural networks could be considered to be synthesized computer programs, but it would be a stretch to label them as such. Relative to traditional computer programs, these models typically lack several features: (a) key functional properties are missing, like the ability to interact with external storage, (b) there is no compact, interpretable source code representation of the learned model (in the case of neural networks), and (c) there is no explicit control flow (e.g. `while` loops and `if` statements). The absence of a precise control flow is a particular hindrance as it can lead to poor generalization. For example, whereas natural computer programs are often built with the inductive bias to use control statements ensuring correct execution on inputs of arbitrary size, models like Recurrent Neural Networks can struggle to generalize from short training instances to instances of arbitrary length.

Several models have already been proposed which start to address the functional differences between neural networks and computer programs. These include Recurrent Neural Networks (RNNs) augmented with a stack or queue memory (Giles et al., 1989; Joulin and Mikolov, 2015; Grefenstette et al., 2015), Neural Turing Machines (Graves et al., 2014), Memory Networks (Weston et al., 2014), Neural GPUs (Kaiser and Sutskever, 2016), Neural Programmer-Interpreters (Reed and de Freitas, 2016), and Neural Random Access Machines (Kurach et al., 2015). These models combine deep neural networks with external memory, external computational primitives, and/or built-in structure that reflects a desired algorithmic structure in their execution. Furthermore, they have been been shown to be trainable by gradient descent. However, they do not fix all of the absences noted above. First, none of these models produce programs as output. That is, the representation of the learned model is not interpretable source code. Instead, the program is hidden inside "controllers" composed of neural networks that decide which operations to perform, and the learned "program" can only be understood in terms of the executions that it produces on specific inputs. Second, there is still no concept of explicit control flow in these models.

These works raise questions of (a) whether new models can be designed specifically to synthesize interpretable source code that may contain looping and branching structures, and (b) whether searching over program space using techniques developed for training deep neural networks is a useful alternative to the combinatorial search methods used in traditional IPS. In this work, we make several contributions in both of these directions.

To address the first question we develop models inspired by intermediate representations used in compilers like LLVM (Lattner and Adve, 2004) that can be trained by gradient descent. These models address all of the deficiencies highlighted at the beginning of this section: they interact with external storage, handle non-trivial control flow with explicit `if` statements and loops, and, when appropriately discretized, a learned model can be expressed as interpretable source code. We note two concurrent works, Adaptive Neural Compilation (Bunel et al., 2016) and Differentiable Forth (Riedel et al., 2016), which implement similar ideas. Each design choice when creating differentiable representations of source code has an effect on the inductive bias of the model and the difficulty of the resulting optimization problem. Therefore, we seek a way of rapidly experimenting with different formulations to allow us to explore the full space of modelling variations.

To address the second question, concerning the efficacy of gradient descent, we need a way of specifying an IPS problem such that the gradient based approach can be compared to a variety of alternative approaches in a like-for-like manner. These alternative approaches originate from both a rich history of IPS in the programming languages community and a rich literature of techniques for inference in discrete graphical models in the machine learning community. To our knowledge, no such comparison has previously been performed.

These questions demand that we explore both a range of model variants and a range of search techniques on top of these models. Our answer to both of these issues is the same: TerpreT, a new probabilistic programming language for specifying IPS problems. TerpreT provides a means for describing an *execution model* (e.g., a Turing Machine, an assembly language, etc.) by defining a parameterization (a program representation) and an interpreter that maps inputs to outputs using the parametrized program. This TerpreT description is independent of any particular inference algorithm. The IPS task is to infer the execution model parameters (the program) given an execution model and pairs of inputs and outputs. To perform inference, TerpreT is automatically "compiled" into an intermediate representation which can be fed to a particular inference algorithm. Interpretable source code can be obtained directly from the inferred model parameters. The driving design principle for TerpreT is to strike a subtle balance between the breadth of expression needed to precisely capture a range of execution models, and the

Technique name	Family	Optimizer/Solver	Description
FMGD (Forward marginals, gradient descent)	Machine learning	TensorFlow	A gradient descent based approach which generalizes the approach used by Kurach et al. (2015).
(I)LP ((Integer) linear programming)	Machine learning	Gurobi	A novel linear program relaxation approach based on adapting standard linear program relaxations to support Gates (Minka and Winn, 2009).
SMT (Satisfiability modulo theories)	Program synthesis	Z3	Translation of the problem into a first-order logical formula with existential constraints.
Sketch	Program synthesis	Sketch	View the TerpreT model as a partial program (the interpreter) containing holes (the source code) to be inferred according to a specification (the input-output examples).

Table 1: Overview of considered TerpreT back-end inference algorithms.

restriction of expression needed to ensure that automatic compilation to a range of different back-ends is tractable.

TerpreT currently has four back-end inference algorithms, which are listed in Table 1: gradient-descent (thus any TerpreT model can be viewed as a differentiable interpreter), (integer) linear program (LP) relaxations, SMT, and the Sketch program synthesis system (Solar-Lezama, 2008). To allow all of these back-ends to be used regardless of the specified execution model requires some generalizations and extensions of previous work. For the gradient descent case, we generalize the approach taken by Kurach et al. (2015), lifting discrete operations to operate on discrete distributions, which then leads to a differentiable system. For the linear program case, we need to extend the standard LP relaxation for discrete graphical models to support `if` statements. In Section 4.3, we show how to adapt the ideas of *gates* (Minka and Winn, 2009) to the linear program relaxations commonly used in graphical model inference (Schlesinger, 1976; Werner, 2007; Wainwright and Jordan, 2008). This could serve as a starting point for further work on LP-based message passing approaches to IPS (e.g., following Sontag et al. (2008)).

Finally, having built TerpreT, it becomes possible to develop understanding of the strengths and weaknesses of the alternative approaches to inference. To understand the limitations of using gradient descent for IPS problems, we first use TerpreT to define a simple example where gradient descent fails, but which the alternative back-ends solve easily. By studying this example we can better understand the possible failure modes of gradient descent. We prove that there are exponentially many local optima in the example and show empirically that they arise often in practice (although they can be mitigated significantly by using optimization heuristics like adding noise to gradients during training (Neelakantan et al., 2016b)). We then perform a comprehensive empirical study comparing different inference back-ends and program representations. We show that some domains are significantly more difficult for gradient descent than others and show results suggesting that gradient descent performs best when given redundant, overcomplete parameterizations. However, the overwhelming trend in the experiments is that the techniques from the programming languages community outperform the machine learning approaches by a significant margin.

In summary, our main contributions are as follows:

- A novel 'Basic Block' execution model that enables learning programs with complex control flow (branching and loops).
- TerpreT, a probabilistic programming language tailored to IPS, with back-end inference algorithms including techniques based on gradient descent, linear programming, and highly-efficient systems from the programming languages community (SMT and Sketch). TerpreT also allows "program

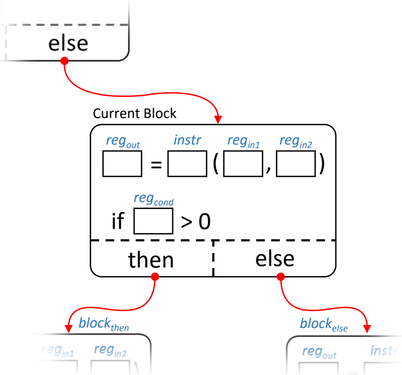


Figure 1: Diagram of the Basic Block program representation. Empty boxes and connections denote per-block unknown parameters to be filled in by the inference algorithm. The choice of which block to go to in the `then` and `else` conditions are also unknown parameters. The domain of unknown parameters is described in the small blue text. An assignment to all unknown parameters yields a program.

sketching", in which a partial solution is provided to the IPS system. For this, some parameters of an execution model can simply be fixed, e.g. to enforce control flow of a specified shape.

- A novel linear program relaxation to handle the `if` statement structure that is common in execution models, and a generalization of the smoothing technique from Kurach et al. (2015) to work on any execution model expressible in TerpreT.
- Analytic and experimental comparisons of different inference techniques for IPS and experimental comparisons of different modelling assumptions.

This report is arranged as follows: We briefly introduce the 'Basic Block' model in Section 2 to discuss what features TerpreT needs to support to allow modeling of rich execution models. In Section 3 we describe the core TerpreT language and illustrate how to use it to explore different modeling assumptions using several example execution models. These include a Turing Machine, Boolean Circuits, a RISC-like assembly language, and our Basic Block model. In Section 4 we describe the compilation of TerpreT models to the four back-end algorithms listed in Table 1. Quantitative experimental results comparing these back-ends on the aforementioned execution models is presented in Section 6. Finally, related work is summarized in Section 7 and we discuss conclusions and future work in Section 8.

2 Motivating Example: Differentiable Control Flow Graphs

As an introductory example, we describe a new execution model that we would like to use for IPS. In this section, we describe the model at a high level. In later sections, we describe how to express the model in TerpreT and how to perform inference.

Control flow graphs (CFGs) (Allen, 1970) are a representation of programs commonly used for static analysis and compiler optimizations. They consist of a set of *basic blocks*, which contain sequences of instructions with no jumps (i.e., straight-line code) followed by a jump or conditional jump instruction to transfer control to another block. CFGs are expressive enough to represent all of the constructs used in modern programming languages like `C++`. Indeed, the intermediate representation of LLVM is based on basic blocks.

Our first model is inspired by CFGs but is limited to use a restricted set of instructions and does not support function calls. We refer to the model as the *Basic Block* model. An illustration of the model appears in Fig. 1. In more detail, we specify a fixed number of blocks B, and we let there be R registers that can take on values $0, \ldots, M-1$. We are given a fixed set of instructions that implement basic

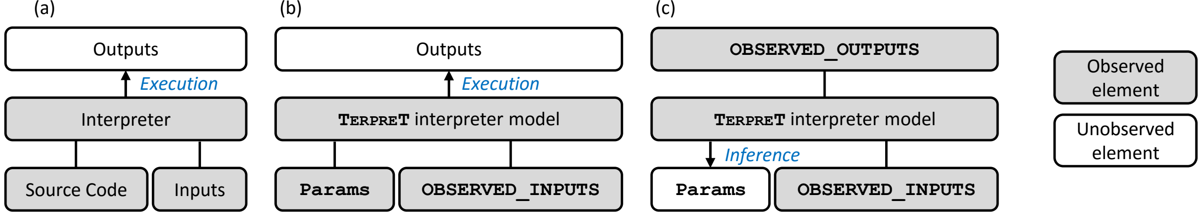


Figure 2: A high level view of the program synthesis task. Forward execution of a traditional program interpreter is shown in (a) for analogy with the forward mode (b) and reverse mode (c) of a TerpreT IPS system.

arithmetic operations, like `ADD`, `INCREMENT`, and `LESS-THAN`. An external memory can be written to and read from using special instructions `READ` and `WRITE`. There is an instruction pointer that keeps track of which block is currently being executed. Each block has a single statement parameterized by two argument registers, the instruction to be executed, and the register in which to store the output. After the statement is executed, a condition is checked, and a branch is taken. The condition is parameterized by a choice of register to check for equality to 0 (C-style interpretation of integers as booleans). Based upon the result, the instruction pointer is updated to be equal to the `then` block or the `else` block. The identities of these blocks are the parameterization of the branch decision.

The model is set up to always start execution in block 0, and a special *end block* is used to denote termination. The program is executed for a fixed maximum number of timesteps T. To represent input-output examples, we can set an initial state of external memory, and assert that particular elements in the final memory should have the desired value upon termination.

The job for TerpreT in this case is to precisely describe the execution model—how statements are executed and the instruction pointer is updated—in a way which can be translated into a fully differentiable interpreter for the Basic Block language or into an intermediate representation for passing to other back-ends. In the next sections, we describe in more detail how TerpreT execution models are specified and how the back-ends work.

3 Front-end: Describing an IPS problem

One of our central aims is to disentangle the description of an execution model from the inference task so that we can perform like-for-like comparisons between different inference approaches to the same IPS task. For reference, the key components for solving an IPS problem are illustrated in Fig. 2. In the forward mode the system is analogous to a traditional interpreter, but in a reverse mode, the system infers a representation of source code given only observed outputs from a set of inputs. Even before devising an inference method, we need both a means of parameterizing the source code of the program, and also a precise description of the interpreter layer's forward transformation. This section describes how these modeling tasks are achieved in TerpreT.

3.1 The TerpreT Probabilistic Programming Language

The full grammar for syntactically correct TerpreT programs is shown in Fig. 3, and we describe the key semantic features of the language in the following sections. For illustration, we use a running example of a simple automaton shown in Fig. 4. In this example the 'source code' is parameterised by a 2×2 boolean array, `ruleTable`, and we take as input the first two values on a binary tape of length T, {`tape[0]`, `tape[1]`}. The forward execution of the interpreter could be described by the following simple `Python` snippet:

```
1 for t in range(1, T - 1):
2         tape[t + 1] = ruleTable[ tape[t - 1], tape [t] ]
```

Const Expr	c	:=	$n \mid v_c \mid f(c_1, \cdots, c_k) \mid c_0 \; op_a \; c_1$
Arith Expr	a	:=	$v \mid c \mid v[a_1, \cdots, a_k] \mid a_0 \; op_a \; a_1 \mid f(a_0, \cdots, a_k)$
Arith Op	op_a	:=	`+` | `-` | `*` | `/` | `%`
Bool Expr	b	:=	$a_0 \; op_c \; a_1$ | `not` b | b_0 `and` b_1 | b_0 `or` b_1
Comp Op	op_c	:=	`==` | `<` | `>` | `<=` | `>=`
Stmt	s	:=	s_0 `;` s_1 | `return` a
		|	a_0`.set_to(`a_1`)` | a`.set_to_constant(`c`)` | a_0`.observe_value(`a_1`)`
		|	`if` b_0`:` s_0 `else:` s_1
		|	`for` v `in range(`c_1`) :` s | `for` v `in range(`c_1, c_2`) :` s
		|	`with` a `as` v `:` s
Decl Stmt	s_d	:=	s_{d_0} `;` s_{d_1} | v_c `=` c | v_c `= #__HYPERPARAM_`v_c`__`
		|	v `= Var(`c`)` | v `= Var(`c`)[`c_1`,` $\cdots$ `,`c_k`]`
		|	v `= Param(`c`)` | v `= Param(`c`)[`c_1`,` $\cdots$ `,`c_k`]`
		|	`@CompileMe([`c_1`,` $\cdots$ `,`c_k`],`c_r`); def` f`(`v_0`,` $\cdots$ `,`v_k`):`s
Input Decl	s_i	:=	`#__IMPORT_OBSERVED_INPUTS__`
Output Decl	s_o	:=	`#__IMPORT_OBSERVED_OUTPUTS__`
Program	p	:=	$s_d; s_i; s; s_o$

Figure 3: The syntax of TerpreT, using natural numbers n, variable names v, constant names v_c and function names f.

Given an observed output, `tape[`$T-1$`]`, inference of a consistent `ruleTable` is very easy in this toy problem, but it is instructive to analyse the TerpreT implementation of this automaton in the following sections. These sections describe variable declaration, control flow, user defined functions and handling of observations in TerpreT.

3.1.1 Declarations and Assignments

We allow declarations to give names to "magic" constants, as in line 1 of Fig. 4. Additionally, we allow the declaration of *parameters* and *variables*, ranging over a finite domain $0, 1, \ldots N-1$ using `Param(`N`)` and `Var(`N`)`, where N has to be a compile-time constant (i.e., a natural number or an expression over constants). Parameters are used to model the source code to be inferred, whereas variables are used to model the computation (i.e., intermediate values). For convenience, (multi-dimensional) arrays of variables can be declared using the syntax `foo = Var(`N`)[`dim_1`,` dim_2`, ...]`, and accessed as `foo[`idx_1`,` idx_2`, ...]`. Similar syntax is available for `Param`s. These arrays can be unrolled during compilation such that unique symbols representing each element are passed to an inference algorithm, i.e., they do not require special support in the inference backend. For this reason, dimensions dim_i and indices idx_i need to be compile-time constants. Example variable declarations can be seen in lines 6 and 11 of Fig. 4.

Assignments to declared variables are not allowed via the usual assignment operator (*var* `=` *expr*) but are instead written as *var*`.set_to(`*expr*`)`, to better distinguish them from assignments to constant variables. Static single assignment (SSA) form is enforced, and it is only legal for a variable to appear multiple times as the target of `set_to` statements if each assignment appears in different cases of a conditional block. Because of the SSA restriction, a variable can only be written once. However, note that programs that perform multiple writes to a given variable can always be translated to their corresponding SSA forms.

```
1  const_T = 5
2
3  ######################################################
4  #  Source code parametrisation                       #
5  ######################################################
6  ruleTable = Param(2)[2, 2]
7
8  ######################################################
9  #  Interpreter model                                 #
10 ######################################################
11 tape = Var(2)[const_T]
12
13 #__IMPORT_OBSERVED_INPUTS__
14 for t in range(1, const_T − 1):
15     with tape[t] as x1:
16         with tape[t − 1] as x0:
17             tape[t + 1].set_to(ruleTable[x0, x1])
18 #__IMPORT_OBSERVED_OUTPUTS__
```

Figure 4: Illustrative example of a TerpreT script and corresponding factor graph which describe a toy automaton that updates a binary `tape` according to the previous two entries and a `rule` (refer to Fig. 7 for definition of graphical symbols).

3.1.2 Control flow

TerpreT supports standard control-flow structures such as `if-else` (where `elif` is the usual shorthand for `else if`) and `for`. In addition, TerpreT uses a unique `with` structure. The need for the latter is induced by our requirement to only use compile-time constants for accessing arrays. Thus, to set the 2nd element of `tape` in our toy example (i.e., the first step of the computation), we need code like the following to access the values of the first two values on the tape:

```
1  if tape[1] == 0:
2        if tape[0] == 0:
3               tape[2].set_to(ruleTable[0,0])
4        elif tape[0] == 1:
5               tape[2].set_to(ruleTable[1,0])
6  elif tape[1] == 1:
7        if tape[0] == 0:
8               tape[2].set_to(ruleTable[0,1])
9        elif tape[0] == 1:
10              tape[2].set_to(ruleTable[1,1])
```

Intuitively, this snippet simply performs case analyses over all possible values of `tape[1]` and `tape[0]`. To simplify this pattern, we introduce the `with` *var* `as` *id*`:` *stmt* control-flow structure, which allows to automate this unrolling, or avoid it for back-ends that do not require it (such as Sketch). To this end, all possible possible values $0, \ldots, N-1$ of *var* (known from its declaration) are determined, and the `with`-statement is transformed into `if` *id* `== 0 then:` $stmt[var/0]$`; elif` *id* `== 1 then:` $stmt[var/1]$`; ...elif` *id* `==` n `then:` $stmt[var/(N-1)]$`;`, where $stmt[var/n]$ denotes the statement *stmt* in which

```python
const_T = 5

@CompileMe([2, 2], 3)
def add(a, b):
    s = a + b
    return s

##################################################
#  Source code parametrisation                   #
##################################################
ruleTable = Param(2)[3]

##################################################
#  Interpreter model                             #
##################################################
tape = Var(2)[const_T]
tmpSum = Var(3)[const_T - 1]

#__IMPORT_OBSERVED_INPUTS__
for t in range(1, const_T - 1):
    tmpSum[t].set_to(add(tape[t - 1], tape[t]))
    with tmpSum[t] as s:
        tape[t + 1].set_to(ruleTable[s])
#__IMPORT_OBSERVED_OUTPUTS__
```

Figure 5: An example TERPRET script and the corresponding factor graph which describe a toy automaton that updates a binary `tape` according to the previous two entries and a `rule`.

all occurrences of the variable *var* have been replaced by n. Thus, the snippet from above can be written as follows.

```python
with tape[1] as x1:
       with tape[0] as x0:
              tape[2].set_to(ruleTable[x0,x1])
```

In TERPRET, `for` loops may only take the shape `for` *id* `in range(`c_1`,` c_2`):` *stmt*, where c_1 and c_2 are compile-time constants. Similar to the `with` statement, we can unroll such loops explicitly during compilation, and thus if the values of c_1 and c_2 are n_1 and n_2, we generate $stmt[id/n_1]$; $stmt[id/(n_1+1)]$; . . . ; $stmt[id/(n_2-1)]$. Using the `with` and `for` statements, we can thus describe the evaluation of our example automaton for `const_T` timesteps as shown in lines 14-17 of Fig. 4.

