A Practical Approach to Analyzing Incremental Programs Using Execution Traces

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 an 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 analyze a given program by utilizing execution traces.**

I. Introduction

Classic programs and algorithms usually work on a fixed input set and produce some kind of output. For input data which changes frequently, however, the program has to be re-executed frequently too, if the output has to up-to-date. However, re-executing the whole program is not always necassary. For example, an algorithm can make use of special data-structures or memorization of intermediate results to adjust the output to match the new input data. The target of such an approach is to create a program which is capable of adopting to input data changes and update the output accordingly, faster than by re-execution. This concept in general is called *incremental computation* [1]. Programs or algorithms which exploit this concept are called *incremental programs* or *incremental algorithms*.

The task of re-evaluating certain parts of a program as soon as the input data changes is called *Change Propagation*. 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* [2].

A. Approaches to incremental computation

While it is possible to create incremental programs or data structures which are optimized for a single purpose, for example *dynamic* or *kinetic data structures*. Naturally, these algorithms usually are complex, single-purpose and can not be composed with other algorithms, which makes them difficult to use in practice [2].

A more general appraoch is to provide a platform which natively supports incremental computation for all programs writton on top of that platform. The approach of automatically inferring a incremental program from a non-incremental program is also called *self-adjusting computation* in literature [3] [2].

Ways for automatically transforming conventional programs into incremental programs are:

- Providing a high-level abstraction, like an incremental database. [4]
- Deriving an incremental program from a non-incremental, functional program. [5] [6]
- Deriving an incremental program from a non-incremental, non-functional program. [7] [8] [9] [10] [11] [12] [13] [14]

While it has been shown that a incremental program can be derived from a functional program automatically [6], this problem is not solved for non-functional programs in general. This especially holds if the underlying programming framework or API is complex, which is practically the case for every usable modern language. While it is possible to express every program in a functional language, many real-world applications are written in non-functional languages. The same argument implies that high-level abstractions, while very useful for certain cases, are not sufficient in general.

For deriving incremental programs from non-functional programs the following approaches exist:

- Memorization using function call caching. [7] [8]
- Finding calls to re-evaluate using dependency graphs. [9]
- A combination of multiple approaches, like memorization and dependency graphs. [11] [12] [13] [14] [10]

While memorization and the utilization of dependency graphs lead to the re-use of some intermediate results, has been shown that these approaches alone do not perform optimal [2]. Combining these procedures, however, leads to optimal results [2] in terms of change propagation overhead. The work of U. Acar et al[2] describes the theoretical and practical concept of incremental computing using memorization and DDGs in detail.

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 [2], whereas the nodes of the graph are usually function calls in the program. In contrast to *Static Dependence Graphs* [15], 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 the call graph and their corresponding results can be re-used [2].

B. Designing suitable algorithms

While the approaches described in the previous section can provide change propagation for any program, the performance of the change propagation strongly depends on the structure of the program and the arising of dependencies.

For example, consider a naive fold operation on a linked list $A = (a_1, ..., a_n)$ with list nodes $a_i = (v_i, n_i)$, each having a value and a next node, and an intial value r_0. We consider the list nodes immutable for our program, however changes can be done from outside the program, for example when the input changes. The fold operation would read the first node a_1 in the list, combine the value v_1 with the inital value r_0 and remember the result r_1. Then, the next a_2 node is read, the value is combined with the previous result r_1 and the result r_2 is stored again.

If we now track dependencies for this operation, each result r_i depends on the previous result r_{i-1} or the initial value r_0. Each result r_i also depends on the value v_i of the corrsponding node a_i. In other words, the fold operation for each node $a_j, j > 1$ depends on the successors of the respective node a_{j-i}.

If now a change to the first node in the input happens, a change propagation always has to re-execute the fold operation for the whole list, since every single intermediate result and the final result depend on the first node of the list. For this case, the asymptotic complexity of change propagation lies within $O(n)$ whereas n donates the input sice. This expected bound also holds for the average case [2]. The algorithm therefore updates in $O(n)$, which is not better than the expected time for a re-execution.

