2nd Symposium on Advances in Approximate Bayesian Inference, 2019 1–9

Graph Tracking in Dynamic Probabilistic Programs via Source Transformations

Philipp Gabler[1], **Martin Trapp**[1], **Hong Ge**[2], **Franz Pernkopf**[1]
PGABLER@STUDENT.TUGRAZ.AT, MARTIN.TRAPP@TUGRAZ.AT, HG344@CAM.AC.UK,
PERNKOPF@TUGRAZ.AT
[1] *Graz University of Technology,* [2] *University of Cambridge*

1. Introduction

Many machine learning methods acting on graph structures can be expressed in terms of message passing (Koller and Friedman, 2009; Ruozzi, 2011). Among them are variational methods for approximate Bayesian inference (Jordan et al., 1999), automatic differentiation (AD) (Baydin et al., 2018), and backpropagation (Dauwels et al., 2005). Concrete implementations of any message passing algorithm naturally require a data structure representing the computation graph. Therefore, it is necessary to either explicitly define this graph or automatically extract it from the model program under question.

A dominant realisation of message passing in modern machine learning is AD. The two common approaches to extract computation graphs for AD are *operator overloading* and *source transformation* (Griewank and Walther, 2008; van Merriënboer et al., 2018). Operator overloading works by extending existing numerical operations to additionally track the executed expressions at runtime on so-called tapes or *Wengert lists* (Bartholomew-Biggs et al., 2000). It is dynamic, in the sense that a new tape is recorded for every execution. However, being implemented on a library level, it usually requires the programmer to use non-native constructs instead of language primitives, leading to cognitive overhead. Further, there are additional runtime costs due to the separate interpretation of derivatives stored on the tape. Source transformation, on the other hand, operates on the syntactic structure of the whole program, during or before compilation. Unlike in operator overloading, it is possible to inspect and exploit control structures directly. This can lead to more efficient results, compared to operator overloading, since the transformation is done only once per program and eligible for compiler optimisations. But in this approach, no records of the actual execution paths are constructed explicitly – only static information is used at compile time, and cannot be accessed for further analysis or transformation. We refer to Baydin et al. (2018) for a survey on AD methods.

There exists a variety of domains in which the execution path of model programs can change at each run. Examples of this are models with non-uniform data, such as parse trees (Socher et al., 2011) or molecular graphs (Bianucci et al., 2000), Bayesian nonparametric models (Hjort et al., 2010), or simply the occurrence of stochastic control flow in any probabilistic model. Such models are called *dynamic models*. The lack of an explicit graph structure makes it impossible, or at least diffult, to apply source transformation approaches for general message passing algorithms on them. Operator overloading is the more direct way for supporting them, since it automatically records a new tape for each input. In fact, many state-of-the-art machine learning libraries support dynamic graphs based on operator

© P. Gabler, M. Trapp, H. Ge & F. Pernkopf.

overloading, e.g. DyNet (Neubig et al., 2017), Chainer (Tokui et al., 2015), and PyTorch (Paszke et al., 2017). TensorFlow Fold (Looks et al., 2017) follows a more elaborate approach and provides "dynamic batching" operators to define static graphs emulating dynamic operations. However, relying on operator overloading makes it impossible to take advantage of the benefits of source transformations, such as i) utilizing information about the control flow, ii) optimizations at compile time and iii) exploiting the model structure for efficient posterior inference, e.g. detection of conjugacy relationships in models (Hoffman et al., 2018).

In this paper, we present automatic extraction of computation graphs suitable for message-passing algorithms on both static and dynamic models, using source transformation instead of operator overloading.[1] Inspired by recent work on differential programming (Innes, 2018a), our approach transforms the intermediate representation, used by the Julia programming language (Bezanson et al., 2017), of any Julia program. This system can be used to dynamically track computation graphs of machine learning models implemented in existing deep learning libraries, e.g. Yuret (2016) and Innes (2018b), and probabilistic programming systems, e.g. Ge et al. (2018), without having to explicitely declare graph structures. The transformation is implemented as a custom part of the compilation process. Its result is passed on to the compiler, where it can be optimised further. At run time, both data and control path are tracked alongside the original calculations, in the form of a so-called *extended Wengert list*. Such a list holds sub-lists of all functions called during execution, enriched by recorded control flow decisions and meta information that can be used to analyse the execution. Thus, the system combines a source transformation with a tape-based runtime approach. This allows us to exploit the model structure for applications such as automatic conjugacy detection.

