

# Automated analysis of algorithms implemented on top of TBD

Walter Tichy, Umut Acar, Thomas Marshall, Emanuel Jöbstl  
Karlsruhe Institute of Technology, Faculty of Computer Science  
Carnegie Mellon University, Computer Science Department

**Abstract**—The principle of incremental programming has been well known for many years. While the automatic transformation of sequential programs into efficient incremental programs is not solved in general, a number of platforms for incremental computation have been created. In this document, we inspect the incremental computation platform TBD. First, we summarize known principles like directed dependency graphs, execution traces, intrinsic trace distance and trace stability, then we create a program model which enables us to apply those principles to TBD programs. We also present an algorithm to calculate the intrinsic trace distance in practice. Finally, we show how we can make automatic optimizations to programs by analyzing the dependency graph.

## I. INTRODUCTION

This section is going to describe the purpose of incremental computation and also explain why the concept of incremental computation is useful for speeding up computations. Furthermore, used and referenced terminology should be outlined.

### A. Approaches to incremental computation

This subsection briefly describes various approaches to incremental computation, and outline their strengths and weaknesses.

These approaches include:

- Providing a platform or framework for incremental programming, utilizing
  - function caching. [1] [2]
  - formal manipulation of the program. [3]
  - differential data flow. [4]
  - a combination of multiple approaches, like memorization and execution traces. [5] [6] [7] [8]
- Providing a high-level abstraction, like an incremental database. [9]
- Deriving an incremental program from a non-incremental, non-functional program. [10]
- Deriving an incremental program from a non-incremental, functional program. [11]

## II. FUNDAMENTAL WORK

The work of U. Acar et al [12] describes the theoretical and practical concept of incremental computing using memorization and DDGs in detail. Since these concepts are important for understanding this writing, they are summarized in this section. First, general terms are explained briefly, then, a short theoretical outline for the term of stable algorithms is provided.

### A. Change Propagation, Dynamic Dependence Graphs and Memorization

*Change Propagation* refers to the task of re-evaluating certain parts of a program as soon as the input data changes. The target of this re-evaluation is to update the output accordingly, as if the whole program would have been re-evaluated. A change propagation algorithm is responsible for selecting the function calls in the program, which have to be re-executed. When the program is executed for the first time, no change propagation happens. This execution is called the *Initial Run* [12].

A *Dynamic Dependence Graph (DDG)* can be described as a data structure, which holds a directed graph for tracking control and data dependencies during execution [12], whereas the nodes of the graph are usually function calls in the program. In contrast to *Static Dependence Graphs* [?], DDGs are mutated during change propagation and therefore adjusted to the new program structure.

*Memorization* is the concept of storing intermediate results and re-using them during change propagation. Memorization can be combined with DDGs, by inserting memorization nodes into the graph. During change propagation, this can lead to a significant performance increase, because entire sub-trees of call graph and their corresponding results can be re-used [12].

### B. Execution Traces

A trace is a theoretical construct which can be described as an ordered tree, whereas nodes represent function calls during the program execution [12]. While similar to the DDG, a trace tracks no data dependencies. A trace usually resembles the call tree of a program.

Each node  $v$  is uniquely described by a tag, consisting of

- the function being called,  $fun(v)$
- the arguments of the function call,  $args(v)$
- the values read in the body of the function,  $reads(v)$
- the values returned to the function from its callees,  $returns(v)$
- the weight of the function,  $w(v)$ , which is equal to its execution time.

Two nodes  $v$  and  $v'$  are equal, denoted  $v \equiv v'$ , if  $fun(v) = fun(v')$ ,  $args(v) = args(v')$ ,  $reads(v) = reads(v')$  and  $returns(v) = returns(v')$ .

### C. Trace distance

Trace distance can basically be described as an edit distance between two execution traces [12] [15].

To find the minimum trace distance of two traces  $T$  and  $T'$ , the so called *cognates* relation can be used.

A set of cognates  $C$  is a relation of two traces  $T$  and  $T'$  with the set of nodes  $V$  and  $V'$ , so that

- $C \subset V \times V'$
- for each  $(v, v') \in C : v \equiv v'$
- no node is paired with more than one node.

