RWTH-Aachen

Bachelor Thesis

LuFG Informatik 12: Software and Tools for Computational Engineering

---

Integer Program Solving Call Tree Reversal

Author:
Michael Herwig

Supervisor:
Johannes Lotz

Examiner:
Uwe Naumann

March 31, 2016

Eidesstattliche Versicherung

______________________________ ______________________________

Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

______________________________ ______________________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

______________________________ ______________________________

Ort, Datum Unterschrift

Contents

1 Introduction 1
2 Call Tree Reversal 3
2.1 Reversal Schema 4
2.2 Result Checkpointing 8
2.3 Call Tree Reversal Problem 11
2.4 Heuristics 13
3 Integer Program Formulation 16
3.1 Basic Formulation 16
3.2 Result Checkpointing Extension 19
4 Call Tree Generation 22
4.1 Constructor 22
4.1.1 Application 22
4.1.2 Contraction 24
4.1.3 Recursion Constructor 26
4.1.4 Obtaining Call Trees 28
4.2 Randomization 29
4.3 Generator And Configuration 29
5 Analyzation 31
5.1 Sequence Structure 31
5.2 Fork Structure 36
6 Included Software 39
7 Conclusion 42
8 Outlook 44
References 46

1 Introduction

The call tree reversal is an algorithmic solution to reverse data flow of program execution with limited available memory using argument checkpointing. Call tree reversal is proven to be NP-complete[3] and, hence, finding a good solution in reasonable time is unlikely. Established heuristics exists for this problem but they lack potential for customization. In addition, they do not consider the possibility of result checkpointing which is an extension to the standard call tree reversal problem. To challenge those issues this thesis introduces an IP formulation solving call tree reversal. This formulation is furthermore extended to support result checkpointing and serves as foundation for further analysis comparing solutions for call tree reversal with and without result checkpointing.

The second section will introduce the call tree reversal problem. Therefore, used notation is defined and the reversal itself is explained in more detail showing insights of an actual reversal as preparation to the mathematical representation. Using the previous inspections section 2.2 will present the result checkpointing extension and point out applicable situations by comparison. After section 2.3 finalize the call tree reversal definition with an example and section 2.4 manifest already existing solutions.

Section 3 derives the IP formulation for solving call tree reversal starting with the basic formulation in section 3.1. Furthermore, the basic formulation will show more insights of an actual reversal and encapsulates recursion as foundation for the result checkpoint extension in section 3.2. Therefore, section 3.2 extends the previous model by additional properties mapping the elaborated characteristics of result checkpointing.

Before the extension can be analyzed a call tree generator is developed in section 4. To do so section 4.1.1 will introduce an abstract mathematical model for tree generations. This model is than simplified in section 4.1.3 to make it handier for practice without losing compatibility and finally mapped to call trees. The following section 4.2 will add randomization into the model and section 4.3 will close up with some practical definitions used for analysis.

In section 5 all technical expertise is used form previous sections to analyze the impact of using result checkpointing. To do so a simple sequence structure is introduced and the call tree generator configured to generate sequences with different chractersitics. Those sequences are then analyzed using different problem descriptions to elaborate different cases which are in turn used to precise the generator configuration leading to more precise and accurate conclusions. Section 5.1 will than use this sequence

structure and charactersitics to build more complex structures which get analyzed again under similar aspects.

The thesis completes with a summarization of developed and used software in section 6. The following conclusion in section 7 recaps all previous sections and concentrate on their perception giving a short synopsis of all major results. Finally, section 8 closes up with impulses for future related work which were out of scope.

2 Call Tree Reversal

The call tree reversal, short CTR, is an algorithmic solution for data-flow reversal without intraprocedural checkpointing, applied to call trees.

Definition 1. *A call tree $\mathcal{T} = (S, r, \chi)$ is a directed rooted tree with vertices S, root $r \in S$ and $\chi[s] \subset S$ denotes the children of s, further called callees of s, for every vertex $s \in S$. Additionally, we call parents callers and vertices subroutines.*

To reverse data-flow program execution splits into two sections, forward and reverse. The forward section basically executes the program as it would normally but in addition records its data-flow within every subroutine by pushing corresponding variables to a stack. Afterwards the reversal part is executed propagating over recorded data. This is done by popping the previous created stack and yields a reversed data-flow which is further interpreted for application purposes.