If we however use a randomized, tree-like fold, which combines groups of neighbors until the input list consists of only a single element, we can reach a bound of $O(log_k(n))$, whereas n denotes the input size and k the expected group sice. The need for randomization arises from the possibility of insertions or deletions in the input data: If we utilize only the position of our list nodes to decide which nodes we combine, an insertion or delition would lead to a change of this decision for each node which is a successor of the inserted or deleted node. A description and detailed analysis of this algorithm is available in [2]. The mentioned algorithm updates in $O(log(n))$, which makes change propagation more efficient than re-execution.

These two examples clearly illustrate that a developer has to take care of dependencies when writing programs. While this can be straight forward for small problems with few dependencies, keeping track of the dependencies and estimating the performance of a program can become complicated very quickly. This is especially true for algorithms where an input update can change the whole call tree. An example for such an algorithm would be a quicksort[16], where a change of the first pivot element can change the partition of the list and therefore all further subcalls.

From this, we conclude that the existence of program analysis tools for incremental algorithms is crucial for progression of incremental computation platforms.

C. Algorithm Analysis

Regarding incremental programs, the asymptotic complexity of propagating input changes through the program is of interest. While inferring asymptotic complexity is a task which is usually done by hand for a given algorithm, this approach can be hard for incremental programs, due to the complexity of underlying models and change propagation algorithms.

From a practical viewpoint however, asymptotic complexity is not always the only important property of a program. For real-world purposes, a benchmark or an analysis of a certain program execution can be sufficient to yield a meaningful statement about program performance [17].

Given two execution traces of the same incremental program with different input data, we present a practical approach to calculate a lower bound for change propagation time between these two traces. We can also show that our approach works independently from the change propagation algorithm utilized by the program. While it is not safely possible to infer asymptotic complexity from the obtained data, we can make statements regarding the expected change propagation time for given update sizes. Especially, we can decide whether the change propagation is reasonably faster than a complete re-execution of the program.

Naturally, the performance of change propagation is not the only interesting property. If the performance is not as good as expected, the question about how to increase performance arises. Since execution traces provide us with detailed sets of control and data dependencies, we can determine and inspect relationships between subsets of the program execution. By utilizing these dependencies, we are able to automatically spot issues in the program, especially regarding the correct use of memorization. Due to the mere count of dependencies which arise even in small programs, this task is cumbersome to be done by hand, while it can be easily automated.

Exploiting these approaches, we create a tool, which assists developers when writing incremental programs: The tool provides metrics about the performance of an incremental program and also suggests changes of the program structure to increase performance where applicable.

II. TBD

The *TBD* (*T*o*B*e*D*etermined) 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 throughly described in [2]. Also, parallel computing is supported. The framework allows a programmer to write software using TBDs programming interface, while

```
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

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 [19] [20]. 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 *modifiables* or short *Mods*. TBD automatically wraps all input data into Mods.

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 modifiables 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 my not modify values outside of it's 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

```
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

values of $mod1$ and $mod2$ are then added and written to $dest$. The nesting of functions as seen in the example is typical for applications on top of TBD.

Since all programs consist of $read$, $write$ and mod functions, and all Modifiables 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 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 List Node. Then, the current node is read from it's 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.

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

Fig. 7. Signature of the *par* method

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 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 usable 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. Due to this, it is generally not possible to use return statements to return values within a TBD program.

III. Theoretic fundamentals

While a very comprehensive theoretical model for incremental programs based on DDGs and memoriation can be found in [2], we have to adjust parts of the model to fit the TBD platform. The concept of a *trace* is necassary as a fundamental approach for gathering information about an incremental program. The approach of *trace distance* is important for the analysis of programs, since it can be used as a metric for the prerformance of the change propagation of a given incremental algorithm.

A. 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 [2]. While similar to the DDG, a trace tracks no data dependencies. A trace usually resembles the call tree of a program.

Definition 1 (General trace node equality) *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 $returns(v) = returns(v')$.

B. Trace distance

Trace distance can basically be described as an edit distance between two execution traces [2] [18].

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

Definition 2 (Cognates) *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 $(v, v') \in V : v \equiv v'$*
- *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 can now be calculated by summing up the weights of all yellow and red nodes.