All nodes of both traces  $T$  and  $T'$  can be colored either blue, yellow or red. Nodes which have a cognate are colored blue. Nodes of  $T$  without a cognate are colored yellow. Nodes of  $T'$  without a cognate are colored red.

The trace distance  $\delta(T, T')$  between two traces  $T$  and  $T'$  can now be calculated by summing up the weights of all yellow and red nodes. This can be formally written as:

$$\delta(T, T') = \sum_{y \in Y} w(y) + \sum_{r \in R} w(r)$$

In this equation,  $Y$  denotes the set of all yellow vertices and  $R$  is the set of all red vertices.

If the cognate relation  $C$  is maximal, the intrinsic distance is minimal, denoted as  $\delta^{min}$ . Also, a maximal cognate relation can be found using a naive greedy algorithm [12] [15].

### D. Stable algorithms

This subsection describes the concepts of stable algorithms, intrinsic trace distance and their relationship.

Also, this section should emphasize that the intrinsic trace distance forms a lower bound for the time needed by change propagation during an update. [12]

## III. TBD

The TBD platform is a framework for incremental computation currently being developed at Carnegie Mellon University (CMU). TBD follows the approach of memorization combined with directed dependency graphs (DDGs), as thoroughly described in [12]. Also, parallel computing is supported. The framework allows a programmer to write software using TBDs programming interface, while TBD automatically takes care about invoking the correct functions for change propagation, in case of an update of the input data. [?] The framework is being developed in the Scala language, which enables us to exploit the reflection capabilities of Scala for analysis [13] [14]. The source code of TBD is available at <https://github.com/twmarshall/tbd>.

### A. Programming interface

TBD needs to keep track of reads and writes of variables in the program. To accomplish this, TBD wraps all values relevant for change propagation into so called *modifiabiles* or short *Mods*. TBD automatically wraps all input data into Mods.

```
def mod[T] (
  initializer: Dest[T] => Changeable[T]
): Mod[T]
```

Fig. 1. Signature of the *mod* method

```
def write[T] (
  dest: Dest[T],
  value: T
): Changeable[T]
```

Fig. 2. Signature of the *write* method

1) *mod*: To create Mods, for example as result of the program execution, TBD provides a method *mod*. The declaration of *mod* can be seen in listing 1. The *mod* method calls a function parameter *initializer* with a *destination* or *Dest* as argument. The value written to the *Dest* by the function parameter is then stored in the *Mod*, which is returned by the *mod* method. Requiring the return type of *Changeable* simply enforces that a write is the last operation inside *initializer*.

2) *write*: To write to a *Dest*, TBD provides a *write* method. The signature of *write* can be found in listing 2. The *write* method simply takes a *Dest* and a value, and writes the value to the given *Dest*. The write method returns a *Changeable*.

3) *read*: The values from within modifiabiles have to be read explicitly. For this purpose, TBD provides a *read* method, which accepts a *Mod* as parameter and then calls a function parameter *reader* with the value of the *Mod* as first argument. The signature can be seen in in listing 3. For *read*, the function parameter *reader* also has to return a *Changeable*. Reads without an enclosed write are not useful, since the *read* method may not modify values outside of its scope.

Listing 4 shows a very simple example, which adds two Mods of type integer. First, *mod* is called to create a *Dest* for the result, then the values of *mod1* and *mod2* are read. The values of *mod1* and *mod2* are then added and written to *dest*. The nesting of functions as seen in the example is typical for TBD.

Since all programs consist of *read*, *write* and *mod* functions, and all Modifiabiles have to be explicitly written, TBD is able to construct a DDG from monitoring the calls to the corresponding functions.