Figure 1 shows a minimal example of a single call from f to g. Every edge denotes the call of a subroutine, indicated by indentation. The execution order is from top to down. Whether a given subroutine shall record or interpret is here determined by its run mode(RECORD, INTERPRET) given as an argument at runtime. Such functions are mostly generated from existing source code and forwards the call to an implementation of the corresponding routine for given run mode[2, p. 78]. A common application of CTR is algorithmic differentiation, thus you will see ADJOIN as run mode in referenced literature for interpretation which is interchangeable.

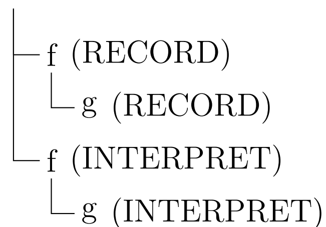


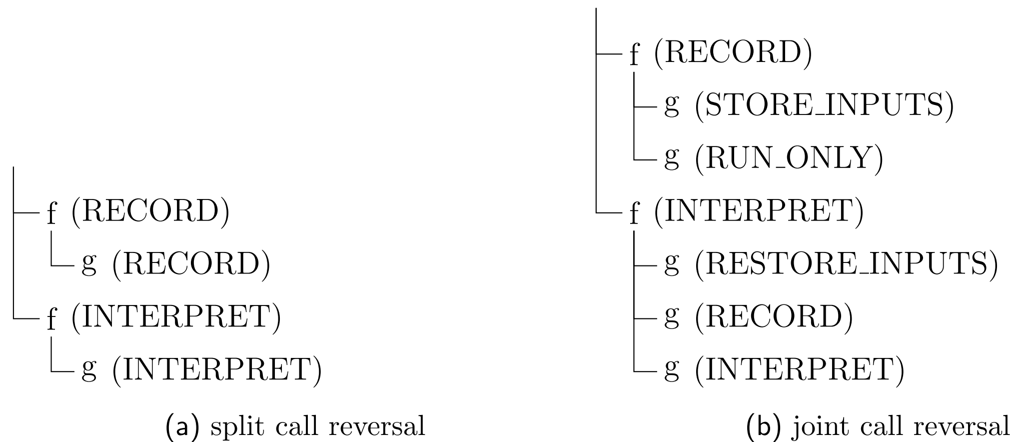
Fig. 1: Illustration of a call tree reversal recording the execution of two subroutines with following interpretation.

Notice only recorded data can be interpreted and recording data demands at least one execution of the corresponding subroutine. Furthermore, recorded data will increase the stack size requiring more available memory which can only be freed after its interpretation. Thus there is a minimal amount of memory required to reverse properly. However, the current solution, like in figure 1, records the entire program execution, before interpretation and therefore requires the most amount of memory possible. For

large call trees the required amount of memory gets swiftly infeasible. While buying more memory is a legit solution it suffers in scalability and dramatically shrinks the size of call tree which can be reversed. Therefore, the following sections will introduce new run modes leading to an algorithmic solution which sinks the required amount of memory at the cost of computation time.

2.1 Reversal Schema

While the previous reversal separated data recording and its interpretation totally form each other the upcoming solution will combine both resulting in a joint execution. Therefore, the forward section will not record the entire data flow and instead store subroutine arguments used to reevaluate data right before it will be reversed. This technique is called *subroutine argument checkpointing* and delivers a "out of context" interpretation independent of the enclosing data flow[2, p. 78]. If a routine is reversed without the context of its subroutines data-flow using subroutine argument checkpointing we refer to it as joint call reversal and otherwise as split call reversal, figure 2 opposes both.



(a) split call reversal

(b) joint call reversal

Fig. 2: Comparison of different reversal techniques according to the same call tree. (a) records the entire data flow and (b) restores subroutine arguments to record its execution right just in time. Notice (b) runs subroutine g twice, once without recording its internal data-flow, here by passing the RUN_ONLY argument and once recording it.

This technique saves memory to the cost of running g twice but before we argue further about memory consumption and computational costs the current call tree definition 2 leaks a mathematic representation to express particular characteristics.