Among probabilistic programming systems, the automatic extraction of extended Wengert lists is most comparable to recording "computation traces" in Church (Goodman et al., 2012). The latter represent a directed graph of the function evaluations and intermediate values in a whole program run, but can only be used for inference in the pure, LISP-like programs the language allows. Similarly, "probabilistic execution traces" in Venture (Mansinghka et al., 2014) are automatically constructed dynamic graph structures representating relationships between random variables. Both kinds of structures are then analysed and modified during inference. Accordingly, both are specialized only for probabilistic programming, while extended Wengert lists are a mechanism to track all execution information in general programs, including mutabilitily and imperative control structures. They can therefore be applied to other tasks as well, including automatic differentiation. Other systems use variants of explicitely defined graphs, to support compilation to static factor graphs in Infer.NET (Minka et al., 2018), runtime message-passing in Pyro (Bingham et al., 2018), or building up continuations in WebPPL and Anglican (Goodman and Stuhlmüller, 2014; Wood et al., 2015).

2. Dynamic Graph Tracking

Julia is a dynamic, JIT-compiled programming language with a strong and expressive type system, using multiple dispatch as its main means of abstraction. The compiler's intermediate representation (IR) resembles a control flow graph in a single static assignment (SSA) form (Muchnick, 1997). It consists of blocks, each containing a list of SSA definitions, followed

1. An implementation of our method in the form of a Julia package is available at `https://github.com/phipsgabler/DynamicComputationGraphs.jl`.

by at least one branching instruction. SSA definitions are unique immutable assignments of variables to the value of a non-nested expression. This SSA form representation can already be considered a natural generalization of Wengert lists (Innes, 2018a).

To track dynamic computation graphs, we perform a transformation of the IR implemented as a so-called *generated function*, which is Julia's mechanism for staged programming (Rompf and Odersky, 2010; Bolewski, 2015). Such generated functions, instead of being directly translated into machine code, emit new IR to the compiler, which is then compiled. Our transformation extends its input with additional statements to record the operations and jumps executed at runtime as nodes on an extended Wengert list, together with relevant metadata. The resulting IR consists of about three to five times as many statements as the original. Note that the transformation, due to JIT compilation, is done at most once per method and then stored as compiled code. However, the tracking happens at every execution during runtime.

Figure 1 illustrates the extended Wengert list for one run of a short stochastic function `geom` (for readability, it is expanded to only three levels). The `geom` function draws a sample from the geometric distribution with parameter `beta`, starting to count at value `n`. On the left, we have its IR in textual form, consisting of two blocks. The central part is the graph of nested nodes. There, values and jumps from the top-level call are recorded in their encountered order, as nodes with "tape references" `@1` to `@9`. SSA variables ($\%i$) occurring in expressions of SSA definitions are also replaced in the nodes by the respective tape references. Each node is linked to the original IR statement it records, as indicated by the red arrows.

In the highlighted part, we see the node corresponding to the statement `%7 = geom(%6, %3)`. It is recorded at reference `@8` with expression `geom(@7, @3)` and value `3`. The values of the arguments of this call can be inspected by following the respective references, indicated by the solid blue arrows to nodes `@7` and `@3`. Since `geom` is not a primitive function, the node holds tape of child nodes as well. In this case, it is equivalent to the top level, due to the recursivity of `geom`. We can see the three arguments `@1`, `@2`, and `@3`, corresponding to the block arguments `%1`, `%2`, and `%3`, with the value of `@2` being now `2` instead of `1`. Further we can see function calls of `rand` and `<` as well as a conditional jump, corresponding to the branch the original IR, followed by calls of `+` and `geom`. Going back the tape references from the result value `@9`, the data path of the trace, recreated on the lower right, can be extracted. Note that the data path can be used for reverse-mode AD, and only these nodes would be recorded in a conventional Wengert list. In our system, however, we also record the nodes on the control path, consisting of `@6` and the nodes it depends on.