4) *memo*: As we already mentioned, TBD not only utilizes DDGs, but also memorization. To accomplish memorization, tbd provides a method to create so-called *Lifts*, which in turn provide a method for memorization, *memo*. The *memo* method accepts a list of parameters, which are used to match

```
def read[T, U <: Changeable[_]] (
  mod: Mod[T],
  reader: T => U
): U
```

Fig. 3. Signature of the *read* method

```

def add(
    tbd: TBD,
    mod1: Mod[Int],
    mod2: Mod[Int]
): Mod[Int] = {
    tbd.mod((dest: Dest[Int]) => {
        tbd.read(mod1) (v1 => {
            tbd.read(mod2) (v2 => {
                tbd.write(dest, v1 + v2)
            })
        })
    })
}

```

Fig. 4. A basic example, utilizing *read*, *write*, and *mod*

```

def memo(
    args: List[_],
    func: () => T
): T

```

Fig. 5. Signature of the *memo* method

this *memo* call and a function parameter *func*. A Lift can be described as memorization context. Calling *memo* with the same parameters as any previous call on the same Lift will yield the same result, without evaluation *func*. If there is no match, *func* will be called and the result will be stored for future memorization. In general, it is important to not share Lift objects between unrelated function calls, but to preserve the same Lift for all calls to the same function. The signature of *memo* can be seen in listing 5.

A typical use case for memorization is list processing. A typical example is shown in 6. First, we define a class for list nodes and the properties *value* of type integer and *next*. Note that the class is immutable. Next, we define a function, *incrementalList*, which initializes a lift and calls a recursive function, *incrementRecursive* with the head of the list and the created lift. The latter function maps each list node to a list node with *value* increased by one. This is done by first creating a Dest *dest* for the new ListNode. Then, the current node is read from its modifiable. If the current node is null, the end of the list is reached and *null* can be written to *dest*. If the current node is not *null*, the value is read, increased, and written again to create the Mod *newValue*, similar to the example in listing 4.

Then, *incrementRecursive* is called recursively with the next node as parameter. The call to *incrementRecursive*, however, is enclosed in a memo operation, with the next node as parameter. If a change propagation happens now, TBD is not going to recursively call all reads again, but will stop as soon as a memo match occurs. This is typically the case as soon as the recursion reaches an unchanged list element.

In the end, a new list node is constructed from the results and returned.

5) *par*: The last crucial method offered by TBD is a method to execute code in parallel, *par*. The *par* method takes two function parameters *one* and *two*, where each

```

def par[T, U](
    one: TBD => T,
    two: TBD => U
): Tuple2[T, U]

```

Fig. 7. Signature of the *par* method

function parameter is executed on a separate worker thread with separate TBD objects. The *par* method blocks until both workers are finished. The signature of *par* is shown in listing 7

## B. Constraints and responsibilities

During change propagation, TBD re-evaluates all *read* calls that read modifiables which have changed, in the same order they were called during the initial run. Obviously, the functions invoked *read*, *mod*, *memo* and *par* may not write variables outside of their scope, or they will easily break change propagation. They have to be side effect free. If, for example, a static variable is written from within a function called by *read*, and then used somewhere else in the program, the system has no way to propagate a change of this variable.

Furthermore, all functions called have to be deterministic. Calling the same function with the same parameters has to lead to the same return value or the same value written to a dest. Otherwise, memorization will not be useable in the program.

For each function parameter passed to the functions *read* or *mod*, the last operation executed in that function parameter has to be a *write*. This is enforced by requiring the return type of *Changeable* for function parameters.

[TODO: describe the order of function calls. ]

## IV. A THEORETICAL MODEL FOR TBD

While we can retain the definition of a trace, we have to adjust the definition of nodes and node equality for our purpose, so we can use it in the next section to construct an algorithm for trace distance calculation.

### A. Trace node equality and similarity

As described in section III, TBD provides *read*, *mod*, *write*, *memo* and *par* methods to the developer. Instead of creating an execution trace out of all functions in the program, we restrict ourselves to a trace consisting of only these functions. It should be noted, that, since we require each function to be side-effect free and deterministic, we could theoretically omit *write* nodes in the DDG, since they directly depend on their corresponding parent nodes. However, including these nodes can provide useful insights during debugging.