Definition 2. *An annotated call tree* $\mathcal{T}_A = (S, r, \chi, m_c, m, c)$ *is an expansion of a call tree with* $m_c[s] \in \mathbb{R}_{\geq 0}$ *denotes the size of the subroutine argument checkpoint for any* $s \in S$ *and* $m[s] \in \mathbb{R}_{\geq 0}^{|\chi[s]|+1}$ *the tape sizes required for execution before(*$m[s]_0$*) and after(*$m[s]_{|\chi[s]|}$*) a subroutine call.* $c[s]$ *denotes the computation cost to run* s*.*

For simplification purposes notation is expanded by constants for every subroutine to refer to its vertex representative by name and $m[f] = \sum_{i=0}^{|\chi[s]|} m_i$ yields the total amount of memory required to fully record subroutine f. Now memory consumption and computational time can be expressed for split call reversal

$$\begin{aligned} \text{MEMORY} &= m[f]_0 + m[g] + m[f]_1 \\ \text{COST} &= \bar{c}[f] + \bar{c}[g] \end{aligned}$$

and joint call reversal.

$$\begin{aligned} \text{MEMORY} &= m[f]_0 + \max\{m_c[g] + m[f]_1, m[g]\} \\ \text{COST} &= \bar{c}[f] + 2 \cdot \bar{c}[g] \end{aligned}$$

Related literature uses operation counts(OPS) to measure the cost of computation. However, the time of execution for operands may depend on several characteristics like the given architecture, memory alignment and used data types(just to name a few). To overcome this problem, we neglect operations and view cost a an uniform unit of measurement.

The previous representation hided the amount of memory saved because f was shown as a single call disregarding the fact f splits into two parts embracing the call to g. The succeeding part $m[f]_1$ is exactly the part saved within the reversal(assuming $m_c[g] < m[g]$) because it is reversed before g and hence can be popped before g gets recorded.

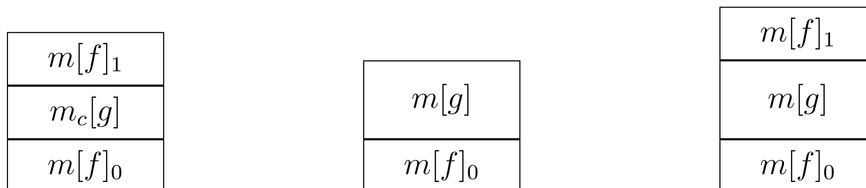


Fig. 3: Illustration of a stack during reversal. The left hand side shows the state of the stack before f is reversed and the center before g is reversed using an argument checkpoint for g. The right hand side opposes a reversal without argument checkpointing.

The current example suffices to show the difference between both reversals but is scarce in complexity for more accurate inspections. Before going into more extensive examples an alternative illustration is introduced packing related instruction into a more compressive view.

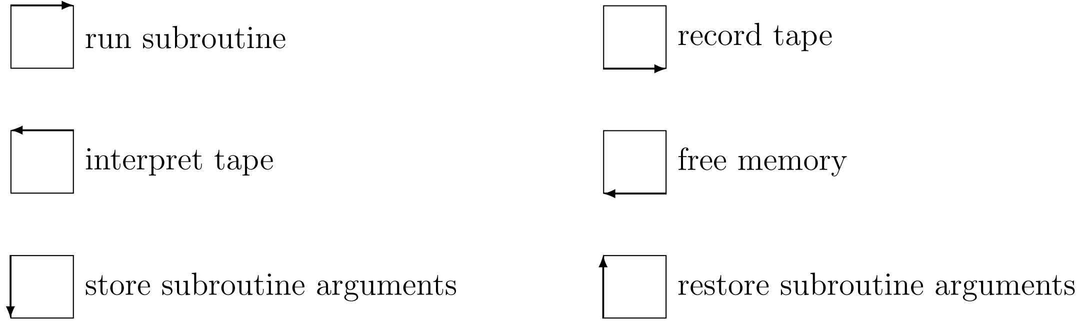


Fig. 4: Overview of possible run modes attached to the borders of subroutine vertices.

The new representation displays a square for every subroutine call, surrounding its name(constant symbol). If a subroutine is invoked the given run mode argument is attached as dart covering one border of the boundary. Possible run mode presentations are summarized in figure 4. Additionally, several run modes are allowed covering multiple borders. This allows the compression of a sequence of invocations for the same subroutine. The order of execution is given by their implicit dependency. Subroutine calls are visualized by directed arcs, ordered left to right and depth first.