Recording an extended Wengert list requires to record all block arguments, SSA definitions, and taken branches, with their actual values and metadata. This is achieved by extending the IR with new statements creating nodes and recording them on a helper data structure. Care needs to be taken to properly record function calls, since we need to ensure that non-primitive functions are recursively tracked. As a special case, all return branches are converted to unconditional jumps to one new block at the end, which contains a single unified return statement. This way, they can be treated in the same way as other branches. Please see the appendix for a pseudo-code specification of the IR transformation in Algorithm 1, and the transformed code for the `geom` function in Figure 3.

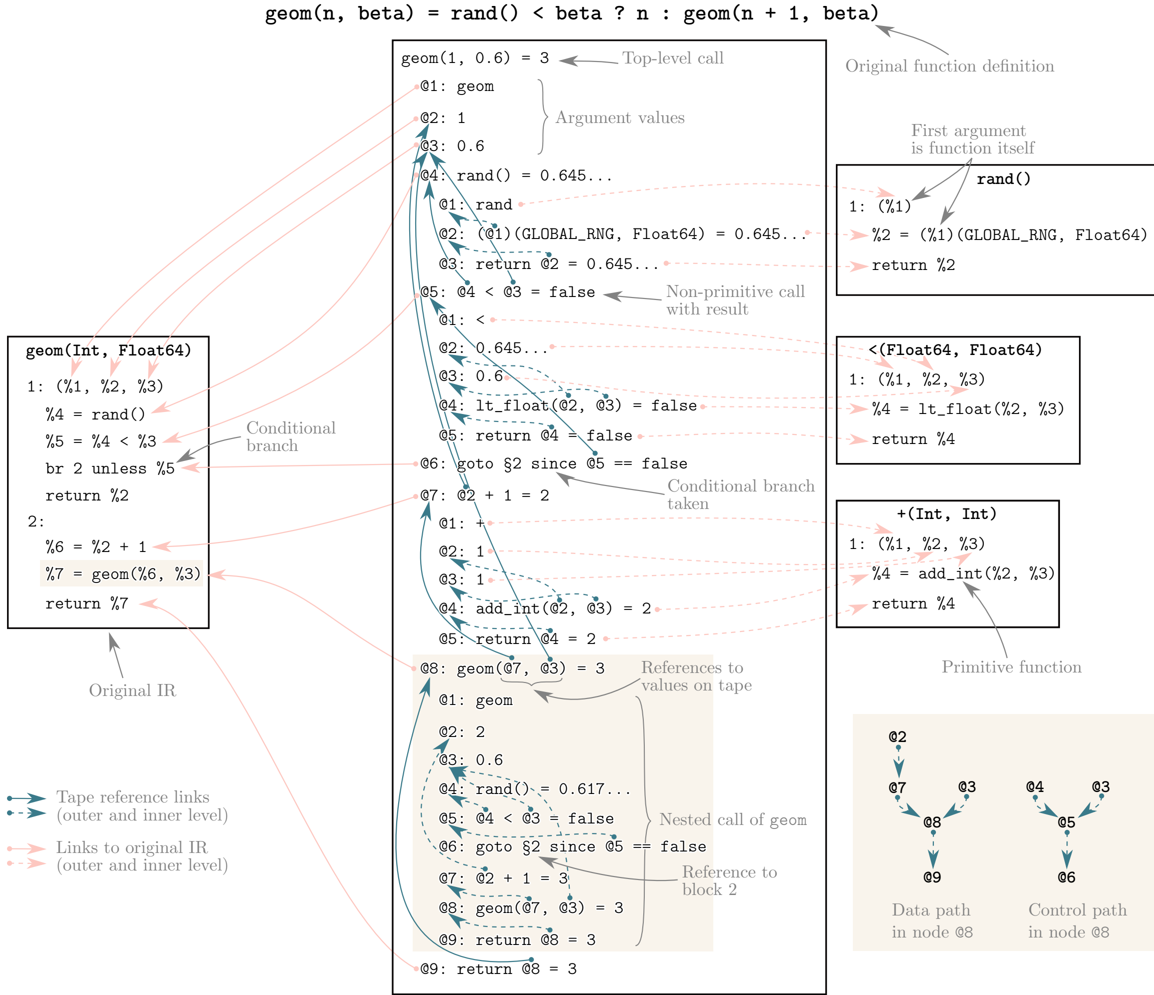

Figure 1: Extended Wengert list for one run of the stochastic function `geom` (only three levels shown). The central box is the tracked graph of the call `geom(1, 0.6)`. The other boxes show the original IR of the called non-primitive functions, to which the nodes are linked. The highlighted part is described in detail in the text.

3. Applications

Our system can be combined with universal probabilistic programming libraries, such as Turing.jl (Ge et al., 2018), to perform efficient inference in dynamic probabilistic models. In particular, once we extracted the extended Wengert list, it is possible to utilise message-passing algorithms such as expectation propagation (Minka, 2001) and variational message passing (Winn and Bishop, 2005) for approximate posterior inference in a dynamic model. Further, we can apply the extracted graphs to obtain and cache conditionals for efficient Gibbs sampling as in WinBUGS (Lunn et al., 2000) and JAGS (Plummer, 2003). Another important avenue is the automatic detection and exploitation of conjugacy relationships. It has been shown that conjugacy exploitation can lead to more efficient posterior inference

$$C \sim \mathcal{B}ernoulli(p)\,, \qquad p \sim \mathcal{B}eta(\alpha, \beta)$$

$$\mu_0 \sim \mathcal{N}ormal(M + 2, \sigma_0^2)\,, \qquad M \sim \begin{cases} \mathcal{N}ormal(m, s), & \text{if } C = 1 \\ \mathcal{G}amma(a, b), & \text{otherwise} \end{cases},$$

$$X_i \overset{iid}{\sim} \mathcal{N}ormal(\mu_0, \sigma_0^2)$$

Figure 2: Generative model for data X_i with dynamic, long-range conjugacy in the priors.