**Definition 1 (Trace nodes)** *Let each node in our execution trace represent a read, mod, write memo or par function. We annotate each node with a tuple of the following values:*

- the node type *t*, which can have the values *read*, *mod*, *write*, *memo* or *par*
- a node tag, a sequence of labels which has a different structure depending on the node type

```

class ListNode(_value: Mod[Int], _next: Mod[ListNode]) {
  val value = _value
  val next = _next
}

def incrementList(tbd: TBD, head: Mod[ListNode]): Mod[ListNode] = {
  val lift = tbd.makeLift()
  incrementRecursive(tbd, head, lift)
}

def incrementRecursive(tbd: TBD, current: Mod[ListNode], lift: Lift[ListNode])
  : Mod[ListNode] = {

  tbd.mod((dest: Dest[ListNode]) => {
    tbd.read(current)(current => {
      if(current == null) {
        tbd.write(dest, null)
      } else {
        val newValue = tbd.mod((destValue: Dest[Int]) => {
          tbd.read(current.value)(value => {
            tbd.write(destValue, value + 1)
          })
        })

        val newNext = lift.memo(List(current.next), () => {
          incrementRecursive(tbd, current.next, lift)
        })

        tbd.write(dest, new ListNode(newValue, newNext))
      }
    })
  })
}

```

Fig. 6. A basic example, utilizing *memo*

Depending on the node type, we define the following node tags:

**Definition 2** Let the tag for read nodes consist of **(a, fun)**, whereas

- *a* is the value of the modifiable being read
- *fun* is the reader function being called

**Definition 3** Let the tag for mod nodes consist of **mod(fun)**, whereas

- *fun* is the initializer function being called

**Definition 4** Let the tag for write nodes consist of **write(a, d)**, whereas

- *a* is the value being written
- *d* is the destination where *a* is being written to

**Definition 5** Let the tag for memo nodes consist of **memo((a<sub>1</sub>, ..., a<sub>n</sub>), fun)**, whereas

- $(a_1, \dots, a_n)$  is the list of values to memo match against
- *fun* is the function being called

**Definition 6** Let the tag for par nodes consist of **par(fun<sub>1</sub>, fun<sub>2</sub>)**, whereas

- *fun<sub>1</sub>* is the first function being called
- *fun<sub>2</sub>* is the second function being called

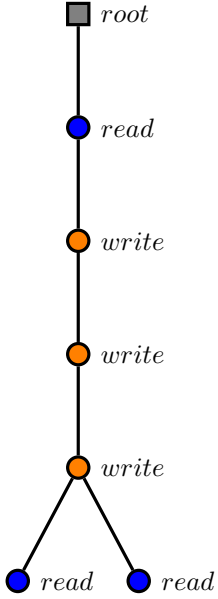
Given these definitions, we now re-define equality of nodes.

**Definition 7 (Node equality)** Let a node *A* and *B* be equal, iff the node type of *A*, *t<sub>a</sub>*, equals the node type of *B*, *t<sub>b</sub>*, and the tag of *A* equals the tag of *B*.

We only compare the tag if the node type already matches. Therefore, we can simply compare each element in the tag of *A* with its counterpart in the tag of *B*.

The tag can consist of objects, value types, modifiables or functions. For functions, showing equality is not solveable in general [16]. With the constraints of TBD programs, however, we are able to create a sufficient equality definition for our purpose.

**Definition 8 (Function execution equality for TBD traces)** A function execution *fun<sub>a</sub>* and a function execution *fun<sub>b</sub>* are equal, iff all of the following conditions apply:



- 1)  $fun_a$  and  $fun_b$  refer to the same symbol in the source code.
- 2) all arguments are equal.
- 3) all free variables bound from an outer scope are equal.

The requirement for side-effect free and deterministic functions leads to the conclusion, that all subcalls to other functions, including any writes, are going to be equal if the function is invoked with the same parameters. We have to take care of free variables in the function, however, since they might influence the behavior of the program. A prime example would be a *read* nested within another *read*, whereas the inner *read* accesses the value provided by the outer *read*, which can be seen in listing ???. If the value of *mod1* changes in the example the inner function performing the addition of *v1* and *v2* is not going to be equal anymore, therefore the *read* node of the inner *read* has changed, even the value of *mod2* stays the same.

For comparing values or objects inside the tag, function parameters or closed free variables we use *deep equality*. Modifiabiles, however should be compared by reference equality. The reason for doing so is to ensure correctness even with complex types, for example like arrays, nested lists or objects. For modifiabiles, the change propagation algorithm takes care of changed values, and automatically calls all subcalls which are affected. The case where the modifiable itself was recreated forms an exception, where we would have to re-execute all reads which would access this modifiable. This leads to the following formal definition:

**Definition 9 (Object equality for TBD traces)** A *pritive* value  $p$  is equal to a primitive value  $k$  iff  $p$  and  $k$  have the same type and the same value.