Definition 3 (Trace distance) *The trace distance $\delta(T, T')$ between two traces T and T' is given by*

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

whereas 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 δ^{min}. Also, a maximal cognate relation can be found using a naive greedy algorithm [2] [18].

The minimal trace distance forms a lower bound for the duration of change propagation, since during change propagation all red vertices have to be deleted, and all yellow vertices have to be re-evaluated [2].

```
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 $memo$

C. Traces in 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 usable on the TBD platform.

1) TBD trace nodes: 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 4 (TBD 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*

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

Definition 5 *Let the tag for $read$ nodes consist of $(\mathbf{a}, \mathbf{fun})$, whereas*

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

Definition 6 *Let the tag for mod nodes consist of $(\mathbf{fun})$, whereas*

- *d is the id of the destination generated by this call*
- *fun is the initializer function being called*

Definition 7 *Let the tag for $write$ nodes consist of $(\mathbf{a}, \mathbf{d})$, whereas*

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

Definition 8 *Let the tag for $memo$ nodes consist of $((\mathbf{a_1}, ..., \mathbf{a_n}), \mathbf{fun})$, whereas*

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

Definition 9 *Let the tag for par nodes consist of $(\mathbf{fun_1}, \mathbf{fun_2})$, whereas*

- *fun_1 is the first function being called*
- *fun_2 is the second function being called*

2) TBD trace node equality: Given these definitions, we now re-define equality of nodes.

Definition 10 (Node equality) *Let a node A and B be equal, iff the node type of A, t_a, equals the node type of B, t_b, and the tag of A equals the tag of B.*

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

The tag can consist of objects, value types, modifiables or functions. For functions, showing equality is not solvable in general [21]. With the constraints of TBD programs, however, we are able to create a sufficient equality definition for our purpose.

Definition 11 (Function execution equality for TBD traces) *A function execution fun_a and a function execution fun_b 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 sub calls 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. An 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 4. 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*. Modifiables, 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 modifiables, the change propagation algorithm takes care of changed values, and automatically calls all sub calls which are affected. The case where the modifiable itself was re-created 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 12 (Object equality for TBD traces) *A primitive 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 $(a_1, ..., a_n)$ is equal to an object B with ordered properties $(b_1, ..., b_n)$ iff A and B have the same type and a_i equals b_i $\forall i \in [1, n]$. Properties of an object can be other objects, modifiables or primitives.

With these definition of trace node equality, we can keep the definition of $Cognates$ and $TraceDistance$ given in in section IV-A.

Figure 8 illustrates a trace of a TBD program. The program executed here is the example found in listing 6. The input consists of a list of three elements, 1, 2 and 3 in this case. Values of the form $d.\alpha$ inside the tag denote dests or mods, whereas α is the unique key. Values of the form $f.\delta$ inside the tag denote anonymous functions, whereas δ is an automatically generated unique identifier.