3.1.3 Operations

TERPRET supports user-defined functions to facilitate modelling interpreters supporting non-trivial instruction sets. For example, `bar(`arg_1`,...,`arg_M`)` will apply the function `bar` to the arguments arg_1,...,arg_M. The function `bar`: $\mathbb{Z}^M \to \mathbb{Z}$ can be defined as a standard `Python` function with the additional decoration `@CompileMe(`*in_domains*`,` *out_domain*`)`, specifying the domains of the input and output variables.

To illustrate this feature, Fig. 5 shows variation of the running example where the automaton updates the `tape` according to a `ruleTable` which depends only on the *sum* of the preceding two entries. This is

implemented using the function `add` in lines 3-6. Note that we use standard `Python` to define this function and leave it up to the compiler to present the function appropriately to the inference algorithm.

3.1.4 Modelling Inputs and Outputs

Using statements from the preceding sections, an execution model can be fully specified, and we now connect this model to input/output observations to drive the program induction. To this end, we use the statements `set_to_constant` (resp. `observe_value`) to model program input (resp. program output). Thus, a single input-output observation for the running example could be written in TerpreT as follows.

```
1  # input
2  tape[0].set_to_constant(1)
3  tape[1].set_to_constant(0)
4
5  # output
6  tape[const_T - 1].observe_value(1)
```

To keep the execution model and the observations separate, we store the observation snippets in a separate file and use preprocessor directives `#__IMPORT_OBSERVED_*__` to pull in the appropriate snippets before compilation (see lines 13 and 18 of Fig. 4). We also allow any constant literals to be stored separately from the TerpreT execution model, and we import these values using preprocessor directives of the form v_c `= #__HYPERPARAM_`v_c`__`.

In general, we want to infer programs from $n_{\text{obs}} > 1$ input-output examples. The simplest implementation achieves this by augmenting each `Var` declaration with an additional array dimension of size n_{obs} and wrapping the execution model in a `for` loop over the examples. Examples of this are the outermost loops in the models in Appendix B.

3.2 Example Execution Models

To illustrate the versatility of TerpreT, we use it to describe four example execution models. Broadly speaking, the examples progress from more abstract execution models towards models which closely resemble assembly languages for RISC machines.

In each case, we present the basic model and fill in three representative synthesis tasks in Table 2 to investigate. In addition, we provide the metrics for the "difficulty" of each task calculated from the minimal computational resources required in a solution. Since the difficulty of a synthesis problem generally depends on the chosen inference algorithm these metrics are primarily intended to give a sense of the scale of the problem. The first difficulty metric, D, is the number of structurally distinct (but not necessarily functionally distinct) programs which would have to be enumerated in a worst-case brute-force search, and the second metric, T, is the unrolled length of all steps in the synthesized program.

3.2.1 Automaton: Turing Machine

A Turing machine consists of an infinite tape of memory cells which each contain one of S symbols, and a head which moves over the tape in one of $H + 1$ states (one state is the special `halt` case). At each execution step, while the head is in an unhalted state h_t, it reads the symbol s_t at its current position, x_t, on the tape, then it writes the symbol `newValue[`s_t`,`h_t`]` to position x_t, moves in the direction specified by `direction[`s_t`,`h_t`]` (one cell left or right or no move) and adopts a new state `newState[`s_t`,`h_t`]`. The source code for the Turing machine is the entries of the control tables `newValue`, `direction` and `newState`, which can be in any of $D = [3S(H+1)]^{SH}$ configurations.

We modify the canonical Turing machine to have a circular tape of finite length, L, as described in the TerpreT model in Appendix B.1. For each of our examples, we represent the symbols on the tape as $\{$`0`$,$ `1`$,$ `blank`$\}$.

TURING MACHINE	H	L	$\log_{10} D$	T	Description
Invert	1	5	4	6	Move from left to right along the tape and invert all the binary symbols, halting at the first `blank` cell.
Prepend zero	2	5	9	6	Insert a "`0`" symbol at the start of the tape and shift all other symbols rightwards one cell. Halt at the first `blank` cell.
Binary decrement	2	5	9	9	Given a tape containing a binary encoded number $b_{\text{in}} > 0$ and all other cells `blank`, return a tape containing a binary encoding of $b_{\text{in}} - 1$ and all other cells `blank`.

BOOLEAN CIRCUITS	R	$\log_{10} D$	T	Description
2-bit controlled shift register	4	10	4	Given input registers (r_1, r_2, r_3), output (r_1, r_2, r_3) if $r_1 == 0$ otherwise output (r_1, r_3, r_2) (i.e. r_1 is a control bit stating whether r_2 and r_3 should be swapped).
full adder	4	13	5	Given input registers $(c_{\text{in}}, a_1, b_1)$ representing a carry bit and two argument bits, output a sum bit and carry bit (s, c_{out}), where $s + 2c_{\text{out}} = c_{\text{in}} + a_1 + b_1$.
2-bit adder	5	22	8	Perform binary addition on two-bit numbers: given registers (a_1, a_2, b_1, b_2), output $(s_1, s_2, c_{\text{out}})$ where $s_1 + 2s_2 + 4c_{\text{out}} = a_1 + b_1 + 2(a_2 + b_2)$.

BASIC BLOCK	M	R	B	$\log_{10} D$	T	Description
Access	5	2	5	14	5	Access the kth element of a contiguous array. Given an initial heap $\texttt{heap}_0\texttt{[0]} = k$, $\texttt{heap}_0\texttt{[}1 : J+1\texttt{]} = \texttt{A[:]}$ and $\texttt{heap}_0\texttt{[}J+1\texttt{]} = 0$, where $\texttt{A[}j\texttt{]} \in \{1, ..., M-1\}$ for $0 \leq j < J$, $J+1 < M$ and $0 \leq k < J$, terminate with $\texttt{heap[0]} = \texttt{A[}k\texttt{]}$.
Decrement	5	2	5	19	18	Decrement all elements in a contiguous array. Given an initial heap $\texttt{heap}_0\texttt{[k]} \in \{2, ..., M-1\}$ for $0 \leq k < K < M$ and $\texttt{heap}_0\texttt{[}K\texttt{]} = 0$, terminate with $\texttt{heap[k]} = \texttt{heap}_0\texttt{[k]} - 1$.
List-K	8	2	8	33	11	Access the kth element of a linked list. The initial heap is $\texttt{heap}_0\texttt{[0]} = k$, $\texttt{heap}_0\texttt{[1]} = p$, and $\texttt{heap}_0\texttt{[2:M]} = \texttt{linkList}$, where `linkList` is a linked list represented in the heap as adjacent [next pointer, value] pairs in random order, and p is a pointer to the head element of `linkList`. Terminate with $\texttt{heap[0]} = \texttt{linkList[}k\texttt{].value}$.

ASSEMBLY	M	R	B	$\log_{10} D$	T	Description
Access	5	2	5	13	5	As above.
Decrement	5	2	7	20	27	
List-K	8	2	10	29	16	

Table 2: Overview of benchmark problems, grouped by execution model. For each benchmark we manually find the minimal feasible resources (e.g. minimum number of registers, Basic Blocks, timesteps etc.). These are noted in this table and we try to automatically solve the synthesis task with these minimal settings.

3.2.2 Straight-line programs: Boolean Circuits

As a more complex model, we now consider a simple machine capable of performing a sequence of logic operations (`AND`, `OR`, `XOR`, `NOT`, `COPY`) on a set of registers holding boolean values. Each operation takes two registers as input (the second register is ignored in the `NOT` and `COPY` operation), and outputs to one register, reminiscent of standard three-address code assembly languages. To embed this example in a real-world application, analogies linking the instruction set to electronic logic gates and linking the registers to electronic wires can be drawn. This analogy highlights one benefit of interpretability in our model: the synthesized program describes a digital circuit which could easily be translated to real hardware (see e.g. Fig. 20). The TerpreT implementation of this execution model is shown in Appendix B.2.

There are $D = H^T R^{3T}$ possible programs (circuits) for a model consisting of T sequential instructions (logic gates) each chosen from the set of $H = 5$ possible operations acting on R registers (wires).

3.2.3 Loopy programs 1: Basic block model

To build loopy execution models, we take inspiration from compiler intermediate languages (e.g., LLVM Intermediate Representation), modeling full programs as graphs of "basic blocks". Such programs operate on a fixed number of registers, and a byte-addressable heap store accessible through special instructions, `READ` and `WRITE`. Each block has an instructions of the form $\mathit{reg}_{\text{out}}$ `=` $\mathit{instr}\ \mathit{reg}_{\text{in1}}\ \mathit{reg}_{\text{in2}}$, followed by a branch decision `if` $\mathit{reg}_{\text{cond}} > 0$ `goto` $\mathit{block}_{\text{then}}$ `else goto` $\mathit{block}_{\text{else}}$ (see Fig. 1, and the TerpreT model in Appendix B.3). This representation can easily be transformed back and forth to higher-level program source code (by standard compilation/decompilation techniques) as well as into executable machine code.

We use an instruction set containing $H = 9$ instructions: `ZERO`, `INC`, `DEC`, `ADD`, `SUB`, `LESSTHAN`, `READ`, `WRITE` and `NOOP`. This gives $D = [HR^4(B+1)^2]^B$ possible programs for a system with R registers and $(B+1)$ basic blocks (including a special stop block which executes `NOOP` and redirects to itself). We consider the case where registers and heap memory cells all store a single data type - integers in the range $0, .., M-1$, where M is the number of memory cells on the heap. This single data type allows both intermediate values and pointers into the heap to be represented in the registers and heap cells.

While this model focuses on interpretability, it also builds on an observation from the results of Kurach et al. (2015). In NRAMs, a RNN-based controller chooses a short sequence of instructions to execute next based on observations of the current program state. However, the empirical evaluation reports that correctly trained models usually picked one sequence of instructions in the first step, and then repeated another sequence over and over until the program terminates. Intuitively, this corresponds to a loop initialization followed by repeated execution of a loop body, something which can naturally be expressed in the Basic Block model.

3.2.4 Loopy programs 2: Assembly model

In the basic block model every expression is followed by a conditional branch, giving the model great freedom to represent rich control flow graphs. However, useful programs often execute a sequence of *several* expressions between each branch. Therefore, it may be beneficial to bias the model to create chains of sequentially ordered basic blocks with only occasional branching where necessary. This is achieved by replacing the basic blocks with objects which more closely resemble lines of assembly code. The instruction set is augmented with the jump statements jump-if-zero (`JZ`$(\mathit{reg}_{\text{in1}}) : \mathit{branchAddr}$), and jump-if-not-zero (`JNZ`$(\mathit{reg}_{\text{in1}}) : \mathit{branchAddr}$), the operation of which are shown in Fig. 6 (and in the TerpreT code in Appendix B.4). Each line of code acts like a conditional branch only if the assigned $\mathit{instr} \in \{\mathtt{JZ}, \mathtt{JNZ}\}$ otherwise it acts like a single expression which executes and passes control to the next line of code. This assembly model can express the same set of programs as the basic block model, and serves as an example of how the design of the model affects the success of program inference.

In addition, we remove `NOOP` from the instruction set (which can be achieved by a jump operation pointing to the next line) leaving $H = 10$ instructions, and we always include a special stop line as the $(B+1)^{\text{th}}$ line of the program. The total size of the search space is then $D = [HR^3(B+1)]^B$.

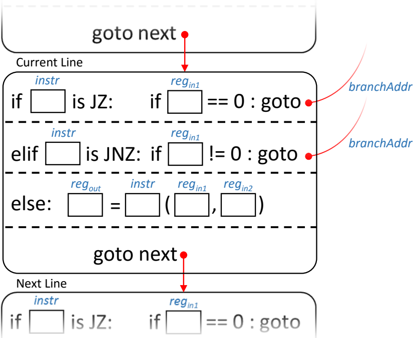


Figure 6: Diagram of the assembly program representation. We present the model using the same graphical style as the Basic Block model in Fig. 1.

4 Back-ends: Solving the IPS problem

TerpreT is designed to be compiled to a variety of intermediate representations for handing to different inference algorithms. This section outlines the compilation steps for each of the back-end algorithms listed in Table 1.

For each back-end we present the compiler transfomation of the TerpreT primitives listed in Fig. 7. For some back-ends, we find it useful to present these transformations via an intermediate *graphical* representation resembling a factor graph, or more specifically, a gated factor graph (Minka and Winn, 2009), which visualises the TerpreT program. Below we describe gated factor graphs and provide the mapping from TerpreT syntax to primitives in these models. Then in Section 4.2 - 4.5 we show how to compile TerpreT for each back-end solver.

4.1 TerpreT for Gated Factor Graph Description

A factor graph is a means of representing the factorization of a complex function or probability distribution into a composition of simpler functions or distributions. In these graphs, inputs, outputs and intermediate results are stored in *variable* nodes linked by *factor* nodes describing the functional relationships between variables. A TerpreT model defines the structure of a factor graph, and an inference algorithm is used to populate the variable nodes with values consistent with observations.

Particular care is needed to describe factor graphs containing conditional branches since the value of a variable X_i in conditions of the form X_i `==` c is not known until inference is complete. This means that we must explore all branches during inference. *Gated* factor graphs can be used to handle these `if` statements, and we introduce additional terminology to describe these gated models below. Throughout the next sections we refer to the TerpreT snippet shown in Fig. 8 for illustration.

Local unary marginal. We restrict attention to the case where each variable X_i is discrete, with finite domain $\mathcal{X}_i = \{0, \dots, N_i - 1\}$. For each variable we instantiate a *local unary marginal* $\mu_i(x)$ defined on the support $x \in \mathcal{X}_i$. In an *integral* configuration, we demand that $\mu_i(x)$ is only non-zero at a particular value x_i^*, allowing us to interpret $X_i = x_i^*$. Some inference techniques relax this constraint and consider a continuous model $\mu_i(x) \in \mathbb{R}\ \forall x \in \mathcal{X}_i$. In these relaxed models, we apply continuous optimization schemes which, if successful, will converge on an interpretable integral solution.

Gates. Following Minka and Winn (2009), we refer to `if` statements as *gates*. More precisely, an `if` statement consists of a condition (an expression that evaluates to a boolean) and a body (a set of assignments or factors). We will refer to the condition as the *gate condition* and the body as the *gate body*. In this work, we restrict attention to cases where all gate conditions are of the form $X_i == ConstExpr$. In future work we could relax this restriction.

Graph element	TERPRET representation	Graphical representation
Random variable (intermediate)	`Xi = Var(`N`)`	
Random variable (inference target)	`Xi = Param(`N`)`	
Observed variable (input)	`Xi.set_to_constant(x*)`	
Observed variable (output)	`Xi.observe_value(x*)`	
Factor (copy)	`X0.set_to(X1)`	
Factor (general)	`X0.set_to(`f`(X1,X2,...))`	

Gates	`if C == 0:`
	   $stmt_0$
	`elif C == 1:`
	   $stmt_1$
	`elif C == 2:`
	   $stmt_2$
	`...`
	`elif C == n:`
	   $stmt_n$	

Figure 7: The main TERPRET primitives and their corresponding graphical representation.

In the example in Fig. 8, there is a nested gate structure. At the outer-most level, there are two gates with gate conditions `(X0 == 0)` (lines 16-20) and `(X0 == 1)` (lines 21-22). Inside the `(X0 == 0)` gate, there are two nested gates (corresponding to `(X1 == 0)` and `(X1 == 1)`).

Path conditions. Each gate A has a *path condition* ψ_A, which is a list of variables and values they need to take on in order for the gate body to be executed. For example, in Fig. 8, the path condition for the innermost gate body on lines 19-20 is $(X_0 = 0, X_1 = 1)$, where commas denote conjunction. We will use the convention that the condition in the deepest gate's `if` statement is the last entry of the path condition. Gates belong to a tree structure, and if gate B with gate condition ϕ_B is nested inside gate A with path condition ψ_A, then we say that A is a parent of B, and the path condition for B is $\psi_B = (\psi_A, \phi_B)$. We can equally speak of the path condition ψ_j of a factor j, which is the path condition of the most deeply nested gate that the factor is contained in.

Active variables. Define a variable X to be *active* in a gate A if both of the following hold:

- X is used in A or one of its descendants, and
- X is declared in A or one of its ancestors.

That is, X is active in A iff A is on the path between X's declaration and one of its uses.

For each gate A in which a variable is active, we instantiate a separate local marginal annotated with the path condition of A (ψ_A). For example, inside the gate corresponding to `(X0 == 0)` in Fig. 8, the

```
1
2   # X4 =  0        if X0 == 0 and X1 == 0
3   #      | X2 + 1   if X0 == 0 and X1 == 1
4   #      | 2*X2     if X0 == 1
5   #
6   # Observe X4 = 5; infer X0, X1, X2
7
8   @CompileMe([2, 10], 10)
9   def Plus(a, b): return (a + b) % 10
10  @CompileMe([10], 10)
11  def MultiplyByTwo(a): return (2 * a) % 10
12
13  X0 = Param(2); X1 = Param(2); X2 = Param(10)
14  X3 = Var(10);  X4 = Var(10)
15
16  if X0 == 0:
17    if X1 == 0:
18      X3.set_to(0); X4.set_to(X3)
19    elif X1 == 1:
20      X3.set_to(Plus(X1, X2)); X4.set_to(X3)
21  elif X0 == 1:
22    X4.set_to(MultiplyByTwo(X2))
23
24  X4.observe_value(5)
```

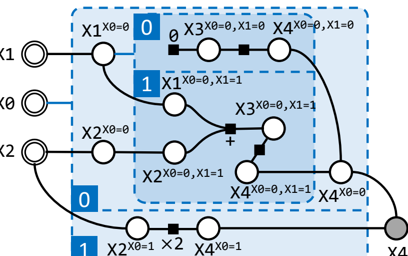


Figure 8: Interpreting TerpreT as a gated factor graph description. We model the inference task shown in lines 1-6 using TerpreT and provide the corresponding gated factor graph using symbols from Fig. 7. The solution to this inference task is `X0 = 0`, `X1 = 1`, and `X2 = 4`).

local marginal for X_i is $\mu_i^{X_0=0}(x)$.[1] In the global scope we drop the superscript annotation and just use $\mu_i(x)$. We can refer to parent-child relationships between different local marginals of the same variable; the parent (child) of a local marginal $\mu_i^{\psi_A}(\cdot)$ is the local marginal for X_i in the parent (child) gate of A.