for continuous models (Hoffman et al., 2018) and, in case of particle-based inference, for universal probabilistic programs (Wigren et al., 2019).

We want to emphasise that, in contrast to simple source transformations of conjugate priors, our approach allows us to account for long-range relationships by tracing trough arithmetic operations and function calls, cf. the model in Figure 2. The illustrated model combines simple conjugacy relationships, e.g. $\mathcal{B}eta$ being a conjugate prior of $\mathcal{B}ernoulli$, with more complex scenarios. In particular, we see that the conjugacy relationship of μ_0 and M depends on the value of C, thus requiring any conjugacy detection algorithm to handle stochastic control flow. Further note that we additionally need to be able to correctly handle transformations of stable distributions, as illustrated by $M + 2$, to successfully detect the existing conjugacy. All of those challenges, dynamic models and tracing through transformations of stable distributions, can be overcome by extending existing literature on conjugacy detection to extended Wengert lists.

4. Future Work

The implementation we work on is not yet completely finished, and currently undergoing an internal redesign. As soon as it is final, we plan to implement a range of test applications. Among those will be backward-mode AD for a probabilistic model using Hamiltonian Monte Carlo methods for inference, and and a system to extract and analyze dependency graphs of random variables, including conjugacy detection. These test cases will then be compared against other existing implementation for comparable tasks with respect to usability and performance.

To provide a more flexible implementation of our approach, we are working a context system, allowing to use Julia's dispatch mechanism to specialise the tracking behaviour for different use cases. Further, we will optimise the implementation for efficiency by inspecting the compiler optimisations on the emitted code and tune the transformation to maximally profit from them.

Acknowledgements

This work was supported by the Austrian Science Fund (FWF) under the project number I2706-N31.

References

Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, December 2000. doi: 10.1016/S0377-0427(00)00422-2.

Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. doi: 10.1137/141000671.