The leftmost subtrees correspond to creating a modifiable for the resulting value, reading the input value and writing the result. The central path holds all calls which recursively read the input and handle memorization. The rightmost and bottom $write$ calls write the resulting new list nodes.

The edges in the trace illustrate control dependencies. However, by observing the keys of dests in the mod and $write$ operations, data dependencies can be found.

IV. Implementing an intrinsic trace distance algorithm for TBD

While [2] already outlines a greedy algorithm for calculating the intrinsic trace distance, there are details we have to take care of for accomplishing an implementation.

A. Implementing node equality

While equality of nodes is defined in section IV-C2, 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 testing anonymous functions passed to $read$, $memo$, mod, and par for equality, we have to compare the the function itself, all parameters, all arguments, and all free variables bound from an outer scope, as described in definition 14. To accomplish this task, we can utilize the Scala macro API [19]. Basically, the Scala macro API enables us to define small programs written in Scala, which are executed during compile time. From within these macros, we can access all information the compiler has and modify the abstract syntax tree (AST) of our program on the fly.

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. The macro generates code which calls the original function and passes the original parameters and the tag as arguments.

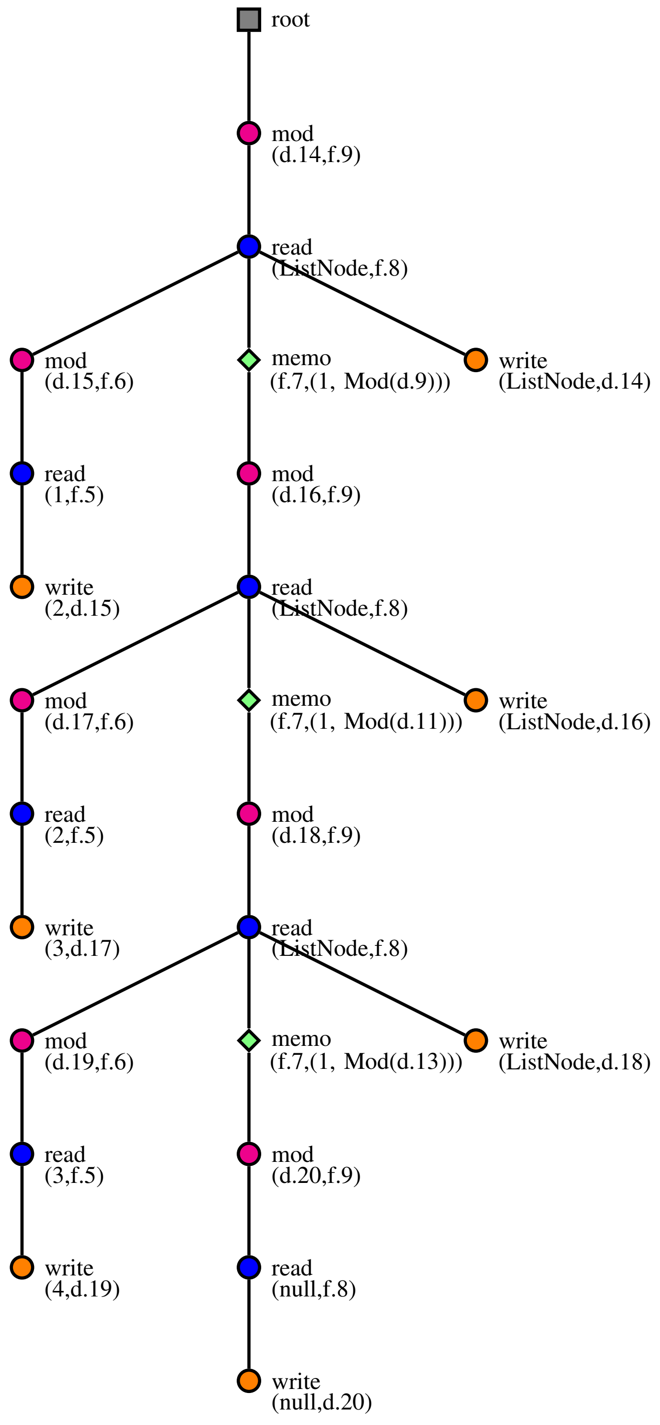

Fig. 8. Trace of a TBD program [Todo: Find more elegant solution for tags.]

During macro expansion, we can simply assign an unique ID to each function. This way, we can easily check whether to function tags refer to the same function. 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, however, 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 $V = (v_1, ..., v_n)$. 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 outermost 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 $D = (d_1, ..., d_m)$. Finally, we compute the set $U = (u_1, ..., u_k) = V \cap D$, 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 [2].

For a naive greedy algorithm, we create a tree- or hash set 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*

Whe have to show that our distance algorithm forms a lower bound for the count of nodes re-evaluated during change propagation.

Lemma 1 *Two equal trace nodes execute equally and make equal subcalls*

Proof: If the nodes are equal, they refer to same TBD function, $read$, $write$, mod, $memo$ or par. These functions are guaranteed to execute the same way if the input parameters are equal. Also, the tag of the node contains all parameters the function depends on, including free variables and higher order functions. Nodes are only equal if the tag equals. Accessing static or class variables from inside function parameters can be ruled out due to the constraints listed in section III-B. ■

Note that this can not be easily proven in a pure formal way, since this would require a theoretical model for the whole Scala language.

Definition 13 (Change propagation algorithm) *Let A be a change propagation algorithm. This means that A propagates input changes through the program and updates the program state and the output accordingly. A change propagation algorithm may only re-order, re-execute or delete nodes from an existing trace. Let $\alpha_A(I, I')$ be the count of re-evaluated nodes for a change propagation with algorithm A from an input I to another input I' for the same program.*

Definition 14 (Optimal change propagation algorithm) *Let $A*$ be an optimal change propagation algorithm. That means that $A*$ re-evaluates as few nodes as possible during change propagation.*

Let I and I' be two inputs for a program. Let T and T' be the traces of the program execution with I and I' as input. To show that our trace distance algorithm finds a lower bound for change propagation, we have to show that $\delta(T, T') = \alpha_A * (I, I')$.