Gate marginals. Let the *gate marginal* of a gate A be the marginal of the gate's condition in the parent gate of A. In Fig. 8, the first outer gate's gate marginal is $\mu_0(0)$, and the second outer gate's is $\mu_0(1)$. In the inner gate, the gate marginal for the (`X1 == 0`) gate is $\mu_1^{X_0=0}(0)$.

4.2 Forward Marginals Gradient Descent (FMGD) Back-end

The factor graphs discussed above are easily converted into computation graphs representing the execution of an interpreter by the following operations.

- Annotate the factor graph edges with the direction of traversal during forwards execution of the TerpreT program.
- Associate executable functions f_i with factor i operating on scope $S_i = \{\boldsymbol{X}, Y\}$. The function transforms the incoming variables $\boldsymbol{X}$ to the outgoing variable, $Y = f_i(\boldsymbol{X})$.

[1] Strictly speaking, this notation does not handle the case where there are multiple gates with identical path conditions; for clearness of notation, assume that all gate path conditions are unique. However, the implementation handles repeated path conditions (by identifying local marginals according to a unique gate id).

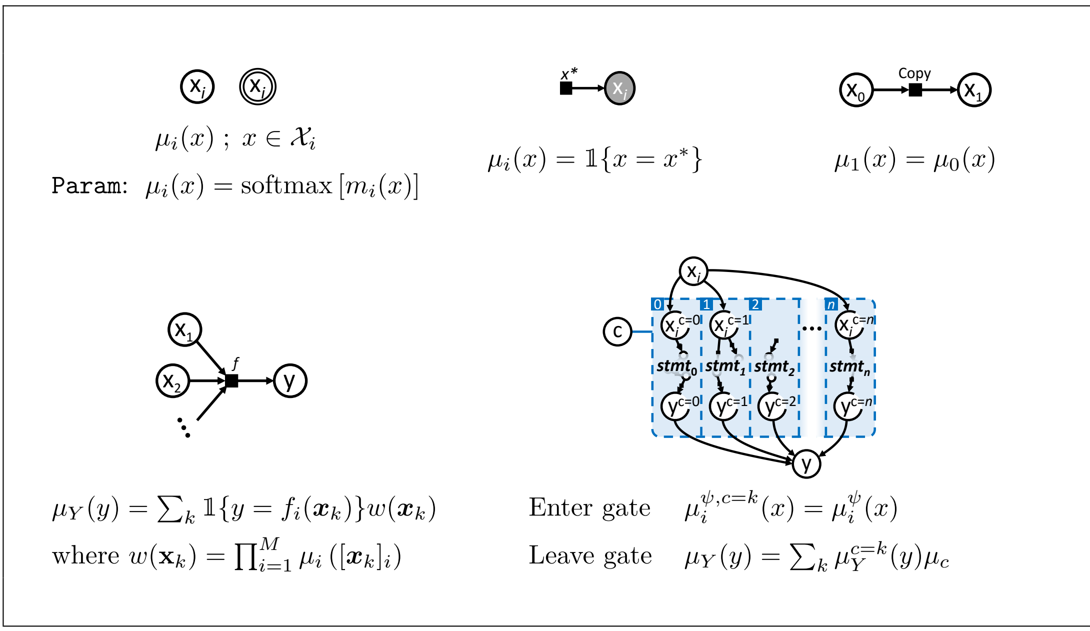


Figure 9: Summary of the forward execution of graphical primitives in the FMGD algorithm. See text for definition of symbols

In the FMGD approach, we initialize the source nodes of this directed graph by instantiating independent random variables $X_p \sim \mu_p$ at each `Param` node, and variables $X_i \sim \text{onehot}(x_i^*)$ at nodes associated with input observations of the form `Xi.set_to_constant`(x_i^*). Here $\text{onehot}(x_i^*)$ is a distribution over $\mathcal{X}_i$ with unit mass at x_i^*. We then propagate these *distributions* through the computation graph using the FMGD approximation, described below, to obtain distributions μ_o at the output nodes associated with an `observe_value`(x_o^*) statement. This fuzzy system of distributions is fully differentiable. Therefore inference becomes an optimization task to maximize the weight $\mu_o(x_o^*)$ assigned to the observations by updating the parameter distributions $\{\mu_p\}$ by gradient descent.

The key FMGD approximation arises whenever a derived variable, Y depends on several immediate input variables, $\boldsymbol{X}$. In an ungated graph, this occurs at factor nodes where $Y = f_i(\boldsymbol{X})$. FMGD operates under the under the approximation that *all* $\boldsymbol{X}$ *are independent*. In this case, we imagine a local joint distribution $\mu_{Y\boldsymbol{X}}$ constructed according to the definition of f_i and the independent unary marginal distributions for $\boldsymbol{X}$. From this distribution we marginalize out all of the input variables to obtain the unary marginal μ_Y (see Section 4.2.1). Only μ_Y is propagated forward out of the factor node and correlations between Y and $\boldsymbol{X}$ (only captured by the full local joint distribution) are lost. In the next section we explicitly define these operations and extend the technique to allow for gates in the factor graph.

It is worth noting that there is a spectrum of approximations in which we form joint distributions for subgraphs of size ranging from single nodes (FMGD) to the full computation graph (enumerative search) with only independent marginal distributions propagated between subgraphs. Moving on this spectrum trades computational efficiency for accuracy as more correlations can be captured in larger subgraphs. An exploration of this spectrum could be a basis for future work.

4.2.1 Forward Marginals...

Fig. 9 illustrates the transformation of each graphical primitive to allow a differentiable forward propagation of marginals through a factor graph. Below we describe more details of factor and gate primitives in this algorithm.

Factors. The scope S of a factor function f contains the M immediate input variables X_i and the immediate output, Y. In this restricted environment, we enumerate the possible outputs Y from all $\prod_{i=1}^{M} |\mathcal{X}_i|$ possible input configurations $\boldsymbol{x}_k$ of the form $[\boldsymbol{x}_k]_i \in \mathcal{X}_i$ for $i \in \{1, ..., M\}$. We then marginalise over the configuration index, k, using weightings $\mu_i([\boldsymbol{x}_k]_i)$ to produce μ_Y as follows:

$$\mu_Y(y) = \sum_k \mathbb{1}\{y = f(\boldsymbol{x}_k)\} w(\boldsymbol{x}_k), \tag{1}$$

where $\mathbb{1}$ is an indicator function and the weighting function w is:

$$w(\boldsymbol{x}_k) = \prod_{i=1}^{M} \mu_i \left([\boldsymbol{x}_k]_i\right). \tag{2}$$

Note that (1) and (2) can be implemented efficiently as a series of tensor contractions of the $M_i + 1$ dimensional binary tensor $I_{y\boldsymbol{x}} = \mathbb{1}\{y = f(\boldsymbol{x})\}$ with the M_i vectors $[\boldsymbol{\mu}_i]_x = \mu_i(x)$.

Gates. We can include gates in the FMGD formulation as follows. Let $B_1, \ldots, B_k$ be the set of child gates of A which are controlled by gate marginal $\mu_B^{\psi_A}(i)$; $i \in \{1, ..., k\}$. Inside gate B_i, there is a subgraph G_i described by TerpreT code T_i which references a set of active variables $\mathcal{B}_i$. We divide $\mathcal{B}_i$ into $\mathcal{L}_i$ containing variables which are written-to during execution of G_i (i.e. appear on the left hand side of expressions in T_i), and $\mathcal{R}_i$ containing variables which are not written-to (i.e. appear only on the right hand side of expressions in T_i). In addition, we use $\mathcal{A}$ to refer to the active variables in A, and $\mathcal{B}^+ \subset \mathcal{A}$ to be variables used in the graph downstream of gates $B_1, \ldots, B_k$ on paths which terminate at observed variables.

On entering gate B_i, we import references to variables in the parent scope, A, for all $X \in \mathcal{R}_i \cap \mathcal{A}$:

$$\mu_X^{\psi_{B_i}} = \mu_X^{\psi_A}. \tag{3}$$

We then run G_i, to produce variables $\mathcal{L}_i$. Finally, when leaving a gate, we marginalise using the gate marginal to set variables $Y \in \mathcal{B}^+$:

$$\mu_Y^{\psi_A}(y) = \sum_{i=1}^{k} \mu_Y^{\psi_{B_i}}(y) \mu_B^{\psi_A}(i). \tag{4}$$

Restrictions on factor functions. The description above is valid for any $f : \times_{i=1}^{M} \mathcal{X}_i \to \mathcal{Y}$, subject to the condition that $\mathcal{Y} \subseteq \mathcal{X}_{\text{out}}$, where $\mathcal{X}_{\text{out}}$ is the domain of the variable which is used to store the output of f. One scenario where this condition could be violated is illustrated below:

```
1  @CompileMe([4],2)
2  def largeTest(x): return 1 if x >= 2 else 0
3  @CompileMe([4],4)
4  def makeSmall(x): return x - 2
5
6  X = Param(4) ; out = Var(4) ; isLarge = Var(2)
7
8  isLarge.set_to(largeTest(X))
9  if isLarge == 0:
10     out.set_to(X)
11 elif isLarge == 1:
12     out.set_to(makeSmall(X))
```

The function `makeSmall` has a range $\mathcal{Y} = \{-2, ..., 1\}$ which contains elements outside $\mathcal{X}_{\text{out}} = \{0, ..., 3\}$. However, deterministic execution of this program does not encounter any error because the path condition `isLarge == 1` guarantees that the invalid cases $\mathcal{Y} \setminus \mathcal{X}_{\text{out}}$ would never be reached. In general, it only makes

sense to violate $\mathcal{Y} \subseteq \mathcal{X}_{\text{out}}$ if we are inside a gate where the path condition ensures that the input values lie in a restricted domain $\tilde{\mathcal{X}}_i \subseteq \mathcal{X}_i$ such that $f : \times_{i=1}^{M} \tilde{\mathcal{X}}_i \rightarrow \mathcal{X}_{\text{out}}$. In this case a we can simply enforce the normalisation of μ_{out} to account for any leaked weight on values $\mathcal{Y} \setminus \mathcal{X}_{\text{out}}$.

$$\mu_Y(y) = \frac{1}{Z} \sum_k \mathbb{1}\{y = f(\mathbf{x}_k)\} w(\mathbf{x}_k), \qquad \text{where} \qquad Z = \sum_{k, y \in \mathcal{X}_{\text{out}}} \mathbb{1}\{y = f(\mathbf{x}_k)\} w(\mathbf{x}_k). \tag{5}$$

With this additional caveat, there are no further constraints on factor functions $f : \mathbb{Z}^M \rightarrow \mathbb{Z}$.

4.2.2 ... Gradient Descent

Given a random initialization of marginals for the the `Param` variables $X_p \in \mathcal{P}$, we use the techniques above to propagate marginals forwards through the TerpreT model to reach all variables, $X_o \in \mathcal{O}$, associated with an `observe_value`(x_o^*) statement. Then we use a cross entropy loss, L, to compare the computed marginal to the observed value.

$$L = -\sum_{X_o \in \mathcal{O}} \log \left[\mu_o(x_o^*)\right]. \tag{6}$$

L reaches its lower bound $L = 0$ if each of the marginals $\mu_p(x)$ representing the `Params` put unit weight on a single value $\mu_p(x_p^*) = 1$ such that the assignments $\{X_p = x_p^*\}$ describe a valid program which explains the observations. The synthesis task is therefore an optimisation problem to minimise L, which we try to solve using backpropagation and gradient descent to reach a zero loss solution.

To preserve the normalisation of the marginals during this optimisation, rather than updating $\mu_p(x)$ directly, we update the log parameters $m_p(x) = [\boldsymbol{m}_p]_x$ defined by $\mu_p(x) = \text{softmax}\left[m_p(x)\right]$. These are initialized according to

$$\exp(\boldsymbol{m}_p) \sim \text{Dirichlet}(\boldsymbol{\alpha}), \tag{7}$$

where $\boldsymbol{\alpha}$ are hyperparameters.

4.2.3 Optimization Heuristics

Using gradient information to search over program space is only guaranteed to succeed if all points with zero gradient correspond to valid programs which explain the observations. Since many different programs can be consistent with the observations, there can be many global optima ($L = 0$) points in the FMGD loss landscape. However, the FMGD approximation can also lead to *local* optima which, if encountered, stall the optimization at an uninterpretable point where $\mu_p(x_p^*)$ assigns weight to several distinct parameter settings. For this reason, we try several different random initializations of $m_p(x)$ and record the fraction of initializations which converge at a global optimum. Specifically, we try two approaches for learning using this model:

- **Vanilla FMGD.** Run the algorithm as presented above, with $\alpha_i = 1$ and using the RMSProp (Tieleman and Hinton, 2012) gradient descent optimization algorithm.

- **Optimized FMGD.** Add the heuristics below, which are inspired by Kurach et al. (2015) and designed to avoid getting stuck in local minima, and optimize the hyperparameters for these heuristics by random search. We also include the initialization scale $\alpha_i = \alpha$ and the gradient descent optimization algorithm in the random search (see Section 5.2 for more details). By setting $\alpha = 1$, parameters are initialized uniformly on the simplex. By setting α smaller, we get peakier distributions, and by setting α larger we get more uniform distributions.

Gradient clipping. The FMGD neural network depth grows linearly with the number of time steps. We mitigate the "exploding gradient" problem (Bengio et al., 1994) by globally rescaling the whole gradient vector so that its $L2$ norm is not bigger than some hyperparameter value C.

Noise. We added random Gaussian noise to the computed gradients after the backpropagation step. Following Neelakantan et al. (2016b), we decay the variance of this noise during the training according to the following schedule:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma} \tag{8}$$

where the values of η and γ are hyperparameters and t is the epoch counter.

Entropy. Ideally, the algorithm would explore the loss surface to find a global minimum rather than fixing on some particular configuration early in the training process, causing the network to get stuck in a local minimum from which it's unlikely to leave. To bias the network away from committing to any particular solution during early iterations, we add an *entropy bonus* to the loss function. Specifically, for each softmax distribution in the network, we subtract the entropy scaled by a coefficient ρ, which is a hyperparameter. The coefficient is exponentially decayed with rate r, which is another hyperparameter.

Limiting the values of logarithms. FMGD uses logarithms in computing both the cost function as well as the entropy. Since the inputs to these logarithms can be very small, this can lead to very big values for the cost function and floating-point arithmetic overflows. We avoid this problem by replacing $\log(x)$ with $\log(\max[x, \epsilon])$ wherever a logarithm is computed, for some small value of ϵ.

Kurach et al. (2015) considered two additional tricks which we did not implement generally.

Enforcing Distribution Constraints. Because of the depth of the networks, propagation of numerical errors can result in $\sum_x \mu_i(x) \neq 1$. Kurach et al. (2015) solve this by adding rescaling operations to ensure normalization. We find that we can avoid this problem by using 64-bit floating-point precision.

Curriculum learning. Kurach et al. (2015) used a curriculum learning scheme which involved first training on small instances of a given problem, and only moving to train on larger instances once the error rate had reduced below a certain value. Our benchmarks contain a small number of short examples (e.g., 5-10 examples acting on memory arrays of up to 8 elements), so there is less room for curriculum learning to be helpful. We manually experimented with hand-crafted curricula for two hard problems (shift and adder), but it did not lead to improvements.

To explore the hyperparameters for these optimization heuristics we ran preliminary experiments to manually chose a distribution over hyperparameter space for use in random search over hyperparameters. The aim was to find a distribution that is broad enough to not disallow reasonable settings of hyperparameters while also being narrow enough so that runs of random search were not wasted on parameter settings that would never lead to convergence. This distribution over hyperparameters was then fixed for all random search experiments.

4.3 (Integer) Linear Program Back-end

We now turn attention to the first alternative back-end to be compared with the FMGD. Casting the TerpreT program as a factor graph allows us to build upon standard practice in constructing LP relaxations for solving maximum a posteriori (MAP) inference problems in discrete graphical models (Schlesinger, 1976; Wainwright and Jordan, 2008). In the following sections we describe how to apply these techniques to the TerpreT models, and in particular, how to extend the methods to handle gates.

4.3.1 LP Relaxation

The inference problem can be phrased as the task of finding the highest scoring configuration of a set of discrete variables $X_0, \ldots, X_{D-1}$. The score is defined as the sum of local factor scores, θ_j, where $\theta_j : \boldsymbol{\mathcal{X}}_j \to \mathbb{R}$, and $\boldsymbol{\mathcal{X}}_j = \times_{i \in S_j} \mathcal{X}_i$ is the joint configuration space of the variables $\boldsymbol{x}$ with indices $S_j = (i_0, \ldots, i_{M_j})$ spanning the scope of factor j. In the simplest case (when we are searching for *any* valid

solution) the factor score at a node representing a function f_j will simply measure the consistency of the inputs ($\boldsymbol{x}_{\setminus 0}$) and output ($x_0$) at that factor:

$$\theta_j(\boldsymbol{x}) = \mathbb{1}\{x_0 = f_j(\boldsymbol{x}_{\setminus 0})\}. \tag{9}$$

Alongside these scoring functions, we can build a set of linear constraints and an overall linear objective function which represent the graphical model as an LP. The variables of this LP are the local unary marginals $\mu_i(x) \in \mathbb{R}$ as before, and new *local factor marginals* $\mu_{S_j}(\boldsymbol{x}) \in \mathbb{R}$ for $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}_j$ associated with each factor, j.

In the absence of gates, we can write the LP as:

$$\begin{aligned}
\max_{\boldsymbol{\mu}} \quad & \sum_j \sum_{\boldsymbol{x} \in \boldsymbol{\mathcal{X}_j}} \mu_{S_j}(\boldsymbol{x}) \theta_j(\boldsymbol{x}) \\
\text{s.t.} \quad & \mu_i(x) \geq 0 \; ; \; \mu_{S_j}(\boldsymbol{x}) \geq 0 \\
& \sum_{x \in \mathcal{X}_i} \mu_i(x) = 1 \\
& \sum_{\substack{\boldsymbol{x} \in \boldsymbol{\mathcal{X}}_j \\ X_i = x}} \mu_{S_j}(\boldsymbol{x}) = \mu_i(x),
\end{aligned} \tag{10}$$

where the final set of constraints say that when X_i is fixed to value x and all other variables are marginalized out from the local factor marginal, the result is equal to the value that the local marginal for X_i assigns to value x. This ensures that factors and their neighboring variables have consistent local marginals.

If all local marginals $\mu(\cdot)$ are *integral*, i.e., restricted to be 0 or 1, then the LP above becomes an integer linear program corresponding exactly to the original discrete optimization problem. When the local marginals are real-valued (as above), the resulting LP is not guaranteed to have equivalent solution to the original problem, and *fractional* solutions can appear. More formally, the LP constraints define what is known as the *local polytope* $\mathcal{M}_L$. which is an outer approximation to the convex hull of all valid integral configurations of the local marginals (known as the marginal polytope $\mathcal{M}$). In the case of program synthesis, fractional solutions are problematic, because they do not correspond to discrete programs and thus cannot be represented as source code or executed on new instances. When a fractional solution is found, heuristics such as rounding, cuts, or branch & bound search must be used in order to find an integral solution.

4.3.2 Linear Constraints in Gated Models

We now extend the LP relaxation above to cater for models with gates. In each gate we instantiate local unary marginals μ_i^ψ for each active variable and local factor marginals $\mu_{S_j}^\psi$ for each factor, where ψ is the path condition of the parent gate.