A modifiable  $x$  is equal to a modifiable  $y$  iff  $x$  and  $y$  refer to the same object in memory.

An object  $A$  with ordered properties  $(a_1, \dots, a_n)$  is equal to an object  $B$  with ordered properties  $(b_1, \dots, b_n)$  iff  $A$  and

$B$  have the same type, and  $a_i$  equals  $b_i \forall i \in [1, n]$ . Properties can be other objects, modifiabiles or primitives.

With these definition of trace node equality, we can keep the definition of *Cognates* and *TraceDistance* from [12].

## V. AN INTRINSIC TRACE DISTANCE ALGORITHM FOR TBD

While [12] already outlines a greedy algorithm for calculating the intrinsic trace distance, there are details we have to take care of for an implementation.

### A. Implementing node equality

While equality of nodes is defined in section IV-A, it still remains open how a equality is implemented. For value types or objects we can use the *equals* method provided by the Scala platform or define our own overload of *equals*, if needed.

For comparing anonymous functions passed to *read*, *memo*, *mod*, and *par*, we have to compare the abstract syntax trees (ASTs) of functions, all parameters, and all arguments, and all free variables bound from an outer scope, as described in definition 8. To accomplish this task, we can utilize the Scala macro API [13]. Basically, the Scala macro API enables us to define macros written in Scala, which are executed during compile time. From within these macros, we can access and modify the AST of our program.

To gather the necessary information for comparing anonymous functions during runtime, we replace the implementations of *read*, *memo*, *mod* and *par* with macros, which extract interesting information and create a tag from it. Then, the macros generate code which calls the original function and passes the original parameters and the tag.

[Include sample of implementation]

The symbol of the anonymous function is well known to the Scala macro, hence easy to extract. The arguments of the function are also well known for all methods provided by TBD, so they can be easily added to the tag.

Finding free variables which are bound from an outer scope is not straight-forward, because at the macro expansion step, the scala compiler has no knowledge about whether a symbol is a function or a variable, or from where it is bound. To extract only the correct symbols, we first create a list of all symbols which occur in the anonymous function  $F$  and store them in a set  $V = (v_1, \dots, v_n)$ . Then, for each  $v_i$ ,  $i \in [1..n]$ , we iterate over all ancestors of  $v_i$  in the AST of  $F$ . If we find a variable definition or parameter which defines a symbol with the same name as  $v_i$ , we know that  $v_i$  is not bound from an outer scope, so we remove it from our list  $V$ .

Then, we iterate over all ancestors in the AST of the outmost enclosing scope of  $F$ . This scope is the class in which  $F$  is defined in most cases. If we find an ancestor which defines a variable or parameter, we add the symbol of that variable to a set  $D = (d_1, \dots, d_m)$ . Finally, we compute the set  $U = (u_1, \dots, u_k) = V \cap D$ , whereas equality of elements in  $V$  and  $D$  is defined by equality of the symbol name. The set  $U$  now contains only symbols, which are used in  $F$ , defined somewhere outside  $F$  and are variables.

Now, we generate code to add the name and value of each symbol  $u_i$  to a Scala list, whereas the list is then added to the tag. The tag is passed to the original function, which adds it to the corresponding node in the DDG.

By applying the described technique, we are now able to create a tag, which can be used to compare nodes which depend on anonymous functions for equality.

### B. Implementing the intrinsic distance algorithm

Given all nodes in two traces  $T_1$  and  $T_2$ , including their tag, the trace distance can be computed like described in [12].

For a naive greedy algorithm, we create a tree- or hashset  $S_1$ , which holds all nodes from  $T_1$ . Then, we test for each node in  $T_2$ , if a node with an equal tag existed in  $S_1$ . If so, we remove the node from  $S_1$ .

When all nodes have been tested, the intrinsic trace distance is given by the size of the set  $|S_1|$  plus the count of nodes from  $T_2$  which were not contained in  $S_1$ .