Anna Maria Bianucci, Alessio Micheli, Alessandro Sperduti, and Antonina Starita. Application of Cascade Correlation Networks for Structures to Chemistry. *Applied Intelligence*, 12(1):117–147, 2000. doi: 10.1023/A:1008368105614.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *arXiv:1810.09538 [cs, stat]*, October 2018.

Jake Bolewski. Staged programming in Julia. Presentation, JuliaCon 2015, Boston, August 2015.

Justin Dauwels, Sascha Korl, and Hans-Andrea Loeliger. Steepest descent as message passing. In *IEEE Information Theory Workshop*, Rotorua, 2005. doi: 10.1109/ITW.2005.1531853.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690, March 2018.

Noah D. Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014.

Noah D. Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. *arXiv:1206.3255 [cs]*, June 2012.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, 2nd edition, 2008.

Nils Lid Hjort, Chris Holmes, Peter Müller, and Stephen G. Walker. *Bayesian Nonparametrics*. Number 28 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge, 2010.

Matthew D. Hoffman, Matthew J. Johnson, and Dustin Tran. Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language. *arXiv:1811.11926 [cs, stat]*, November 2018.

Michael J. Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *arXiv:1810.07951 [cs]*, October 2018a.

Mike Innes. Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, 2018b. doi: 10.21105/joss.00602.

Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999. doi: 10.1023/A:1007665907178.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, 2009.

Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep Learning with Dynamic Computation Graphs. *arXiv:1702.02181 [cs, stat]*, February 2017.

David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS - A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000. doi: 10.1023/A:1008929526011.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv:1404.0099 [cs, stat]*, March 2014.

Thomas P. Minka. Expectation Propagation for Approximate Bayesian Inference. In *UAI'01: Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, Seattle, 2001.

Tom Minka, John Winn, John Guiver, Yordan Zaykov, Dany Fabian, and John Bronskill. Infer.NET 0.3. http://dotnet.github.io/infer, 2018. Microsoft Research Cambridge.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. DyNet: The Dynamic Neural Network Toolkit. *arXiv:1701.03980 [cs, stat]*, January 2017.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Workshop Autodiff*, 2017.

Martyn Plummer. JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, 2003.

Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *ACM SIGPLAN Notices*, 46, October 2010. doi: 10.1145/1868294.1868314.

Nicholas Robert Ruozzi. *Message Passing Algorithms for Optimization.* Dissertation, Yale University, 2011.

Richard Socher, Cliff Chiung-Yu Lin, Andrew Y. Ng, and Christopher D. Manning. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 129–136, USA, 2011. Omnipress.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: A Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Twenty-Ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

Bart van Merriënboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. Automatic differentiation in ML: Where we are and where we should be going. *arXiv:1810.11530 [cs, stat]*, October 2018.

Anna Wigren, Riccardo Sven Risuleo, Lawrence Murray, and Fredrik Lindsten. Parameter elimination in particle Gibbs sampling. In *Advances in Neural Information Processing Systems*, 2019.

John Winn and Christopher M. Bishop. Variational Message Passing. *Journal of Machine Learning Research*, 6:661–694, 2005.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A New Approach to Probabilistic Programming Inference. *arXiv:1507.00996 [cs, stat]*, July 2015.

Deniz Yuret. Knet: Beginning deep learning with 100 lines of Julia. In *Machine Learning Systems Workshop at NIPS*, 2016.

Appendix

Algorithm 1: IR transformation to record an extended Wengert list (simplified)

This transformation happens inside a generated function called by `trackcall`, which assembles the resulting value and IR into a new node with the correct metadata.