The constraints in the LP are then updated to handle these gate specific marginals as follows:

Normalization constraints. The main difference in the Gate LP from the standard LP is how normalization constraints are handled. The key idea is that *each local marginal in gate A is normalized to sum to A's gate marginal.* Thus the local marginal for X_i in the gate with path condition $(\psi, Y = y)$ with gate marginal μ_Y is:

$$\sum_{x \in \mathcal{X}_i} \mu_i^{\psi, Y=y}(x) = \mu_Y^\psi(y). \tag{11}$$

For local marginals in the global scope (not in any gate), the marginals are constrained to sum to 1, as in the standard LP.

Factor local marginals. The constraint enforcing local consistency between the factor local marginals and the unary local marginals is augmented with path condition superscripts:

$$\sum_{\boldsymbol{x} \in \boldsymbol{\mathcal{X}}_j : X_i = x} \mu_{S_j}^{\psi_A}(\boldsymbol{x}) = \mu_i^{\psi_A}(x). \tag{12}$$

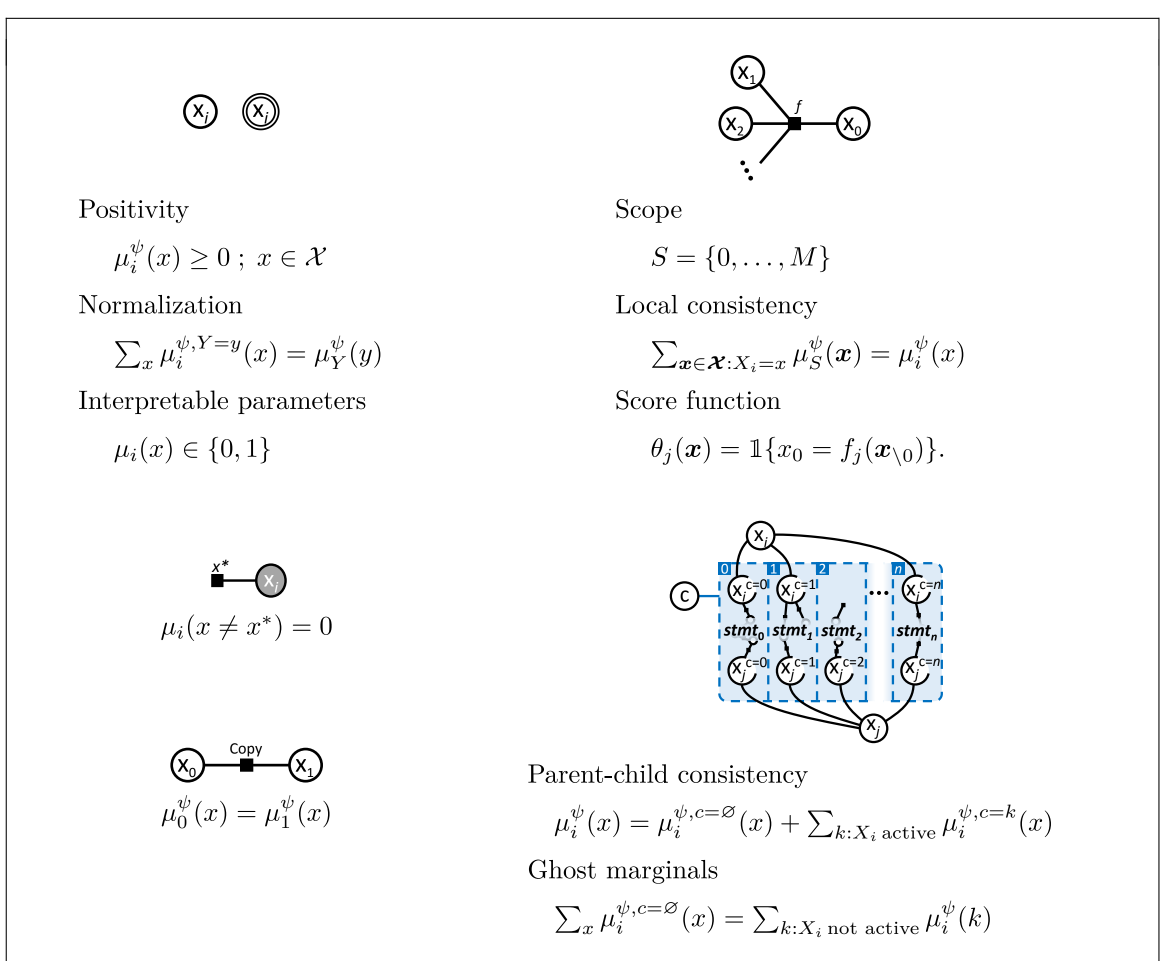


Figure 10: Summary of the construction of a mixed integer linear program from a gated factor graph. Removing the binary constraint $\mu_i(x) \in \{0, 1\}$ on the parameters produces a continuous LP relaxation. See main text for definition of symbols.

Parent-child consistency. There needs to be a relationship between different local marginals for the same variable. We do this by enforcing consistency between parent-child local marginals. Let A be a parent gate of B, and let X_i be active in both A and B. Then we need to enforce consistency between $\mu_i^{\psi_A}(x)$ and $\mu_i^{\psi_B}(x)$. It is not quite as simple as setting these quantities equal; in general there are multiple children gates of A, and X may be active in many of them. Let $B_1, \ldots, B_K$ be the set of children gates of A, and suppose that X is active in all of the children. Then the constraint is

$$\sum_{k=1}^{K} \mu_i^{\psi_{B_k}}(x) = \mu_i^{\psi_A}(x) \quad \forall x \in \mathcal{X}_i. \tag{13}$$

This can be thought of as setting a parent local marginal to be a weighted average of children local marginals, where the "weights" come from children marginals being capped at their corresponding gate marginal's value.

Ghost marginals. A problem arises if a variable is used in some but not all children gates. It may be tempting in this case to replace the above constraint with one that leaves out the children where the variable is inactive:

$$\sum_{k : X_i \text{ is active}} \mu_i^{\psi_{B_k}}(x) = \mu_i^{\psi_A}(x). \tag{14}$$

This turns out to lead to a contradiction. To see this, consider X_3 in Fig. 8. X_3 is inactive in the `(X0 == 1)` gate, and thus the parent-child consistency constraints would be

$$\mu_3^{X_0=0}(x) = \mu_3(x) \quad \forall x. \tag{15}$$

However, the normalization constraints for these local marginals are

$$\sum_x \mu_3^{X_0=0}(x) = \mu_0(0) \tag{16}$$

$$\sum_x \mu_3(x) = 1. \tag{17}$$

This implies that $\mu_0(0) = 1$, which means we must assign zero probability to the case when X_3 is not active. This removes the possibility of $X_0 = 1$ from consideration which is clearly undesirable, and if there are disjoint sets of variables active in the different children cases, then the result is an infeasible LP.

The solution is to instantiate *ghost marginals*, which are local marginals for a variable in the case where it is undefined (hence the term "ghost"). We denote a ghost marginal with a path condition entry where the value is set to $\varnothing$, as in $\mu_3^{X_0=\varnothing}(x)$. Ghost marginals represent the distribution over values in all cases where a variable is not defined, so the normalization constraints are defined as follows:

$$\sum_x \mu_i^{X_0=\varnothing}(x) = \sum_{k:X_i \text{ is not active}} \mu_0(k). \tag{18}$$

Finally, we can fix the parent-child consistency constraints in the case where a variable is active in some children. The solution is to consider the ghost marginal as one of the child cases. In the example of X_3, the constraint would be the following:

$$\mu_3(x) = \mu_3^{X_0=\varnothing}(x) + \sum_{k:X_3 \text{ is active}} \mu_3^{X_0=k}(x) \text{ for all } x \in \mathcal{X}_3. \tag{19}$$

The full set of constraints for solving TerpreT IPS problems using gated (integer) LPs is summarized in Fig. 10.

4.4 SMT Back-end

At its core, an IPS problem in TerpreT induces a simple linear integer constraint system. To exploit mature constraint-solving systems such as Z3 (de Moura and Bjørner, 2008), we have implemented a satisfiability modulo theories (SMT) back-end. For this, a TerpreT instance is translated into a set of constraints in the SMT-LIB standard (Barrett et al., 2015), after which any standard SMT solver can be called.

To this end, we have defined a syntax-guided transformation function $[\![\cdot]\!]^{\mathsf{E}}_{\mathsf{SMT}}$ that translates TerpreT expressions into SMT-LIB expressions over integer variables, shown in Fig. 11. We make use of the unrolling techniques discussed earlier to eliminate arrays, `for` loops and `with` statements. When encountering a function call as part of an expression, we use *inlining*, i.e., replace the call by the function definition in which formal parameters have been replaced by actual arguments. This means that some TerpreT statements have to be expressed as SMT-LIB expressions, and also means that the SMT back-end only supports a small subset of functions, namely those that are using only TerpreT (but not arbitrary Python) constructs.

Building on $[\![\cdot]\!]^{\mathsf{E}}_{\mathsf{SMT}}$, we then define the statement translation function $[\![\cdot]\!]_{\mathsf{SMT}}$ shown in Fig. 12. Every statement is translated into a list of constraints, and a solution to the IPS problem encoded by a TerpreT program p is a solution to the conjunction of all constraints generated by $[\![p]\!]^{\mathsf{E}}_{\mathsf{SMT}}$. Together with the unrolling of loops for a fixed length, this approach is reminiscent of bounded model checking techniques (e.g. (Clarke et al., 2001)) which for a given program, search for an input that shows some behavior. Instead, we take the input as given, and search for a program with the desired behavior.

$$
\begin{array}{lcl}
[\![n]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & n \\
[\![v_c]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & v_c \\
[\![f(c_1,\cdots,c_k)]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & [\![s[v_0/c_0,\cdots,v_n/c_k]]\!]^{\mathsf{E}}_{\mathsf{SMT}} \quad \text{for } \texttt{def } f\texttt{(}v_0\texttt{,}\cdots\texttt{,}v_k\texttt{):}s \\
[\![c_0 \; op_a \; c_1]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(}op_a \; [\![c_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![c_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![v]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & v \\
[\![a_0 \; op_a \; a_1]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(}op_a \; [\![a_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![a_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![f(a_0,\cdots,a_k)]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & [\![s[v_0/a_0,\cdots,v_k/a_k]]\!]^{\mathsf{E}}_{\mathsf{SMT}} \quad \text{for } \texttt{def } f\texttt{(}v_0\texttt{,}\cdots\texttt{,}v_k\texttt{):}s \\
[\![\texttt{not } b]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(not } [\![b]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![b_0 \texttt{ and } b_1]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(and } [\![b_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![b_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![b_0 \texttt{ or } b_1]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(or } [\![b_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![b_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![a_0 \; op_c \; a_1]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(}op_c \; [\![a_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![a_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![v \texttt{ = } a\texttt{;}s]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & [\![s[v/a]]\!]^{\mathsf{E}}_{\mathsf{SMT}} \\
[\![\texttt{if } b_0\texttt{: } s_0 \texttt{ else: } s_1]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & \texttt{(ite } [\![b_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![s_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![s_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)} \\
[\![\texttt{return } a]\!]^{\mathsf{E}}_{\mathsf{SMT}} & = & [\![a]\!]^{\mathsf{E}}_{\mathsf{SMT}}
\end{array}
$$

Figure 11: A syntax-directed translation $[\![\cdot]\!]^{\mathsf{E}}_{\mathsf{SMT}}$ of TerpreT expressions to SMT-LIB 2. Here, $s[var/expr]$ replaces all occurrences of var by $expr$.

$$
\begin{array}{lcl}
[\![s_0 \texttt{ ; } s_1]\!]_{\mathsf{SMT}} & = & [\![s_0]\!]_{\mathsf{SMT}} \; @ \; [\![s_1]\!]_{\mathsf{SMT}} \\
[\![a_0\texttt{.set_to(}a_1\texttt{)}]\!]_{\mathsf{SMT}} & = & [\texttt{(= } [\![a_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![a_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)}] \\
[\![a\texttt{.set_to_constant(}c\texttt{)}]\!]_{\mathsf{SMT}} & = & [\texttt{(= } [\![a]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![c]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)}] \\
[\![a_0\texttt{.observe_value(}a_1\texttt{)}]\!]_{\mathsf{SMT}} & = & [\texttt{(= } [\![a_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \; [\![a_1]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)}] \\
[\![\texttt{if } b_0\texttt{: } s_0 \texttt{ else: } s_1]\!]_{\mathsf{SMT}} & = & [\texttt{(=> } [\![b_0]\!]^{\mathsf{E}}_{\mathsf{SMT}} \texttt{ (and } [\![s_0]\!]_{\mathsf{SMT}}\texttt{)),} \\
 & & \;\texttt{(=> (not } [\![b_0]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{) (and } [\![s_1]\!]_{\mathsf{SMT}}\texttt{))}] \\
[\![s_{d_0} \texttt{ ; } s_{d_1}]\!]_{\mathsf{SMT}} & = & [\![s_{d_0}]\!]_{\mathsf{SMT}} \; @ \; [\![s_{d_1}]\!]_{\mathsf{SMT}} \\
[\![v \texttt{ = } c]\!]_{\mathsf{SMT}} & = & [\texttt{(= } v \; [\![c]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)}] \\
[\![v \texttt{ = Var(}c\texttt{)}]\!]_{\mathsf{SMT}} & = & [\texttt{(>= } v \texttt{ 0), (< } v \; [\![c]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)}] \\
[\![v \texttt{ = Param(}c\texttt{)}]\!]_{\mathsf{SMT}} & = & [\texttt{(>= } v \texttt{ 0), (< } v \; [\![c]\!]^{\mathsf{E}}_{\mathsf{SMT}}\texttt{)}]
\end{array}
$$

Figure 12: A syntax-directed translation $[\![\cdot]\!]_{\mathsf{SMT}}$ of TerpreT statements to SMT-LIB 2. Here, $[\cdots]$ are lists of constraints, and @ concatenates such lists.

```
harness void tripleSketch(int x){
  int h = ??;  // hole for unknown constant
  assert h * x == x + x + x;
}
```

Figure 13: A simple sketch example.

4.5 Sketch Back-end

The final back-end which we consider is based on the Sketch (Solar-Lezama, 2008) program synthesis system, which allows programmers to write partial programs called *sketches* while leaving fragments unspecified as *holes*. The goal of the synthesizer is to automatically fill in these holes such that the completed program conforms to a desired specification. The Sketch system supports multiple forms of specifications such as input-output examples, assertions, reference implementation, etc.

Background. The syntax for the Sketch language is similar to the C language with only one additional feature – a symbol `??` that represents an unknown constant integer value. A simple example sketch is shown in Fig. 13, which represents a partial program with an unknown integer `h` and a simple assertion. The `harness` keyword indicates to the synthesizer that it should compute a value for `h` such that in the complete function, all assertions are satisfied for all input values `x`. For this example, the Sketch synthesizer computes the value $\texttt{h} = 3$ as expected.

The unknown integer values can be used to encode a richer hypothesis space of program fragments. For example, the sketch in Fig. 14 uses an integer hole to describe a space of binary arithmetic operations. The Sketch language also provides a succinct language construct to specify such expression choices : `lhs {| + | - | * | / | % |} rhs`.

```
int chooseArithBinOp(int lhs, int rhs){
  int c = ??; // unknown constant integer value
  assert c < 5;
  if(c == 0) return lhs + rhs;
  if(c == 1) return lhs - rhs;
  if(c == 2) return lhs * rhs;
  if(c == 3) return lhs / rhs;
  if(c == 4) return lhs % rhs;
}
```

Figure 14: Using unknown integer values to encode a richer set of unknown expressions.

The Sketch synthesizer uses a counter-example guided inductive synthesis algorithm (CEGIS) (Solar-Lezama et al., 2006) to efficiently solve the second order exists-forall synthesis constraint. The key idea of the algorithm is to divide the process into phases : i) a synthesis phase that computes the value of unknowns over a finite set of input-output examples, and ii) a verification phase that checks if the current solution conforms to the desired specification. If the current completed program satisfies the specification, it returns the program as the desired solution. Otherwise, it computes a counter-example input that violates the specification and adds it to the set of input-output examples and continues the synthesis phase. More details about the CEGIS algorithm in Sketch can be found in Solar-Lezama (2008).

Compiling TerpreT to Sketch. In Fig. 15, we present a syntax-directed translation of the TerpreT language to Sketch. The key idea of the translation is to model `Param` variables as unknown integer constants (and integer arrays with constant values) such that the synthesizer computes the values of parameters to satisfy the observation constraints. For a `Param`(N) integer value, the translation creates corresponding unknown integer value `??` with an additional constraint that the unknown value should be less than N. Similarly, for the `Param`(N) array values, the translation creates a Sketch array with unknown integer values, where each value is constrained to be less than N. The `set_to` statements are

translated to assignment statements whereas the `observe` statements are translated to `assert` statements in SKETCH. The user-defined functions are translated directly to corresponding functions in sketch, whereas the `with` statements are translated to corresponding assignment statements. The sketch translation of the TERPRET model in Fig. 4(a) is shown in Fig. 16.

5 Analysis

One motivation of this work was to compare the performance of the gradient based FMGD technique for IPS with other back-ends. Below we present a task which all other back-ends solve easily, but FMGD is found to fail due to the prevalence of local optima.

5.1 Failure of FMGD

Kurach et al. (2015) and Neelakantan et al. (2016b) mention that many random restarts and a careful hyperparameter search are needed in order to converge to a correct deterministic solution. Here we develop an understanding of the loss surface that arises using FMGD in a simpler setting, which we believe sheds some light on the local optima structure that arises when using FMGD more generally.

Let $x_0, \ldots, x_{K-1}$ be binary variables with $x_0 = 0$ and all others unobserved. For each $k = 0, \ldots, K-1$, let $y_k = (x_k + x_{(k+1) \mod K}) \mod 2$ be the parity of neighboring x variables connected in a ring shape. Suppose all y_k are observed to be 0 and the goal is to infer the values of each x_k. The TERPRET program is as follows, which we refer to as the *Parity Chain* model:

```
1  const_K = 5
2  x = Param(2)[const_K]
3  y = Var(2)[const_K]
4
5  @CompileMe([2,2], 2)
6  def Parity(a,b): return (a + b) % 2
7
8  x[0].set_to_constant(0)
9
10 for k in range(K):
11    y[k].set_to(Parity(x[k], x[(k+1) % K]))
12    y[k].observe_value(0)
```

Clearly, the optimal configuration is to set all $x_k = 0$. Here we show analytically that there are exponentially many suboptimal local optima that FMGD can fall into, and experimentally that the probability of falling into a suboptimal local optimum grows quickly in K.