### C. Proof of correctness

[We shall also proof the correctness of our algorithm. It is of importance that we take the changes of the program model outlined in the previous section into account.]

## VI. AUTOMATIC OPTIMIZATION OF PROGRAMS

This section is going to describe how analyzing the Directed Dependency Graph (DDG) can be used for automatic optimization. For accomplishing this task we can utilize the following features of the DDG:

- Caller/callee dependencies.
- Dependencies of modifiabiles<sup>1</sup>.
- Dependencies of bound variables which are not modifiabiles.

Furthermore, using the intrinsic distance algorithm, we can recognize which nodes are deleted, inserted or retained [12], which can be used to optimize the program to accomplish faster change propagation.

The exact contents described in this chapter are still to be determined, based on our findings. Possible approaches include, but may not be limited to:

- Function call reordering.
- Insertion of explicit memorization calls.
- Detection of cascading updates, which could be omitted.

<sup>1</sup>Pointer-like variables which have to be explicitly read and written, and therefore support automatic change propagation

## VII. EVALUATION

This section is going to demonstrate the usefulness of the described techniques using real-world algorithms, like map, reduce and quicksort.

Basically, it is shown how it is possible to optimize a classic implementation (without memorization) of each algorithm, so that change propagation time lies within the same complexity class as the theoretical lower bound for updates for this algorithm.

## VIII. CONCLUSION

This section will conclude and summarize with the findings of this work.

### A. Future work

The final section briefly outlines problems encountered but not solved during the writing of this thesis, as well as encourages future research on interesting issues of incremental computation.

## REFERENCES

- [1] A. Heydon, R. Levin, and Y. Yu, "Caching function calls using precise dependencies," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 311–320, 2000.
- [2] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989, p. 315328. [Online]. Available: <http://dl.acm.org/citation.cfm?id=75305>
- [3] R. F. Cohen and R. Tamassia, "Dynamic expression trees and their applications," in *SODA*, 1991, pp. 52–61.
- [4] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray, "Composable incremental and iterative data-parallel computation with naiad," Tech. Rep. MSR-TR-2012-105, October 2012. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=174076>
- [5] M. A. Hammer, U. A. Acar, and Y. Chen, "CEAL: a C-based language for self-adjusting computation," in *ACM Sigplan Notices*, vol. 44. ACM, 2009, p. 2537. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1542480>
- [6] Y. Chen, U. A. Acar, and K. Tangwongsan, "Functional Programming for Dynamic and Large Data with Self-Adjusting Computation," 2014. [Online]. Available: <http://www.mpi-sws.org/~chenyan/papers/icfp14.pdf>
- [7] U. A. Acar, A. Ahmed, and M. Blume, "Imperative self-adjusting computation," in *ACM SIGPLAN Notices*, vol. 43. ACM, 2008, p. 309322. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1328476>
- [8] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 6, pp. 990–1034, 2006.
- [9] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," in *OSDI*, vol. 10, 2010, p. 115. [Online]. Available: [https://www.usenix.org/legacy/events/osdi10/tech/full\\_papers/Peng.pdf?origin=publication\\_detail](https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Peng.pdf?origin=publication_detail)
- [10] Y. A. Liu and T. Teitelbaum, "Systematic derivation of incremental programs," *Science of Computer Programming*, vol. 24, no. 1, pp. 1–39, 1995.
- [11] R. Ley-Wild, M. Fluet, and U. A. Acar, "Compiling self-adjusting programs with continuations," in *ACM Sigplan Notices*, vol. 43, no. 9. ACM, 2008, pp. 321–334.
- [12] U. A. Acar, "Self-adjusting computation," 2005.
- [13] E. Burmako, "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming," in *Proceedings of the 4th Workshop on Scala*. ACM, 2013, p. 3.
- [14] M. Stocker, "Scala refactoring," Ph.D. dissertation, HSR Hochschule für Technik Rapperswil, 2010.

- [15] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vitter, and S. L. M. Woo, "Dynamizing static algorithms, with applications to dynamic trees and history independence," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2004, pp. 531–540.
- [16] A. Church, "A note on the entscheidungsproblem," *The journal of symbolic logic*, vol. 1, no. 01, pp. 40–41, 1936.