Missing from the description are the recording of metadata, the exact constructions of nodes, and the mechanisms to correctly rename SSA variables during the transformation and tape references at runtime.

```
input  : Original ir object of the tracked function
output : Transformed new_ir
initialize emtpy IR new_ir;
for old_block in blocks(ir) do
    add block new_block to new_ir;
    if old_block is the first block then
        add statement %recorder to new_block, setting up a recorder data structure;
    for arg in arguments(old_block) do
        // copy all arguments
        add argument to new_block;
    if there are branches to old_block then
        // record branches jumping to this block after they happened
        add argument %branch_node to new_block;
        add statement to new_block, recording %branch_node on %recorder;
    for arg in arguments(old_block) do
        // record argument values
        add statement %node to new_block, creating a node for arg;
        add statement to new_block, recording %node on %recorder;
    for stmt in statements(old_block) do
        if stmt is a normal call then
            add statement %call_node to new_block, calling trackcall on stmt;
            add statement to new_block, recording %call_node on the %recorder;
        else if stmt is a "special" call or constant then
            add statement %node to new_block, creating a node for stmt;
            add statement to new_block, recording %node on the %recorder;
    for branch in branches(old_block) do
        if branch is a return branch then
            // substitute return by a branch to the ''return block''
            add statement %return_node to new_block, creating a return node corresponding to branch;
            add branch to new_block, targeting the return block, copying branch's argument, with
              %return_node as extra argument;
        else
            add statement %branch_node to new_block, creating branch node for branch;
            add branch to new_block, copying branch, with %branch_node as extra argument;
begin set up "return block"
    add block return_block to new_ir;
    add arguments %return_value, %return_node to return_block;
    add statement to return_block, recording %return_node on the %recorder;
    add statement %result to return_block, creating a tuple of %return_value and the %recorder;
    add return to return_block, returning %result;
```

```
1: (%4, %1, %2, %3)
  %5 = GraphRecorder(<original IR>, %4)
  %6 = TapeConstant(%1)
  %11 = record!(%5, ArgumentNode(%6, <metadata>))
  %12 = TapeConstant(%2)
  %17 = record!(%5, ArgumentNode(%12, <metadata>))
  %18 = TapeConstant(%3)
  %23 = record!(%5, ArgumentNode(%18, <metadata>))
  %25 = TapeConstant(rand)
  %26 = tuple()
  %27 = getindex(TapeValue)
  %31 = trackcall(<ctx>, rand, %25, %26, %27, <metadata>)
  %32 = record!(%5, %31)
  %34 = TapeConstant(:<)
  %35 = tuple(%32, %3)
  %36 = tapeify(%5, :(%4))
  %37 = tapeify(%5, :(%3))
  %38 = getindex(TapeValue, %36, %37)
  %42 = trackcall(<ctx>, :<, %34, %35, %38, <metadata>)
  %43 = record!(%5, %42)
  %44 = getindex(TapeValue)
  %45 = tapeify(%5, :(%5))
  %49 = JumpNode(2, %44, %45, <metadata>)
  %50 = tapeify(%5, :(%2))
  %54 = ReturnNode(%50, <metadata>)
  br 2 (%49) unless %43
  br 3 (%2, %54)
2: (%55)
  %56 = record!(%5, %55)
  %58 = TapeConstant(:+)
  %59 = tuple(%2, 1)
  %60 = tapeify(%5, :(%2))
  %61 = TapeConstant(1)
  %62 = getindex(TapeValue, %60, %61)
  %66 = trackcall(<ctx>, :+, %58, %59, %62, <metadata>)
  %67 = record!(%5, %66)
  %69 = TapeConstant(geom)
  %70 = tuple(%67, %3)
  %71 = tapeify(%5, :(%6))
  %72 = tapeify(%5, :(%3))
  %73 = getindex(TapeValue, %71, %72)
  %77 = trackcall(<ctx>, geom, %69, %70, %73, <metadata>)
  %78 = record!(%5, %77)
  %79 = tapeify(%5, :(%7))
  %83 = ReturnNode(%79, <metadata>)
  br 3 (%78, %83)
3: (%84, %85)
  %86 = record!(%5, %85)
  %87 = tuple(%84, %5)
  return %87
```

```
1: (%1, %2, %3)
  %4 = rand()
  %5 = %4 < %3
  br 2 unless %5
  return %2
2:
  %6 = %2 + 1
  %7 = geom(%6, %3)
  return %7
```

Figure 3: Left: Original IR of the function `geom(n, beta) = rand() < beta ? n : geom(n + 1, beta)` (some details left out for readability, hence the missing variable numbers). Right: transformation of the original IR of `geom`. This code is heavily inlined and made more compact for display purposes, and therefore not in proper SSA form anymore, but very close to the actual IR.