To show that there are exponentially many local optima, we give a technique for enumerating them and show that the gradient is equal to $\mathbf{0}$ at each. Letting $m_i(a)$ for $i \in \{0, \ldots K-1\}, a \in \{0,1\}$ be the model parameters and $\mu_i = \frac{\exp m_i(1)}{\exp m_i(0) + \exp m_i(1)}$, the main observation is that locally, a configuration of $[\mu_{i-1}, \mu_i, \mu_{i+1}] = [0, .5, 1]$ or $[\mu_{i-1}, \mu_i, \mu_{i+1}] = [1, .5, 0]$ gives rise to zero gradient on $m_{i-1}(\cdot), m_i(\cdot), m_{i+1}(\cdot)$, as does any configuration of $[\mu_{i-1}, \mu_i, \mu_{i+1}] = [0, 0, 0]$ or $[\mu_{i-1}, \mu_i, \mu_{i+1}] = [1, 1, 1]$. This implies that any configuration of a sequence of μ's of the form $[0, .5, 1, 1, \ldots, 1, .5, 0]$ also gives rise to zero gradients on all the involved m's. We then can choose any configuration of alternating μ (so e.g., $\mu_2, \mu_4, \mu_6, \ldots, \mu_K \in \{0,1\}^{K/2}$), and then fill in the remaining values of $\mu_1, \mu_3, \ldots$ so as to create a local optimum. The rule is to set $\mu_i = .5$ if $\mu_{i-1} + \mu_{i+1} = 1$ and $\mu_i = \mu_{i-1} = \mu_{i+1}$ otherwise. This will create "islands" of 1's, with the boundaries of the islands set to be .5. Each configuration of islands is a local optimum, and there are at least $2^{(K-1)/2}$ such configurations. A formal proof appears in Appendix A.

One might wonder if these local optima arise in practice. That is, if we initialize $m_i(a)$ randomly, will we encounter these suboptimal local optima? Experiments in Section 5.2 show that the answer is yes. The local optima can be avoided in small models by using optimization heuristics such as gradient noise, but

$$
\begin{aligned}
\llbracket n \rrbracket_{\mathsf{Sk}} &= n \\
\llbracket v_c \rrbracket_{\mathsf{Sk}} &= v_c \\
\llbracket f(c_1, \cdots, c_k) \rrbracket_{\mathsf{Sk}} &= f\texttt{(}\llbracket c_1 \rrbracket_{\mathsf{Sk}}\texttt{,} \cdots \texttt{,}\llbracket c_k \rrbracket_{\mathsf{Sk}}\texttt{)} \\
\llbracket c_0 \; op_a \; c_1 \rrbracket_{\mathsf{Sk}} &= \llbracket c_0 \rrbracket_{\mathsf{Sk}} \; op_a \; \llbracket c_1 \rrbracket_{\mathsf{Sk}} \\
\llbracket v \rrbracket_{\mathsf{Sk}} &= v \\
\llbracket v[a_1, \cdots, a_k] \rrbracket_{\mathsf{Sk}} &= v\texttt{[}a_1\texttt{]}\texttt{[}\cdots\texttt{]}\texttt{[}a_k\texttt{]} \\
\llbracket a_0 \; op_a \; a_1 \rrbracket_{\mathsf{Sk}} &= \llbracket a_0 \rrbracket_{\mathsf{Sk}} \; op_a \; \llbracket a_1 \rrbracket_{\mathsf{Sk}} \\
\llbracket f(a_0, \cdots, a_k) \rrbracket_{\mathsf{Sk}} &= f\texttt{(}\llbracket a_0 \rrbracket_{\mathsf{Sk}}\texttt{,} \cdots \texttt{,}\llbracket a_k \rrbracket_{\mathsf{Sk}}\texttt{)} \\
\llbracket \texttt{not } b \rrbracket_{\mathsf{Sk}} &= \texttt{!(}\llbracket b \rrbracket_{\mathsf{Sk}}\texttt{)} \\
\llbracket b_0 \texttt{ and } b_1 \rrbracket_{\mathsf{Sk}} &= \llbracket b_0 \rrbracket_{\mathsf{Sk}} \texttt{ \&\& } \llbracket b_1 \rrbracket_{\mathsf{Sk}} \\
\llbracket b_0 \texttt{ or } b_1 \rrbracket_{\mathsf{Sk}} &= \texttt{(}\llbracket b_0 \rrbracket_{\mathsf{Sk}} \texttt{ || } \llbracket b_1 \rrbracket_{\mathsf{Sk}}\texttt{)} \\
\llbracket a_0 \; op_c \; a_1 \rrbracket_{\mathsf{Sk}} &= \llbracket a_0 \rrbracket_{\mathsf{Sk}} \; op_c \; \llbracket a_1 \rrbracket_{\mathsf{Sk}} \\
\llbracket s_0 \texttt{ ; } s_1 \rrbracket_{\mathsf{Sk}} &= \llbracket s_0 \rrbracket_{\mathsf{Sk}}\texttt{;} \; \llbracket s_1 \rrbracket_{\mathsf{Sk}} \\
\llbracket a_0\texttt{.set_to(}a_1\texttt{)} \rrbracket_{\mathsf{Sk}} &= \llbracket a_0 \rrbracket_{\mathsf{Sk}} \texttt{ = } \llbracket a_1 \rrbracket_{\mathsf{Sk}}\texttt{;} \\
\llbracket a\texttt{.set_to_constant(}c\texttt{)} \rrbracket_{\mathsf{Sk}} &= \llbracket a \rrbracket_{\mathsf{Sk}} \texttt{ = } \llbracket c \rrbracket_{\mathsf{Sk}}\texttt{;} \\
\llbracket a_0\texttt{.observe_value(}a_1\texttt{)} \rrbracket_{\mathsf{Sk}} &= \texttt{assert } \llbracket a_0 \rrbracket_{\mathsf{Sk}} \texttt{ == } \llbracket a_1 \rrbracket_{\mathsf{Sk}}; \\
\llbracket \texttt{return } a \rrbracket_{\mathsf{Sk}} &= \texttt{return } \llbracket a \rrbracket_{\mathsf{Sk}}\texttt{;} \\
\llbracket \texttt{if } b_0\texttt{: } s_0 \texttt{ else: } s_1 \rrbracket_{\mathsf{Sk}} &= \texttt{if (}\llbracket b_0 \rrbracket_{\mathsf{Sk}}\texttt{) } \{\llbracket s_0 \rrbracket_{\mathsf{Sk}}\} \texttt{ else } \{\llbracket s_1 \rrbracket_{\mathsf{Sk}}\} \\
\llbracket \texttt{for } v \texttt{ in range}(c_1) \texttt{ : } s \rrbracket_{\mathsf{Sk}} &= \texttt{for(int } v \texttt{ = 0; } v \texttt{ < } \llbracket c_1 \rrbracket_{\mathsf{Sk}}\texttt{; } v\texttt{++) } \{\llbracket s \rrbracket_{\mathsf{Sk}}\} \\
\llbracket \texttt{for } v \texttt{ in range}(c_1, c_2) \texttt{ : } s \rrbracket_{\mathsf{Sk}} &= \texttt{for(int } v \texttt{ = } \llbracket c_1 \rrbracket_{\mathsf{Sk}}\texttt{; } v \texttt{ < } \llbracket c_2 \rrbracket_{\mathsf{Sk}}\texttt{; } v\texttt{++) } \{\llbracket s \rrbracket_{\mathsf{Sk}}\} \\
\llbracket \texttt{with } a \texttt{ as } v \texttt{ : } s \rrbracket_{\mathsf{Sk}} &= \{\texttt{int } v \texttt{ = } \llbracket a \rrbracket_{\mathsf{Sk}}\texttt{; } \llbracket s \rrbracket_{\mathsf{Sk}}\} \\
\llbracket s_{d_0} \texttt{ ; } s_{d_1} \rrbracket_{\mathsf{Sk}} &= \llbracket s_{d_0} \rrbracket_{\mathsf{Sk}}\texttt{; } \llbracket s_{d_1} \rrbracket_{\mathsf{Sk}} \\
\llbracket v \texttt{ = } c \rrbracket_{\mathsf{Sk}} &= \texttt{int } v \texttt{ = } \llbracket c \rrbracket_{\mathsf{Sk}}\texttt{;} \\
\llbracket v \texttt{ = Var(}c\texttt{)} \rrbracket_{\mathsf{Sk}} &= \texttt{int } v\texttt{;} \\
\llbracket v \texttt{ = Var(}c\texttt{)[}c_1\texttt{,} \cdots \texttt{,}c_k\texttt{]} \rrbracket_{\mathsf{Sk}} &= \texttt{int[}\llbracket c_1 \rrbracket_{\mathsf{Sk}}\texttt{][}\cdots\texttt{][}\llbracket c_k \rrbracket_{\mathsf{Sk}}\texttt{] } v\texttt{;} \\
\llbracket v \texttt{ = Param(}c\texttt{)} \rrbracket_{\mathsf{Sk}} &= \texttt{int } v \texttt{ = ??; assert } v \texttt{ < } \llbracket c \rrbracket_{\mathsf{Sk}}\texttt{;} \\
\llbracket v \texttt{ = Param(}c\texttt{)[}c_1\texttt{,} \cdots \texttt{,}c_k\texttt{]} \rrbracket_{\mathsf{Sk}} &= \texttt{int[}\llbracket c_1 \rrbracket_{\mathsf{Sk}}\texttt{][}\cdots\texttt{][}\llbracket c_k \rrbracket_{\mathsf{Sk}}\texttt{] } v\texttt{;} \\
&\quad\; \forall i_1 \in \llbracket c_1 \rrbracket_{\mathsf{Sk}}, \cdots, i_k \in \llbracket c_k \rrbracket_{\mathsf{Sk}} : \\
&\qquad v\texttt{[}i_1\texttt{][}\cdots\texttt{][}i_k\texttt{] = ??; assert } v\texttt{[}i_1\texttt{][}\cdots\texttt{][}i_k\texttt{] < } \llbracket c \rrbracket_{\mathsf{Sk}}\texttt{;} \\
\llbracket \texttt{@CompileMe([}c_1\texttt{,} \cdots \texttt{,}c_k\texttt{],}c_r\texttt{)} \rrbracket_{\mathsf{Sk}} &= \texttt{;} \\
\llbracket \texttt{def } f\texttt{(}v_0\texttt{,} \cdots \texttt{,}v_k\texttt{):}s \rrbracket_{\mathsf{Sk}} &= \texttt{int } f\texttt{(}v_0\texttt{, } \cdots \texttt{,}v_k\texttt{) } \{\llbracket s \rrbracket_{\mathsf{Sk}}\}
\end{aligned}
$$

Figure 15: A syntax-directed translation $\llbracket \cdot \rrbracket_{\mathsf{Sk}}$ of TerpreT programs to Sketch.

```
int const_n = 5;
int[2][2] ruleTable = (int[2][2]) ??;
for(int i=0; i<2; i++){
  for(int j=0; j<2; j++){
    assert ruleTable[i][j] < 2;
  }
}
int[const_n] tape;

// assignment statements for input initialisations

for(int t=1; t<const_n-1; t++){
  int x1 = tape[t];
  int x0 = tape[t-1];
  tape[t+1] = ruleTable[x0,x1];
}

// assert statements for corresponding outputs
```

Figure 16: The sketch translation for the TerpreT model shown in Fig. 4(a) .

as the models grow larger (length 128), we were not able to find any configuration of hyperparameters that could solve the problem from a random initialization. The other inference algorithms will solve these problems easily. For example, the LP relaxation from Section 4.3 will lead to integral solutions for tree-structured graphical models, which is the case here.

5.2 Parity Chain Experiments

Here we provide an empirical counterpart to the theoretical analysis in the previous section. Specifically, we showed that there are exponentially many local optima for FMGD to fall into in the Parity Chain model, but this does not necessarily mean that these local optima are encountered in practice. It is conceivable that there is a large basin of attraction around the global optimum, and smaller, negligible basins of attraction around the suboptimal local optima.

To answer this question, we run Vanilla FMGD (no optimization heuristics) with random initialization parameters chosen so that initial parameters are drawn uniformly from the simplex. Measuring the fraction of runs (from 100 random initializations) that converge to the global optimum then gives an estimate of the volume of parameter space that falls within the basin of attraction for the global optimum. Results for chain lengths of $K = 4, 8, 16, 32, 64, 128$ appear in the Vanilla FMGD row of Table 3. FMGD is able to solve very small instances reliably, but performance quickly falls off as K grows. This shows that the basins of attraction for the suboptimal local optima are large.

Next, we try the optimization heuristics discussed in Section 4.2.3. For each chain length K, we draw 100 random hyperparameter settings from the manually chosen hyperparameter distribution. At each hyperparameter setting, we run 10 runs with different random seeds and measure the fraction of runs that converge to the global optimum. In the "Best Hypers" row of Table 3, we report the percentage of successes from the hyperparameter setting that yielded the best results. In the "Average Hypers" row, we report the percentage of success across all 1000 runs.

Note that the 80% success rate for Best Hypers on $K = 64$ is an anomaly, as it was able to find a setting of hyperparameters for which the random initialization had very little effect, and the successful runs followed nearly identical learning trajectories. See Fig. 17 for a plot of optimization objective versus epoch. Successful runs are colored blue while unsuccessful ones are in red. The large cluster of successful runs were all from the same hyperparameter settings.

	$K = 4$	$K = 8$	$K = 16$	$K = 32$	$K = 64$	$K = 128$
Vanilla FMGD	100%	53%	14%	0%	0%	0%
Best Hypers	100%	100%	100%	70%	80%	0%
Average Hypers	84%	42%	21%	4%	1%	0%

Table 3: Percentage of runs that converge to the global optimum for FMGD on the Parity Chain example.

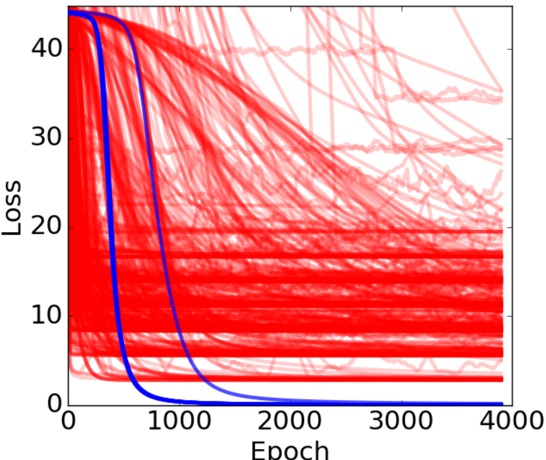


Figure 17: Loss versus epoch for all runs of the random search on Parity Chain ($K = 64$).

6 Experiments

We now turn attention to experimental results. Our primary aim is to better understand the capabilities of the different back-ends on a range of problems, and to establish some trends regarding the performance of the back-ends as problem properties are varied.

6.1 Benchmarks Results

We now present the main results of this investigation: a benchmarking of all four inference techniques listed in Table 1 on all twelve synthesis tasks listed in Table 2. As described in Section 3.2, the tasks are split across four execution models of increasing practicality, with each model set three tasks of increasing difficulty. For any given task we ensure a fair test by presenting all four back-end compilers with the same TerpreT program (as listed in Appendix B) and the same set of input-output examples. Since the different back-ends use very different approaches to solve the tasks, the only comparable metric to record is the wall time to reach a solution.

With the exception of the FMGD algorithm, we set a timeout of 4 hours for each back-end on each task (excluding any compilation time), give each algorithm a single run to find a solution, and do not tune the algorithms to each specific task. For the FMGD algorithm we run both the Vanilla and the Optimized form. In the Vanilla case we report the fraction of 20 different random initializations which lead to a globally optimal solution and also the wall clock time for 1000 epochs of the gradient descent algorithm (which is the typical number of iterations required to reach convergence on a successful run). In the Optimized FMGD case, we follow a similar protocol as in the previous section but allow training to run for 2000 epochs. We use the same manually chosen distribution over hyperparameters to perform a random search, drawing 120 random settings of hyperparameters. For each setting we run the learning with 20 different random initializations. For the ListK tasks, the runtime was very long, so we ran for fewer settings of hyperparameters (28 for Assembly and 10 for Basic Block). As before, in the Optimized case we report the success rate for the best hyperparameters found and also for the average across all runs in the random search.

Our results are compiled in Table 4, from which we can draw two high level conclusions:

Back end algorithm. There is a clear tendency for traditional techniques employing constraint solvers (SMT and Sketch) to outperform the machine learning methods, with Sketch being the only system able to solve all of these benchmarks before timeout (see Section 6.3).

Nevertheless, the machine learning methods have qualitatively appealing properties. Firstly, they are primarily *optimizers* rather than solvers[2], and additional terms could be added to the cost function of the optimization to find programs with desired properties (e.g. minimal length (Bunel et al., 2016) or resource usage). Secondly, FMGD makes the synthesis task fully differentiable, allowing its incorporation into a larger end-to-end differentiable system (Kurach et al., 2015). This encourages us to persevere with analysis of the FMGD technique, and in particular to study the surprising failure of this method on the simple boolean circuit benchmarks in Section 6.2.

Interpreter models. Table 4 highlights that the precise formulation of the interpreter model can affect the speed of synthesis. Both the Basic Block and Assembly models are equally expressive, but the Assembly model is biased towards producing straight line code with minimal branching. In all cases where synthesis was successful the Assembly representation is seen to outperform the Basic Block model in terms of synthesis time. The only anomaly is that the Optimized FMGD algorithm is able to find a solution in the Decrement task using the Basic Block model, but not the Assembly model. This could be because the *minimal* solution to this program is shorter in the Basic Block architecture than in the Assembly model ($T = 18$ vs. 27 respectively). We observe in Section 6.2.2 that increasing the size of a model by adding *superfluous* resources can help the FMGD algorithm to converge on a global optimum. However, we generally find that synthesis is difficult if the minimal solution is already large.

[2] Both Sketch and SMT (in the form of max-SMT) can also be configured to be optimizers

				FMGD				ILP	SMT	Sketch
	$\log_{10}(D)$	T	N	Time	Vanilla	Best Hypers	Average Hypers	Time	Time	Time
TURING MACHINE										
Invert	4	6	5	76.5	100%	100%	51%	0.6	0.7	3.1
Prepend zero	9	6	5	98	60%	100%	37%	17.0	0.9	2.6
Binary decrement	9	9	5	163	5%	25%	2%	191.9	1.6	3.3
BOOLEAN CIRCUITS										
2-bit controlled shift register	10	4	8	-	-	-	-	2.5	0.7	2.7
Full adder	13	5	8	-	-	-	-	38	1.9	3.5
2-bit adder	22	8	16	-	-	-	-	13076.5	174.4	355.4
BASIC BLOCK										
Access	14	5	5	173.8	15%	50%	1.1%	98.0	14.4	4.3
Decrement	19	18	5	811.7	-	5%	0.04%	-	-	559.1
List-K	33	11	5	-	-	-	-	-	-	5493.0
ASSEMBLY										
Access	13	5	5	134.6	20%	90%	16%	3.6	10.5	3.8
Decrement	20	27	5	-	-	-	-	-	-	69.4
List-K	29	16	5	-	-	-	-	-	-	16.8

Table 4: Benchmark results. For FMGD we present the time in seconds for 1000 epochs and the success rate out of {20, 20, 2400} random restarts in the {Vanilla, Best Hypers and Average Hypers} columns respectively. For other back-ends we present the time in seconds to produce a synthesized program. The symbol - indicates timeout (> 4h) or failure of any random restart to converge. N is the number of provided input-output examples used to specify the task in each case.

6.2 Zooming in on FMGD Boolean Circuits

There is a stark contrast between the performance of FMGD and the alternatives on the Boolean Circuit problems. On the Controlled Shift and Full Adder benchmarks, each run of FMGD took $35 - 80\times$ as long as the SMT back-end. On top of this, we ran $120 \times 20 = 2400$ runs during the random search. However, there were no successful runs.

6.2.1 Slow convergence

While most runs converged to a local optimum during the 2000 epochs they were allocated, some cases had not. Thus, we decided to allocate the algorithm $5\times$ as many epochs (10,000) and run the random search over. This did produce some successes, although very few. For the Controlled Shift problem, 1 of 2400 runs converged, and for the Full Adder, 3 of 2400 runs converged. Thus it does appear that results could be improved somewhat by running FMGD for longer. However, given the long runtimes of FMGD relative to the SMT and Sketch back-ends, this would not change the qualitative conclusions from the previous section.