Lemma 2 $\delta(T, T') \leq \alpha(I, I')$

Proof: Let Y be the set off all nodes of T without a cognate, let R be the set of all nodes of T' without a cognate. That means, that for all nodes in Y ther is no corresponding node with an equal tag in trace T' and vice versa. Therefore, we have to at least re-evaluate all nodes within Y and R, whereas the removal of a node counts as re-evaluation. Thus, the count of re-evaluated nodes is greater or equal to the trace distance. ■

Lemma 3 $\delta(T, T') \geq \alpha(I, I')$

Proof: Let B be the set of all nodes in T and T' which have a cognate and are therefore equal. Lets assume that ther is a vertex v in B which is re-executed during change propagation by the optimal algorithm A. Due to lemma 1 we know that this re-execution was unnecassary. Thereforew, our assumtion was wrong. We know now that an optimal algorithm does not re-evaluate vertixes which have a cognate, or in other words, it may only re-evaluate vertices which have no cognate. A the count of all vertices without a cognate equals the trace distance, we know that the count of re-evaluated nodes is lower or equal than the trace distance. ■

Theorem 1 $\delta(T, T') = \alpha(I, I')$

Proof: Follows directly from lemma 2 and 3. ■

V. 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 modifiables[1].
- Dependencies of bound variables which are not modifiables.

Furthermore, using the intrinsic distance algorithm, we can recognize which nodes are deleted, inserted or retained [2], 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.

VI. 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.

VII. 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.

B. Discussion

References

[1] G. Ramalingam and T. Reps, "A categorized bibliography on incremental computation," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 502–510. [Online]. Available: http://doi.acm.org/10.1145/158511.158710

[2] U. A. Acar, "Self-adjusting computation," 2005.

[3] R. Harper, *Self-adjusting computation*. Springer, 2004, p. 12. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-27836-8_1

[4] 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

[5] Y. A. Liu and T. Teitelbaum, "Systematic derivation of incremental programs," *Science of Computer Programming*, vol. 24, no. 1, pp. 1–39, 1995.

[1]Pointer-like variables which have to be explicitly read and written, and therefore support automatic change propagation

[6] 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.

[7] A. Heydon, R. Levin, and Y. Yu, “Caching function calls using precise dependencies,” *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 311–320, 2000.

[8] 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

[9] R. F. Cohen and R. Tamassia, “Dynamic expression trees and their applications,” in *SODA*, 1991, pp. 52–61.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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.

[15] A. Demers, T. Reps, and T. Teitelbaum, “Incremental evaluation for attribute grammars with application to syntax-directed editors,” in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1981, p. 105116. [Online]. Available: http://dl.acm.org/citation.cfm?id=567544

[16] C. A. Hoare, “Quicksort,” *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[17] F. I. Vokolos and E. J. Weyuker, “Performance testing of software systems,” in *Proceedings of the 1st international workshop on Software and performance*. ACM, 1998, pp. 80–87.

[18] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, 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.

[19] 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.

[20] M. Stocker, “Scala refactoring,” Ph.D. dissertation, HSR Hochschule für Technik Rapperswil, 2010.

[21] A. Church, “A note on the entscheidungsproblem,” *The journal of symbolic logic*, vol. 1, no. 01, pp. 40–41, 1936.