6.2.2 Varying the problem dimension

We take inspiration from neural network literature which approaches the issue of stagnation in local minima by increasing the dimension of the problem. It has been argued that local optima become increasingly rare in neural network loss surfaces as the dimension of the hidden layers increase, and instead saddle points become increasingly common (Dauphin et al., 2014). Exchanging local minima for saddle points is beneficial because dynamic learning rate schedules such as RMSProp are very effective at handling saddle points and plateaus in the loss function.

To assess how dimensionality affects FMGD, we first take a minimal example in the boolean circuit domain: the task of synthesizing a NAND gate. The minimum solution for this task is shown in Fig. 18(a), along with an example configuration which resides at one local minimum of the FMGD loss surface. For a synthesis task involving two gates and two wires, there are a total of 14 independent degrees of freedom to be optimized, and there is only one global optimum. Increasing the available resources to three gates and three wires, gives an optimization problem over 30 dimensions and several global minima. The contrast between the learning trajectories in these two cases is shown in Fig. 18. We attempt to infer the presence of saddle points by exploring the loss surface with vanilla gradient descent and a small learning rate. Temporary stagnation of the learning is an indication of a saddle-like feature. Such features are clearly more frequently encountered in the higher dimensional case where we also observe a greater overall success rate (36% of 100 random initializations converge on a global optimum in the low dimensional case vs. 60% in the high dimensional case).

These observations are consistent with the intuition from Dauphin et al. (2014), suggesting that we will have more success in the benchmark tasks if we provide more resources (i.e. a higher dimensional problem) than required to solve the task. We perform this experiment on the full adder benchmark by varying the number of wires and gates used in the synthesized solution. The results in Fig. 19 show the expected trend, with synthesis becoming more successful as the number of redundant resources increases above the minimal 4 wires and 5 gates. Furthermore, we see no clear increase in the expected time for FMGD to arrive at a solution as we increase the problem size (calculated as the time for 1000 epochs divided by the success rate). This is dramatically different to the trend seen when applying constraint solvers to the synthesis problem, where broadly speaking, increasing the number of constraints in the problem increases the time to solution (see Fig. 19(c)).

This feature of FMGD is particularly interesting when the minimal resources required to solve a problem is not known before synthesis. Whereas over-provisioning resources will usually harm the other back-ends we consider, it can help FMGD. The discovered programs can then be post-processed to recover some efficiency (see Fig. 20)

6.3 Challenge Benchmark

Before leaving this section, we note that Sketch has so far solved all of the benchmark tasks. To provide a goal for future work, we introduce a final benchmark which none of the back-ends are currently able to

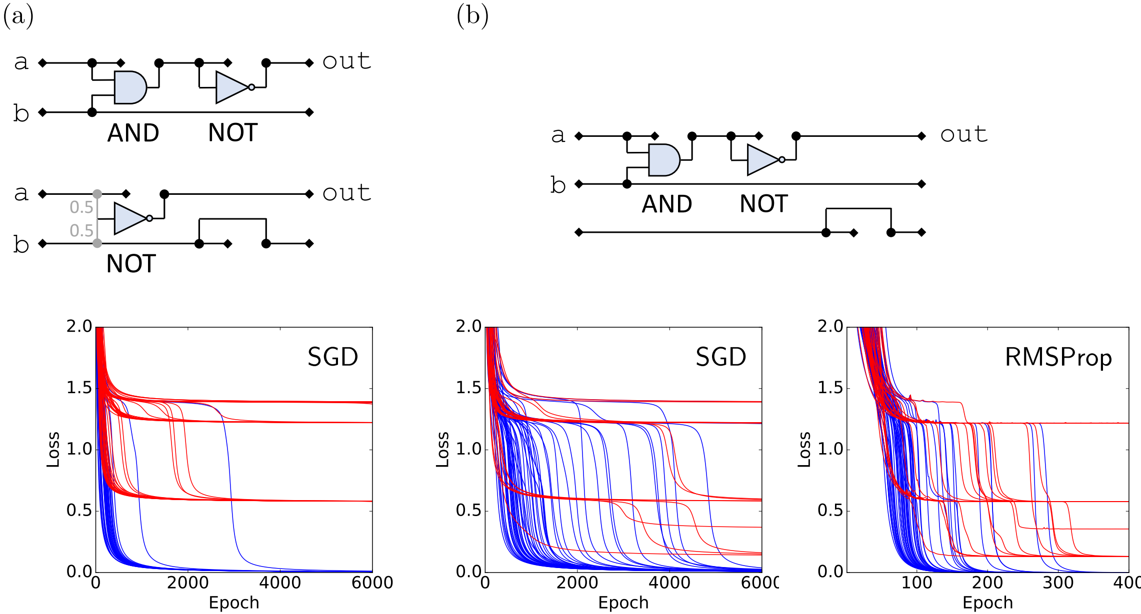
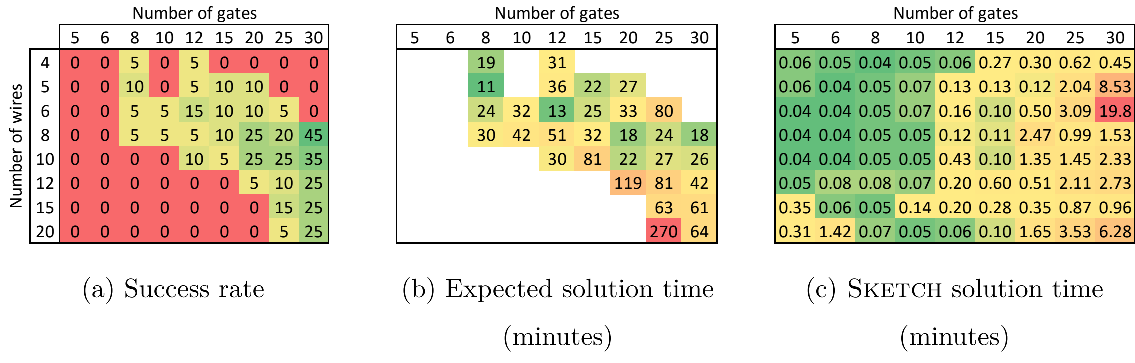


Figure 18: Comparison of the learning trajectories when we increase the resources available to the synthesizer. (a) The minimal solution for producing a `NAND` gate, and a locally optimal configuration found by FMGD (the marginal for the input to the first gate puts 50% weight on both wires). We plot the trajectory of 100 random initializations during learning, highlighting the successful runs in blue and the failures in red. (b) Adding an extra wire and gate changes the learning trajectories significantly, introducing plateaus indicative of saddle-points in the loss function. We can use RMSProp rather than SGD (stochastic gradient descent) to navigate these features (note the change in the horizontal scale when using RMSProp)



Number of wires \ Number of gates	5	6	8	10	12	15	20	25	30
4	0	0	5	0	5	0	0	0	0
5	0	0	10	0	5	10	10	0	0
6	0	0	5	5	15	10	10	5	0
8	0	0	5	5	5	10	25	20	45
10	0	0	0	0	10	5	25	25	35
12	0	0	0	0	0	0	5	10	25
15	0	0	0	0	0	0	0	15	25
20	0	0	0	0	0	0	0	5	25

(a) Success rate

Number of gates	5	6	8	10	12	15	20	25	30
4			19		31				
5			11		36	22	27		
6			24	32	13	25	33	80	
8			30	42	51	32	18	24	18
10					30	81	22	27	26
12							119	81	42
15								63	61
20								270	64

(b) Expected solution time (minutes)

Number of gates	5	6	8	10	12	15	20	25	30
4	0.06	0.05	0.04	0.05	0.06	0.27	0.30	0.62	0.45
5	0.06	0.04	0.05	0.07	0.13	0.13	0.12	2.04	8.53
6	0.04	0.04	0.05	0.07	0.16	0.10	0.50	3.09	19.8
8	0.04	0.04	0.05	0.05	0.12	0.11	2.47	0.99	1.53
10	0.04	0.04	0.05	0.05	0.43	0.10	1.35	1.45	2.33
12	0.05	0.08	0.08	0.07	0.20	0.60	0.51	2.11	2.73
15	0.35	0.06	0.05	0.14	0.20	0.28	0.35	0.87	0.96
20	0.31	1.42	0.07	0.05	0.06	0.10	1.65	3.53	6.28

(c) SKETCH solution time (minutes)

Figure 19: The effect of increasing the dimension of the FMGD synthesis problem. We vary the number of wires and gates used in the synthesis of the full adder citcuit and present both (a) the percentage of 20 random initializations which converge to a global solution and (b) the expected time (in minutes) to solution (time for 1000 epochs / success rate). The solution time for the SKETCH—backend is shown in (c) for comparison.

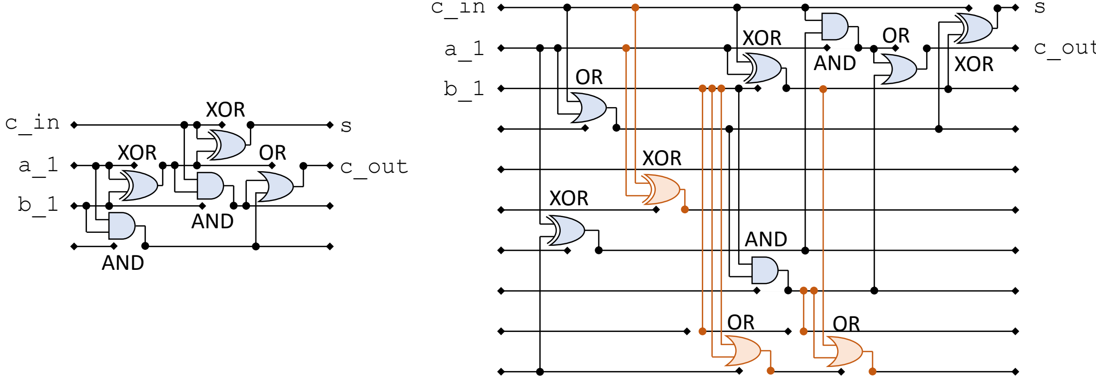


Figure 20: Comparison of the minimal Full Adder solution and the synthesized solution when redundant resources are provided. The gates highlighted in orange can be removed in post-processing without affecting the output.

solve from 5 input-output examples even after 40 hours:

ASSEMBLY	M	R	B	$\log_{10} D$	T	Description
Merge	17	6	22	103	69	Merge two contiguous sorted lists into one contiguous sorted list. The first entries in the initial heap are $\texttt{heap}_0\texttt{[0]} = p_1$, $\texttt{heap}_0\texttt{[1]} = p_2$, $\texttt{heap}_0\texttt{[2]} = p_{\text{out}}$, where p_1 (p_2) is a pointer to the first (second) sorted sublist (terminated with a 0), and p_{out} is a pointer to the head of the desired output list. All elements of the sorted sublists are larger than 0 and all unused cells in $\texttt{heap}_0$ are initialized to 0.

7 Related Work

Probabilistic Programming and Graphical Models There are now many probabilistic programming systems specialized to different use-cases. One dominant axis of variability is in the expressiveness of the language. Some probabilistic programming languages, exemplified by Church (Goodman et al., 2008), allow great freedom in the expressibility of the language, including constructs like recursion and higher order functions. The cost of the expressibility in the language is that the inference techniques cannot be as specialized, and thus these systems tend to use general Markov Chain Monte Carlo methods for inference. On the other end of the spectrum are more specialized systems like Infer.NET (Minka et al., 2014) and Stan (Carpenter, 2015; Stan Development Team, 2015). These systems restrict models to be constructed of predefined building blocks and do not support arbitrary program constructs like recursion and higher order functions. They do generally support basic loops and branching structure, however. In Infer.NET, for example, loops are unrolled, and `if` statements are handled via special constructs known as Gates (Minka and Winn, 2009). The result is that the program can be viewed as a finite gated factor graph, on which message passing inference can be performed.

In these terms, TerpreT is most similar to Infer.NET, and its handling of loops and `if` statements are inspired by Infer.NET. Compared to Infer.NET, TerpreT is far more extreme in the restrictions that it places upon modelling constructs. The benefit is that the restricted language allows us to support a broader range of back-ends. Looking forward, Infer.NET provides inspiration for how TerpreT might be extended to handle richer data types like real numbers and strings.

Another related line of work is in casting program synthesis as a problem of inference in probabilistic models. Gulwani and Jojic (2007) phrase program synthesis as inference in a graphical model and use

belief propagation inference. In future work, we would like to create a belief propagation-based back-end for TerpreT. The problem of inducing samplers for probability distributions has also been cast as a problem of inference in a probabilistic program (Perov and Wood, 2016). Lake et al. (2015) induce probabilistic programs by performing inference in a probabilistic model describing how primitives are composed to form types and instances of types.

Neural Networks with Memory In common neural network architectures handling sequences of inputs, "memory" only manifests itself as the highly compressed state of the network. This is problematic when the task at hand requires to relate inputs that are far apart from each other, which more recent models try to mitigate using tools such as Long Short-Term Memory (Hochreiter and Schmidhuber, 1997; Graves, 2013) and Gated Recurrent Units (Cho et al., 2014). However, such recurrent units do not entirely solve the problems with long-range interactions, and a range of additional techniques have been employed to improve results (e.g., (Mikolov et al., 2015; Koutník et al., 2014)).

An alternative solution to this problem is to extend networks by providing access to external storage. Initial extensions provided a stack (Giles et al., 1989) or a scratch pad simplified to a stack (Mozer and Das, 1992), and the controlling network learned when to push data to and pop (load) data from that stack.[3] Recently, similar ideas have been picked up again, leading to stack and queue-augmented recurrent nets (Joulin and Mikolov, 2015; Grefenstette et al., 2015), memory networks with freely addressable storage (Weston et al., 2014; Sukhbaatar et al., 2015), and extensions that additionally use registers for intermediate results (Kurach et al., 2015).

Neural Networks Learning Algorithms Recurrent neural networks with access to memory are, essentially, learnable implementations of the Von Neumann architecture. A number of recent advances build on this observation to learn algorithms from input-output data (Graves et al., 2014; Joulin and Mikolov, 2015; Neelakantan et al., 2016a; Reed and de Freitas, 2016; Zaremba et al., 2016). While these approaches differ in (a) the underlying execution models (e.g., Turing Machines, Random Access Machines, Stack Automata), (b) learning methods (e.g., from input/output samples or action sequences, supervised or by reinforcement learning), and (c) program domains (arithmetic, simple data structure manipulation, image manipulation), they share the overall idea of training a deep neural network that learns to manipulate data by repeatedly calling deterministic "modules" (or "actions" or "interfaces") from a predefined set. These models are able to learn and repeat simple algorithmic patterns, but are all not interpretable; what they have learned only becomes evident through actions on concrete inputs.

Very recent work has improved on this aspect and is closest to our approach. To support adaptive neural compilation (Bunel et al., 2016), a machine model similar to our assembly model (cf. Section 3.2.4) was introduced. This allows a user to *sketch* a (partial) program as input, and then use deep learning methods to optimise it. The result is again a program in the chosen assembly language, and can be displayed easily. Differentiable Forth (Riedel et al., 2016) is a similar step in this direction, where the learning task is to fill in holes in a partial Forth program.

Program Synthesis The area of program synthesis has recently seen a renewed interest in the programming language community (Alur et al., 2015). There have been many synthesis techniques developed for a wide range of problems including data wrangling (Gulwani et al., 2012; Polozov and Gulwani, 2015), inference of efficient synchronization in concurrent programs, synthesizing efficient low-level code from partial programs (Solar-Lezama et al., 2005), compilers for low-power spatial architectures (Phothilimthana et al., 2014), efficient compilation of declarative specifications (Kuncak et al., 2010), statistical code completion (Raychev et al., 2016), and automated feedback generation for programming assignments (Singh et al., 2013). These techniques can be broadly categorized using three dimensions: i) specification mechanism, ii) complexity of hypothesis space, and iii) search algorithm. The different forms of specifications include input-output examples, partial programs, reference implementation, program traces etc. The hypothesis space of possible programs is typically defined using a domain-specific language, which is designed to be expressive enough to encode majority of desired tasks but at the same time concise enough for efficient

[3] Interestingly enough, extracting an interpretable deterministic pushdown automaton from a trained stack-using recurrent network was already proposed in (Das et al., 1992).

learning. Finally, some of the common search algorithms include constraint-based symbolic synthesis algorithms (Solar-Lezama, 2008; Reynolds et al., 2015), smart enumerative algorithms with pruning (Udupa et al., 2013), version-space algebra based search algorithms (Gulwani et al., 2012; Gulwani, 2011), and stochastic search (Schkufza et al., 2013). There has also been some recent work on learning from inputs in addition to the input-output examples to guide the synthesis algorithm (Singh, 2016), and synthesizing programs without any examples by performing a joint inference over the program and the inputs to recover compressed encodings of the observed data (Ellis et al., 2015).

In this work, we are targeting specifications based on input-output examples as this form of specification is most natural to the work in the machine learning community. For defining the hypothesis space, we use our probabilistic programming language TerpreT, and we currently support compilation to intermediate representations for four inference (search) algorithms. The key difference between our work and most of the previous work in the program synthesis community is that our language is built to allow compilation to different inference algorithms (from both the machine learning community and programming languages community) which enables like-to-like comparison. We note that another recent effort SyGuS (Alur et al., 2015) aims to unify different program synthesis approaches using a common intermediate format based on context-free grammars so that different inferences techniques can be compared, but the TerpreT language allows for encoding richer programming models than SyGuS, and also allows for compilation to gradient-descent based inference algorithms.

8 Discussion & Future Work

We presented TerpreT, a probabilistic programming language for specifying IPS problems. TerpreT can be used in combination with the FMGD back-end to produce differentiable interpreters for a wide range of program representations and languages. TerpreT has several other back-ends including one based on linear programming and two that are strong alternatives from the programming languages community.

The biggest take-away from the experimental results is that the methods from programming languages significantly outperform the machine learning approaches. We believe this is an important take-away for machine learning researchers studying program synthesis. However, we remain optimistic about the future of machine learning-based approaches to program synthesis, and we do not wish to discourage work in this area. Quite the opposite; we hope that this work stimulates further research in the area and helps to clarify how machine learning is likely to be useful. The setting in this work is a minimal version of the program synthesis problem, in which the main challenge is efficiently searching over program space for programs that meet a given input-output specification. The conclusion from our experiments is that gradient descent is inferior to constraint-based discrete search algorithms for this task.

Our results also raise an interesting question when taken in comparison to (Kurach et al., 2015). The NRAM model is reported to solve problems that our FMGD approach was not able to. We would like to better understand what the source of this discrepancy is. The two main differences are in the execution model and in the program parameterization. In the NRAM execution model, all instructions are executed in all timesteps (possibly multiple times), and it is up to the controller to decide how to wire them up. This creates additional parallelism and redundancy relative to the Basic Block or Assembly models. We speculate that this makes it possible in the NRAM model for multiple hypotheses to be developed at once with little overlap in the memory locations used. This property may make the optimization easier. The other major difference is in the controller. The NRAM model uses a neural network controller that maps from the state of registers to the operations that are to be performed. In the Basic Block and Assembly models, this is done via the instruction pointer that is updated based upon control flow decisions in the program. We speculate that the neural network controller operates in a less constrained space (since a different operation may be performed at each timestep if the model pleases), and it offers some bias to the search. We suspect neural networks may be biased towards repeating circuits in the earlier stages of training due to their typical behavior of first predicting averages before specializing to make strongly input-dependent predictions. In future work we would like to explore these questions more rigorously, in hopes of finding general principles that can be used to develop more robust inference algorithms.

In future work, there are several extensions to TerpreT that we would like to develop. First, we would like to extend the TerpreT language in several ways. We would like to support non-uniform priors over program variables (which will require converting the SMT and Sketch back-ends to use max-SMT

solvers). There are several data types that we would like to support, including floating point numbers, strings, and richer data types. Second, we would like to continue to expand the number of back-ends. Natural next steps are back-ends based on local search or Markov Chain Monte Carlo, and on message passing inference in graphical models, perhaps taking inspiration from Sontag et al. (2008). Third, we would like to build higher level languages on top of TerpreT, to support more compact specification of common TerpreT programming patterns.

More generally, we believe the opportunities for IPS come not from improving discrete search in this setting, but in re-phrasing the program synthesis problem to be more of a pattern-matching and big-data problem, and in augmenting the specification beyond just input-output examples (for example, incorporating natural language). In these cases, the importance of the discrete search component decreases, and we believe there to be many opportunities for machine learning. As we move forward in these directions, we believe TerpreT will continue to be valuable, as it makes it easy to build a range of differentiable interpreters to be used in conjunction with larger learning systems.

Acknowledgements

We thank several people for discussions that helped improve this report: Tom Minka for discussions related to the Gates LP relaxation; John Winn for several discussions related to probabilistic programming and gates; Ryota Tomioka for discussions related to the FMGD loss surface; Andy Gordon for pushing us towards the probabilistic programming formulation of TerpreT; Abdel-rahman Mohamed for discussions related to neural networks and program synthesis; Jack Feser for being the first non-author TerpreT user; Aditya Nori for helpful discussions about program synthesis; and Matej Balog for a critical reading of this manuscript.

A Proof of Lemma 1

Lemma 1. *All island structures have zero gradient.*

Proof. Notationally, let $\boldsymbol{s} = \{s_i(a) \mid i \in \{1, \ldots, K\}, a \in \{0, 1\}\}$ be the free parameters, where $s_i(a)$ is the unnormalized log probability that x_i is equal to a. The probability over x_i is then given by a softmax; i.e., $p(x_i = a) = \mu_i(a) = \frac{\exp s_i(a)}{\exp s_i(0) + \exp s_i(1)}$. Let $\boldsymbol{\mu}$ be the set $\{\mu_i(a) \mid i \in \{1, \ldots, K\}, a \in \{0, 1\}\}$. Let the objective $o(\boldsymbol{s})$ be the log probability of the observations, i.e., $o(\boldsymbol{s}) = \sum_i \log p(y_i = 0 \mid \boldsymbol{s}) = \sum_i \log p(y_i = 0 \mid \boldsymbol{\mu})$.

The plan is to show that for each island structure described above, the partial derivative $\frac{\partial o(\boldsymbol{s})}{\partial s_i(a)}$ is 0 for every i and a. This can be done by computing the partial derivatives and showing that they are 0 for each possible local configuration that arises in an island structure.

First, let us derive the gradient contribution from a single observation $y_i = 0$. By the definition of parity and the FMGD model,

$$p(y_i = 0 \mid \boldsymbol{\mu}) = p(x_i = 0)p(x_{i+1} = 0) + p(x_i = 1)p(x_{i+1} = 1) \tag{20}$$

$$= \mu_i(0)\mu_{i+1}(0) + \mu_i(1)\mu_{i+1}(1). \tag{21}$$

The partial derivative $\frac{\partial \log p(y_i=0)}{\partial s_j(a)}$ can be computed via the chain rule as

$$\frac{\partial \log p(y_i = 0)}{\partial s_j(a)} = \frac{\partial \log p(y_i = 0)}{\partial p(y_i = 0)} \frac{\partial p(y_i = 0)}{\partial s_j(b)}. \tag{22}$$

Each of these can be computed straight-forwardly. The partial derivative $\frac{\partial \log p(y_i=0)}{\partial p(y_i=0)}$ is $\frac{1}{p(y_i=0)}$. The partial derivative $\frac{\partial p(y_i=0)}{\partial s_j(b)}$ is as follows:

$$\frac{\partial p(y_i = 0)}{\partial s_j(b)} = \begin{cases} \frac{\partial \mu_i(0)}{\partial s_i(b)} \mu_{i+1}(0) + \frac{\partial \mu_i(1)}{\partial s_i(b)} \mu_{i+1}(1) & \text{if } j = i \\ \mu_i(0) \frac{\partial \mu_{i+1}(0)}{\partial s_{i+1}(b)} + \mu_i(1) \frac{\partial \mu_{i+1}(1)}{\partial s_{i+1}(b)} & \text{if } j = i + 1 \\ 0 & \text{otherwise.} \end{cases} \tag{23}$$

Finally, the partial derivative of the softmax is

$$\frac{\partial \mu_i(b)}{\partial s_i(a)} = \mu_i(b) \frac{\partial \log \mu_i(b)}{\partial s_i(a)} \tag{24}$$

$$= \mu_i(b) \frac{\partial}{\partial s_i(a)} \left[s_i(b) - \log\left(\exp s_i(0) + \exp s_i(1)\right) \right] \tag{25}$$

$$= \mu_i(b) \left(1\{a = b\} - \mu_i(a) \right). \tag{26}$$

Putting these together, we get the full partial derivatives:

$$\frac{\partial \log p(y_i = 0)}{\partial s_j(a)} = \frac{1}{p(y_i = 0)} \cdot \begin{cases} \mu_i(0)\mu_{i+1}(0) \left(1\{a = 0\} - \mu_i(a)\right) + \mu_i(1)\mu_{i+1}(1) \left(1\{a = 1\} - \mu_i(a)\right) & \text{if } j = i \\ \mu_i(0)\mu_{i+1}(0) \left(1\{a = 0\} - \mu_{i+1}(a)\right) + \mu_i(1)\mu_{i+1}(1) \left(1\{a = 1\} - \mu_{i+1}(a)\right) & \text{if } j = i + 1 \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

Each $s_j(a)$ contributes to two terms in the objective: $\log p(y_{j-1} = 0)$ and $\log p(y_j = 0)$. Thus the gradient on $s_j(a)$ is the sum of gradient contributions from these two terms:

$$\frac{\partial o(\boldsymbol{s})}{\partial s_j(a)} = \frac{\partial \log p(y_{j-1} = 0)}{\partial s_j(a)} + \frac{\partial \log p(y_j = 0)}{\partial s_j(a)}. \tag{28}$$

Restricting attention to $a = 0$ (the case where $a = 1$ follows similarly and is omitted), we can simplify further:

$$\frac{\partial o(\boldsymbol{s})}{\partial s_j(0)} = \frac{\partial \log p(y_{j-1} = 0)}{\partial s_j(0)} + \frac{\partial \log p(y_j = 0)}{\partial s_j(0)} \tag{29}$$

$$= \frac{1}{p(y_{j-1} = 0)} \left(\mu_{j-1}(0)\mu_j(0)(1 - \mu_j(0)) + \mu_{j-1}(1)\mu_j(1)(-\mu_j(0)) \right) \tag{30}$$

$$+ \frac{1}{p(y_j = 0)} \left(\mu_j(0)\mu_{j+1}(0) \left(1 - \mu_j(0)\right) + \mu_j(1)\mu_{j+1}(1)(-\mu_j(0)) \right) \tag{31}$$

$$= \frac{1}{p(y_{j-1} = 0)} \left(\mu_{j-1}(0)\mu_j(0)\mu_j(1) - \mu_{j-1}(1)\mu_j(1)\mu_j(0) \right) \tag{32}$$

$$+ \frac{1}{p(y_j = 0)} \left(\mu_j(0)\mu_{j+1}(0)\mu_j(1) - \mu_j(1)\mu_{j+1}(1)\mu_j(0) \right) \tag{33}$$

$$= \mu_j(0)\mu_j(1) \left(\frac{\mu_{j-1}(0) - \mu_{j-1}(1)}{p(y_{j-1} = 0)} + \frac{\mu_{j+1}(0) - \mu_{j+1}(1)}{p(y_j = 0)} \right) \tag{34}$$

The first note is that if $\mu_j(0) = 0$ or $\mu_j(1) = 0$, then the gradient for $s_j(0)$ is 0. Thus, we only need to consider triplets $(\mu_{j-1}(\cdot), \mu_j(\cdot), \mu_{j+1}(\cdot))$ where $\mu_j(a) = .5$. The only two cases that arise in island structures are $(0, .5, 1)$ and $(1, .5, 0)$. Consider the first case, $(0, .5, 1)$:

$$= \mu_j(0)\mu_j(1) \left(\frac{\mu_{j-1}(0) - \mu_{j-1}(1)}{p(y_{j-1} = 0)} + \frac{\mu_{j+1}(0) - \mu_{j+1}(1)}{p(y_j = 0)} \right) \tag{35}$$

$$= .5 \cdot .5 \cdot \left(\frac{1 - 0}{.5} + \frac{0 - 1}{.5} \right) \tag{36}$$

$$= .5 \cdot .5 \cdot (2 - 2) = 0. \tag{37}$$

The second case $(1, .5, 0)$ is similar:

$$= \mu_j(0)\mu_j(1) \left(\frac{\mu_{j-1}(0) - \mu_{j-1}(1)}{p(y_{j-1} = 0)} + \frac{\mu_{j+1}(0) - \mu_{j+1}(1)}{p(y_j = 0)} \right) \tag{38}$$

$$= .5 \cdot .5 \cdot \left(\frac{0 - 1}{.5} + \frac{1 - 0}{.5} \right) \tag{39}$$

$$= .5 \cdot .5 \cdot (-2 + 2) = 0. \tag{40}$$

Thus, the gradients with respect to $s_j(a)$ are zero for all j and a when an island-structured configuration is given. The objective is clearly suboptimal (since some $p(y_i = 0)$ are less than 1 at the island boundaries). Thus, each island structure is a suboptimal local optimum.

□

B Benchmark models

B.1 Turing Machine

```
const_nStateMem  = #__HYPERPARAM_const_nStateMem__
const_nStateHead = #__HYPERPARAM_const_nStateHead__
const_nTimesteps = #__HYPERPARAM_const_nTimesteps__
const_tapeLength = #__HYPERPARAM_const_tapeLength__
const_nDir = 3

const_nInstances = #__HYPERPARAM_const_nInstances__

@CompileMe([const_tapeLength,const_nDir],const_tapeLength)
def move(pos, direction):
    if   direction == 0: return pos
    elif direction == 1: return (pos + 1) % const_tapeLength
    elif direction == 2: return (pos - 1) % const_tapeLength
@CompileMe([const_tapeLength, const_tapeLength], 2)
def equalityTestPos(a,b): return 1 if a == b else 0
@CompileMe([const_nStateHead, const_nStateHead], 2)
def equalityTestState(a,b): return 1 if a == b else 0

#######################################################
#  Source code parametrisation                        #
#######################################################
newValue  = Param(const_nStateMem)[const_nStateHead, const_nStateMem]
direction = Param(const_nDir)[const_nStateHead, const_nStateMem]
newState  = Param(const_nStateHead)[const_nStateHead, const_nStateMem]

#######################################################
#  Interpreter model                                  #
#######################################################
# Memory tape
tape = Var(const_nStateMem)[const_nInstances, const_nTimesteps, const_tapeLength]
# Machine head
curPos   = Var(const_tapeLength)[const_nInstances, const_nTimesteps]
curState = Var(const_nStateHead)[const_nInstances, const_nTimesteps]
isHalted = Var(2)[const_nInstances, const_nTimesteps]
# Temporary values
tmpActiveCell = Var(2)[const_nInstances, const_nTimesteps - 1, const_tapeLength]
tmpMemState   = Var(const_nStateMem)[const_nInstances, const_nTimesteps - 1]

#__IMPORT_OBSERVED_INPUTS__

# Initialize machine head
for n in range(const_nInstances):
    curPos[n,0].set_to_constant(0)
    curState[n,0].set_to_constant(1)
    isHalted[n,0].set_to_constant(0)

# Run the Turing machine
for n in range(const_nInstances):          # loop over I/O examples
    for t in range(const_nTimesteps - 1):  # loop over program timesteps

        # Carry forward unmodified tape and head if halted
```

```
        if isHalted[n,t] == 1:
            for m in range(const_tapeLength):
                tape[n,t+1,m].set_to(tape[n,t,m])
            curState[n,t+1].set_to(curState[n,t])
            curPos[n,t+1].set_to(curPos[n,t])
            isHalted[n,t+1].set_to(isHalted[n,t])

        # Perform Turing update if not halted
        elif isHalted[n,t] == 0:
            with curState[n,t] as s:
                with curPos[n,t] as x:
                    with tape[n,t,x] as Tx:
                        tmpMemState[n,t].set_to(newValue[s,Tx])
                        curPos[n,t+1].set_to(move(x, direction[s,Tx]))
                        curState[n,t+1].set_to(newState[s,Tx])

            # Machine halts if head enters state 0
            isHalted[n,t+1].set_to(equalityTestState(0,curState[n,t+1]))

            # Write temporary value to tape
            for m in range(const_tapeLength):
                tmpActiveCell[n,t,m].set_to(equalityTestPos(m, curPos[n,t]))
                if tmpActiveCell[n,t,m] == 1:
                    tape[n,t+1,m].set_to(tmpMemState[n,t])
                elif tmpActiveCell[n,t,m] == 0:
                    tape[n,t+1,m].set_to(tape[n,t,m])

# Machine must be halted at end of execution
for n in range(const_nInstances):
    isHalted[n,const_nTimesteps - 1].observe_value(1)

#__IMPORT_OBSERVED_OUTPUTS__
```

B.2 Boolean circuits

```
const_nGates = #__HYPERPARAM_const_nGates__
const_nWires = #__HYPERPARAM_const_nWires__
const_nGateTypes = 5

const_nInstances = #__HYPERPARAM_const_nInstances__

@CompileMe([const_two,const_two],const_two)
def AND(a,b): return int(a and b)
@CompileMe([const_two,const_two],const_two)
def OR(a,b): return int(a or b)
@CompileMe([const_two,const_two],const_two)
def XOR(a,b): return int(a ^ b)
@CompileMe([const_two],const_two)
def NOT(a): return int(not a)
@CompileMe([const_two],const_two)
def NOOP(a): return a
@CompileMe([const_nWires,const_nWires],const_two)
def equalityTest(a,b): return 1 if a == b else 0

```

```
20  #####################################################
21  #  Source code parametrisation                      #
22  #####################################################
23  gate = Param(const_nGateTypes)[const_nGates]
24  in1  = Param(const_nWires)[const_nGates]
25  in2  = Param(const_nWires)[const_nGates]
26  out  = Param(const_nWires)[const_nGates]
27  
28  #####################################################
29  #  Interpreter model                                #
30  #####################################################
31  wires = Var(2)[const_nInstances, const_nGates + 1, const_nWires]
32  tmpOutput  = Var(2)[const_nInstances, const_nGates]
33  tmpDoWrite = Var(2)[const_nInstances, const_nGates, const_nWires]
34  tmpArg1 = Var(2)[const_nInstances, const_nGates]
35  tmpArg2 = Var(2)[const_nInstances, const_nGates]
36  
37  #__IMPORT_OBSERVED_INPUTS__
38  
39  # Run the circuit
40  for n in range(const_nInstances):  # loop over I/O examples
41      for g in range(const_nGates):  # loop over sequential gates
42  
43          # Load gate inputs
44          with in1[g] as i1:
45              with in2[g] as i2:
46                  tmpArg1[n,g].set_to(wires[n,g,i1])
47                  tmpArg2[n,g].set_to(wires[n,g,i2])
48  
49          # Compute gate output
50          if gate[g] == 0:
51              tmpOutput[n,g].set_to(  AND(tmpArg1[n,g], tmpArg2[n,g]) )
52          elif gate[g] == 1:
53              tmpOutput[n,g].set_to(   OR(tmpArg1[n,g], tmpArg2[n,g]) )
54          elif gate[g] == 2:
55              tmpOutput[n,g].set_to(  XOR(tmpArg1[n,g], tmpArg2[n,g]) )
56          elif gate[g] == 3:
57              tmpOutput[n,g].set_to(  NOT(tmpArg1[n,g]) )
58          elif gate[g] == 4:
59              tmpOutput[n,g].set_to( NOOP(tmpArg1[n,g]) )
60  
61          # Write gate output
62          for w in range(const_nWires):
63              tmpDoWrite[n,g,w].set_to(equalityTest(out[g], w))
64              if tmpDoWrite[n,g,w] == 1:
65                  wires[n,g + 1,w].set_to(tmpOutput[n,g])
66              elif tmpDoWrite[n,g,w] == 0:
67                  wires[n,g + 1,w].set_to(wires[n,g,w])
68  
69  #__IMPORT_OBSERVED_OUTPUTS__
```

B.3 Basic-block model

```
1  const_nBlocks     = #__HYPERPARAM_const_nBlocks__
```

```
2   const_nRegisters = #__HYPERPARAM_const_nRegisters__
3   const_nTimesteps = #__HYPERPARAM_const_nTimesteps__
4   const_maxInt     = #__HYPERPARAM_const_maxInt__
5   const_nInstructions = 7
6   const_nActions = 2
7   const_nInstrPlusAct = const_nInstructions + const_nActions
8   const_noopIndex = const_nInstructions
9
10  const_nInstances = #__HYPERPARAM_const_nInstances__
11
12  @CompileMe([const_nInstrPlusAct], 2)
13  def isInstruction(a): return 1 if a < const_nInstructions else 0
14  @CompileMe([const_nInstrPlusAct], const_nInstructions)
15  def extractInstruction(a): return a
16  @CompileMe([const_nInstrPlusAct], const_nActions)
17  def extractAction(a): return a - const_nInstructions
18  @CompileMe([const_maxInt, const_maxInt], 2)
19  def equalityTestValue(a,b): return 1 if a == b else 0
20  @CompileMe([const_nRegisters, const_nRegisters], 2)
21  def equalityTestReg(a, b): return 1 if a == b else 0
22  @CompileMe([const_maxInt], 2)
23  def greaterThanZero(a): return 1 if a > 0 else 0
24
25  @CompileMe([], const_maxInt)
26  def ZERO(): return 0
27  @CompileMe([const_maxInt], const_maxInt)
28  def INC(a): return (a + 1) % const_maxInt
29  @CompileMe([const_maxInt], const_maxInt)
30  def DEC(a): return (a - 1) % const_maxInt
31  @CompileMe([const_maxInt, const_maxInt], const_maxInt)
32  def ADD(a, b): return (a + b) % const_maxInt
33  @CompileMe([const_maxInt, const_maxInt], const_maxInt)
34  def SUB(a, b): return (a - b) % const_maxInt
35  @CompileMe([const_maxInt, const_maxInt], const_maxInt)
36  def LESSTHAN(a, b): return 1 if a < b else 0
37
38  #####################################################
39  #  Source code parametrisation                      #
40  #####################################################
41  instructions = Param(const_nInstrPlusAct)[const_nBlocks]
42  arg1s = Param(const_nRegisters)[const_nBlocks]
43  arg2s = Param(const_nRegisters)[const_nBlocks]
44  rOuts = Param(const_nRegisters)[const_nBlocks]
45  thenBlocks = Param(const_nBlocks)[const_nBlocks]
46  elseBlocks = Param(const_nBlocks)[const_nBlocks]
47  rConds = Param(const_nRegisters)[const_nBlocks]
48
49  #####################################################
50  #  Interpreter model                                #
51  #####################################################
52  # Program pointer
53  curBlocks = Var(const_nBlocks)[const_nInstances,const_nTimesteps]
54  # Memory
55  registers = Var(const_maxInt)[const_nInstances,const_nTimesteps,const_nRegisters]
56  heap = Var(const_maxInt)[const_nInstances,const_nTimesteps,const_maxInt]
```

```
57  # Temporary values
58  tmpIsInstr = Var(2)[const_nInstances,const_nTimesteps-1]
59  tmpInstr   = Var(const_nInstructions)[const_nInstances,const_nTimesteps-1]
60  tmpAction  = Var(const_nActions)[const_nInstances,const_nTimesteps-1]
61  tmpArg1Val = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
62  tmpArg2Val = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
63  tmpOutput  = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
64  tmpDoWrite = Var(2)[const_nInstances,const_nTimesteps-1, const_nRegisters]
65  tmpCondVal = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
66  tmpGotoThen  = Var(2)[const_nInstances,const_nTimesteps-1]
67  tmpWriteHeap = Var(2)[const_nInstances,const_nTimesteps-1,const_maxInt]
68
69  # Initialize block 0 to a spining STOP block
70  instructions[0].set_to_constant(const_noopIndex)
71  thenBlocks[0].set_to_constant(0)
72  elseBlocks[0].set_to_constant(0)
73
74  # Initialize the program pointer to block 1 and the registers to 0
75  for n in range(const_nInstances):
76      curBlocks[n,0].set_to_constant(1)
77      for r in range(const_nRegisters):
78          registers[n,0,r].set_to_constant(0)
79
80  #__IMPORT_OBSERVED_INPUTS__
81
82  # Run the program
83  for n in range(const_nInstances):        # loop over I/O examples
84      for t in range(const_nTimesteps-1):  # loop over program timesteps
85          with curBlocks[n,t] as pc:
86
87              # Load block inputs
88              with arg1s[pc] as a1:
89                  tmpArg1Val[n,t].set_to(registers[n,t,a1])
90              with arg2s[pc] as a2:
91                  tmpArg2Val[n,t].set_to(registers[n,t,a2])
92
93              # Determine whether block performs a heap ACTION or register INSTRUCTION
94              tmpIsInstr[n,t].set_to(isInstruction(instructions[pc]))
95
96              # Handle heap ACTIONS
97              if tmpIsInstr[n,t] == 0:
98                  tmpAction[n,t].set_to(extractAction(instructions[pc]))
99
100                 # Actions affect the heap ...
101                 if tmpAction[n,t] == 0:   # NOOP
102                     for m in range(const_maxInt):
103                         heap[n,t+1,m].set_to(heap[n,t,m])
104                 elif tmpAction[n,t] == 1: # WRITE
105                     for m in range(const_maxInt):
106                         tmpWriteHeap[n,t,m].set_to(equalityTestValue(tmpArg1Val[n,t], m))
107                         if tmpWriteHeap[n,t,m] == 1:
108                             heap[n,t+1,m].set_to(tmpArg2Val[n,t])
109                         elif tmpWriteHeap[n,t,m] == 0:
110                             heap[n,t+1,m].set_to(heap[n,t,m])
111
```

```
112                 # ... and do not affect registers
113                 for r in range(const_nRegisters):
114                     registers[n,t+1,r].set_to(registers[n,t,r])
115 
116             # Handle register INSTRUCTIONS
117             elif tmpIsInstr[n,t] == 1:
118                 tmpInstr[n,t].set_to(extractInstruction(instructions[curBlocks[n,t]]))
119 
120                 # Instructions affect registers ...
121                 if tmpInstr[n,t] == 0:
122                     tmpOutput[n,t].set_to( ZERO() )
123                 elif tmpInstr[n,t] == 1:
124                     tmpOutput[n,t].set_to( INC(tmpArg1Val[n,t]) )
125                 elif tmpInstr[n,t] == 2:
126                     tmpOutput[n,t].set_to( DEC(tmpArg1Val[n,t]) )
127                 elif tmpInstr[n,t] == 3:
128                     tmpOutput[n,t].set_to( ADD(tmpArg1Val[n,t],tmpArg2Val[n,t]) )
129                 elif tmpInstr[n,t] == 4:
130                     tmpOutput[n,t].set_to( SUB(tmpArg1Val[n,t],tmpArg2Val[n,t]) )
131                 elif tmpInstr[n,t] == 5:
132                     tmpOutput[n,t].set_to( LESSTHAN(tmpArg1Val[n,t],tmpArg2Val[n,t])
133                 elif tmpInstr[n,t] == 6: # READ
134                     with tmpArg1Val[n,t] as a1:
135                         tmpOutput[n,t].set_to(heap[n,t,a1])
136 
137                 for r in range(const_nRegisters):
138                     tmpDoWrite[n,t,r].set_to(equalityTestReg(rOuts[pc], r))
139                     if tmpDoWrite[n,t,r] == 1:
140                         registers[n,t+1,r].set_to(tmpOutput[n,t])
141                     elif tmpDoWrite[n,t,r] == 0:
142                         registers[n,t+1,r].set_to(registers[n,t,r])
143 
144                 # ... and do not affect the heap
145                 for m in range(const_maxInt):
146                     heap[n,t+1,m].set_to(heap[n,t,m])
147 
148             # Perform branching according to condition register
149             with rConds[pc] as rc:
150                 tmpCondVal[n,t].set_to(registers[n,t+1,rc])
151 
152             tmpGotoThen[n,t].set_to(greaterThanZero(tmpCondVal[n,t]))
153             if tmpGotoThen[n,t] == 1:
154                 curBlocks[n,t+1].set_to(thenBlocks[pc])
155             elif tmpGotoThen[n,t] == 0:
156                 curBlocks[n,t+1].set_to(elseBlocks[pc])
157 
158 # Program must terminate in the STOP block
159 for n in range(const_nInstances):
160     curBlocks[n,const_nTimesteps - 1].observe_value(0)
161 #__IMPORT_OBSERVED_OUTPUTS__
```

B.4 Assembly Model

```
1 const_nLines      = #__HYPERPARAM_const_nBlocks__
```

```
2  const_nRegisters = #__HYPERPARAM_const_nRegisters__
3  const_nTimesteps = #__HYPERPARAM_const_nTimesteps__
4  const_maxInt     = #__HYPERPARAM_const_maxInt__
5  const_nInstructions = 7
6  const_nActions = 1
7  const_nBranches = 2
8  const_nInstrActBranch = const_nInstructions + const_nActions + const_nBranches
9
10 const_nInstances = #__HYPERPARAM_const_nInstances__
11
12 @CompileMe([const_nInstrActBranch], 3)
13 def instructionType(a):
14     if a < const_nInstructions:
15         return 0
16     elif a < (const_nInstructions + const_nActions):
17         return 1
18     else:
19         return 2
20 @CompileMe([const_nInstrActBranch], const_nInstructions)
21 def extractInstruction(a): return a
22 @CompileMe([const_nInstrActBranch], const_nBranches)
23 def extractBranch(a): return a - const_nInstructions - const_nActions
24 @CompileMe([const_nRegisters, const_nRegisters], 2)
25 def equalityTestReg(a,b): return 1 if a == b else 0
26 @CompileMe([const_maxInt, const_maxInt], 2)
27 def equalityTestValue(a,b): return 1 if a == b else 0
28 @CompileMe([const_nLines, const_nLines], 2)
29 def equalityTestLine(a,b): return 1 if a == b else 0
30 @CompileMe([const_maxInt], 2)
31 def valueEqualsZero(a): return 1 if a == 0 else 0
32 @CompileMe([const_nLines], const_nLines)
33 def incLine(a): return (a + 1) % const_nLines
34
35 @CompileMe([], const_maxInt)
36 def ZERO(): return 0
37 @CompileMe([const_maxInt], const_maxInt)
38 def INC(a): return (a + 1) % const_maxInt
39 @CompileMe([const_maxInt, const_maxInt], const_maxInt)
40 def ADD(a, b): return (a + b) % const_maxInt
41 @CompileMe([const_maxInt, const_maxInt], const_maxInt)
42 def SUB(a, b): return (a - b) % const_maxInt
43 @CompileMe([const_maxInt], const_maxInt)
44 def DEC(a): return (a - 1) % const_maxInt
45 @CompileMe([const_maxInt, const_maxInt], const_maxInt)
46 def LESSTHAN(a,b): return 1 if a < b else 0
47
48 #######################################################
49 #  Source code parametrisation                        #
50 #######################################################
51 instructions = Param(const_nInstrActBranch)[const_nLines]
52 branchAddr   = Param(const_nLines)[const_nLines]
53 arg1s = Param(const_nRegisters)[const_nLines]
54 arg2s = Param(const_nRegisters)[const_nLines]
55 rOuts = Param(const_nRegisters)[const_nLines]
56
```

```
57  ######################################################
58  #  Interpreter model                                 #
59  ######################################################
60  # Program pointer
61  curLine = Var(const_nLines)[const_nInstances,const_nTimesteps]
62  # Memory
63  registers = Var(const_maxInt)[const_nInstances,const_nTimesteps,const_nRegisters]
64  heap = Var(const_maxInt)[const_nInstances,const_nTimesteps,const_maxInt]
65  # Temporary values
66  tmpInstrActBranch = Var(3)[const_nInstances, const_nTimesteps-1]
67  tmpInstr   = Var(const_nInstructions)[const_nInstances,const_nTimesteps-1]
68  tmpBranch  = Var(const_nBranches)[const_nInstances,const_nTimesteps-1]
69  tmpArg1Val = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
70  tmpArg2Val = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
71  tmpOutput  = Var(const_maxInt)[const_nInstances,const_nTimesteps-1]
72  tmpDoWrite = Var(2)[const_nInstances,const_nTimesteps-1, const_nRegisters]
73  tmpBranchIsZero = Var(2)[const_nInstances,const_nTimesteps-1]
74  tmpWriteHeap = Var(2)[const_nInstances,const_nTimesteps-1,const_maxInt]
75  isHalted = Var(2)[const_nInstances, const_nTimesteps - 1]
76
77  # Initialize the program pointer to block 1 and the registers to 0
78  for n in range(const_nInstances):
79      curLine[n,0].set_to_constant(1)
80      for r in range(const_nRegisters):
81          registers[n,0,r].set_to_constant(0)
82
83  #__IMPORT_OBSERVED_INPUTS__
84
85  # Run the program
86  for n in range(const_nInstances):        # loop over I/O examples
87      for t in range(const_nTimesteps-1):  # loop over program timesteps
88          # Halt if we jump to line 0
89          isHalted[n,t].set_to(equalityTestLine(curLine[n,t],0))
90
91          # If not halted, execute current line
92          if isHalted[n,t] == 0:
93              with curLine[n,t] as pc:
94                  # Load line inputs
95                  with arg1s[pc] as a1:
96                      tmpArg1Val[n,t].set_to(registers[n,t,a1])
97                  with arg2s[pc] as a2:
98                      tmpArg2Val[n,t].set_to(registers[n,t,a2])
99
100                 # Determine whether line performs a register INSTRUCTION, a heap ACTION or
101                 # a control flow BRANCH
102                 tmpInstrActBranch[n,t].set_to(instructionType(instructions[pc]))
103
104                 # Handle register INSTRUCTIONS
105                 if tmpInstrActBranch[n,t] == 0:
106                     tmpInstr[n,t].set_to(extractInstruction(instructions[pc]))
107
108                     # Instructions affect registers ...
109                     if tmpInstr[n,t] == 0:
110                         tmpOutput[n,t].set_to( ZERO() )
111                     elif tmpInstr[n,t] == 1:
```

```
112                         tmpOutput[n,t].set_to( INC(tmpArg1Val[n,t]) )
113                     elif tmpInstr[n,t] == 2:
114                         tmpOutput[n,t].set_to( ADD(tmpArg1Val[n,t],tmpArg2Val[n,t]) )
115                     elif tmpInstr[n,t] == 3:
116                         tmpOutput[n,t].set_to( SUB(tmpArg1Val[n,t],tmpArg2Val[n,t]) )
117                     elif tmpInstr[n,t] == 4:
118                         tmpOutput[n,t].set_to( DEC(tmpArg1Val[n,t]) )
119                     elif tmpInstr[n,t] == 5:
120                         tmpOutput[n,t].set_to( LESSTHAN(tmpArg1Val[n,t],tmpArg2Val[n,t]) )
121                     elif tmpInstr[n,t] == 6:
122                         with tmpArg1Val[n,t]:
123                             tmpOutput[n,t].set_to(heap[n,t,tmpArg1Val[n,t]])
124
125                     for r in range(const_nRegisters):
126                         tmpDoWrite[n,t,r].set_to(equalityTestReg(rOuts[pc], r))
127                         if tmpDoWrite[n,t,r] == 1:
128                             registers[n,t+1,r].set_to(tmpOutput[n,t])
129                         elif tmpDoWrite[n,t,r] == 0:
130                             registers[n,t+1,r].set_to(registers[n,t,r])
131
132                     # ... and do not affect the heap
133                     for m in range(const_maxInt):
134                         heap[n,t+1,m].set_to(heap[n,t,m])
135
136                     # Progress to the next line
137                     curLine[n,t+1].set_to(incLine(pc))
138
139                 # Handle heap ACTIONS
140                 elif tmpInstrActBranch[n,t] == 1:
141                     # The only action is to write to the heap
142                     for m in range(const_maxInt):
143                         tmpWriteHeap[n,t,m].set_to(equalityTestValue(tmpArg1Val[n,t],m))
144                         if tmpWriteHeap[n,t,m] == 1:
145                             heap[n,t+1,m].set_to(tmpArg2Val[n,t])
146                         elif tmpWriteHeap[n,t,m] == 0:
147                             heap[n,t+1,m].set_to(heap[n,t,m])
148
149                     # Actions do not affect the registers
150                     for r in range(const_nRegisters):
151                         registers[n,t+1,r].set_to(registers[n,t,r])
152
153                     # Progress to the next line
154                     curLine[n,t+1].set_to(incLine(pc))
155
156                 # Handle control flow BRANCHES
157                 elif tmpInstrActBranch[n,t] == 2: # Branch
158                     tmpBranch[n,t].set_to(extractBranch(instructions[pc]))
159                     tmpBranchIsZero[n,t].set_to(valueEqualsZero(tmpArg1Val[n,t]))
160
161                     # BRANCHES affect the program counter ...
162                     if tmpBranch[n,t] == 0:   # JZ
163                         if tmpBranchIsZero[n,t] == 1:
164                             curLine[n,t+1].set_to(branchAddr[pc])
165                         elif tmpBranchIsZero[n,t] == 0:
166                             curLine[n,t+1].set_to(incLine(pc))
```

```
167                     elif tmpBranch[n,t] == 1: # JNZ
168                         if tmpBranchIsZero[n,t] == 0:
169                             curLine[n,t+1].set_to(branchAddr[pc])
170                         elif tmpBranchIsZero[n,t] == 1:
171                             curLine[n,t+1].set_to(incLine(pc))
172 
173                     # ... and do not affect the registers and heap.
174                     for r in range(const_nRegisters):
175                         registers[n,t+1,r].set_to(registers[n,t,r])
176                     for m in range(const_maxInt):
177                         heap[n,t+1,m].set_to(heap[n,t,m])
178 
179         # Carry forward unmodified registers and heap if halted
180         elif isHalted[n,t] == 1:
181             for r in range(const_nRegisters):
182                 registers[n,t+1,r].set_to(registers[n,t,r])
183             for m in range(const_maxInt):
184                 heap[n,t+1,m].set_to(heap[n,t,m])
185             curLine[n,t+1].set_to(curLine[n,t])
186 
187 # Program must terminate in the STOP line
188 for n in range(const_nInstances):
189     curLine[n,const_nTimesteps-1].observe_value(0)
190 #__IMPORT_OBSERVED_OUTPUTS__
```

References

Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.5. Technical report, The University of Iowa, 2015. Available at `http://smt-lib.org/`.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Alan W Biermann. The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Adaptive neural compilation. *CoRR*, abs/1605.07969, 2016. URL `http://arxiv.org/abs/1605.07969`.

Bob Carpenter. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2015.

KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014. URL `http://arxiv.org/abs/1409.1259`.

Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

Sreerupa Das, C. Lee Giles, and Guo-Zheng Sun. Using prior knowledge in a {NNPDA} to learn context-free languages. In *Proceedings of the 5th Conference on Advances in Neural Information Processing Systems, NIPS 1992*, pages 65–72, 1992.

Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *14th Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, pages 337–340, 2008.

Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems NIPS*, pages 973–981, 2015.

C. Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. Higher order recurrent networks and grammatical inference. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 380–387, 1989.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence (UAI)*, 2008.

Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL `http://arxiv.org/abs/1308.0850`.

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

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

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

Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *ACM SIGPLAN Notices*, volume 42, pages 277–289. ACM, 2007.

Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 190–198, 2015.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *Proceedings of the 4th International Conference on Learning Representations.*, 2016.

Jan Koutník, Klaus Greff, Faustino J. Gomez, and Jürgen Schmidhuber. A clockwork RNN. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014*, pages 1863–1871, 2014.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2015. URL `http://arxiv.org/abs/1511.06392`.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

Tomas Mikolov, Armand Joulin, Sumit Chopra, Michaël Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. In *Proceedings of the 3rd International Conference on Learning Representations 2015*, 2015.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Tom Minka and John Winn. Gates. In *Advances in Neural Information Processing Systems*, pages 1073–1080, 2009.

Michael Mozer and Sreerupa Das. A connectionist symbol manipulator that discovers the structure of context-free languages. In *Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]*, pages 863–870, 1992.

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

Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. In *Proceedings of the International Conference on Learning Representations 2015*, 2016b.

Yura Perov and Frank Wood. Automatic sampler discovery via probabilistic programming and approximate bayesian computation. In *International Conference on Artificial General Intelligence*, pages 262–273. Springer, 2016.

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodík. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *PLDI*, page 42, 2014.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *OOPSLA*, pages 107–126, 2015.

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In *POPL*, pages 761–774, 2016.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. 2016.

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV*, pages 198–216, 2015.

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

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316, 2013.

MI Schlesinger. Syntactic analysis of two-dimensional visual signals in the presence of noise. *Cybernetics and systems analysis*, 12(4):612–628, 1976.

Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

David Sontag, Talya Meltzer, Amir Globerson, Tommi S Jaakkola, and Yair Weiss. Tightening lp relaxations for map using message passing. In *Uncertainty in Artificial Intelligence (UAI)*, 2008.

Stan Development Team. Stan: A c++ library for probability and sampling, version 2.10.0, 2015. URL `http://mc-stan.org/`.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2440–2448, 2015.

Phillip D Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.

Martin J Wainwright and Michael I Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 2008.

Tomas Werner. A linear programming approach to max-sum problem: A review. *IEEE transactions on pattern analysis and machine intelligence*, 29(7):1165–1179, 2007.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *Proceedings of the 3rd International Conference on Learning Representations 2015*, 2014. URL `http://arxiv.org/abs/1410.3916`.

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