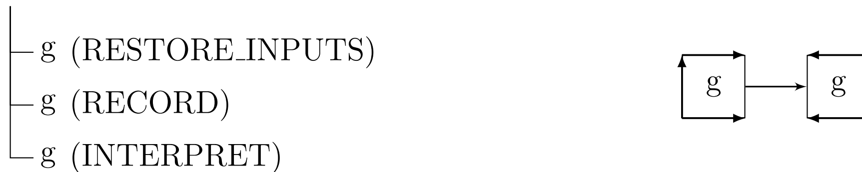
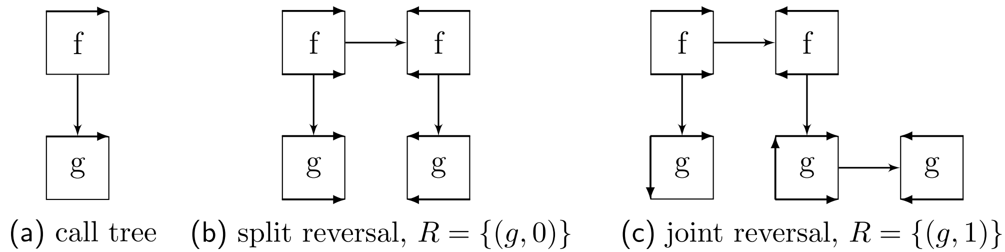


Fig. 5: Example of an equivalent reversal using different representations. Note the right hand side is more accurate showing implicit dependencies like the demand of running the subroutine in order to record it.

So far the example contained only one call to a subroutine but considering multiple calls it is possible to choose for every call to either store an argument checkpoint or record the entire execution.

Definition 3. *Let $\mathcal{T} = (S, r, \chi)$ be a call tree. We call $\overline{y} = y[s]_{s \in S} \in \sigma^{|S|}$ a reversal scheme of $\mathcal{T}$ and σ the possible reversal modes for every subroutine.*

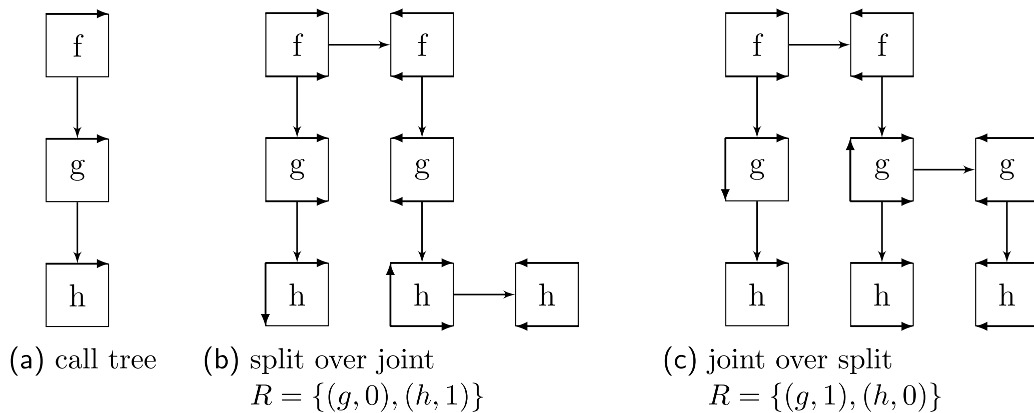
Notice definition 3 attaches reversal modes to subroutines. This is admissible because every subroutine within the call tree has only one parent and hence the reversal mode always refer to the call from its unambiguous caller. For root nodes the reversal mode can assumed to be split mode. Figure 6 shows appliance of reversal modes to the preceding call tree example. For convenience reversal schemas for a given call tree $\mathcal{C} = (S, r, \chi)$ are written as a set of tuples $R = \{(s, \overline{y}[s]) | s \in S \setminus \{r\}\} \cup \{(r, 0)\}$ and may omit the fixed pendant $\cup\{(r, 0)\}$.



(a) call tree (b) split reversal, $R = \{(g, 0)\}$ (c) joint reversal, $R = \{(g, 1)\}$

Fig. 6: Overview of different reversals(center, right) according to the same call tree(left).

Choosing reversal schemas for call trees with more than two nodes results in combinations of run modes and how they are nested into each other. Generally, two cases are of interest, split over joint and joint over split. Figure 7 opposes both.



(a) call tree (b) split over joint $R = \{(g, 0), (h, 1)\}$ (c) joint over split $R = \{(g, 1), (h, 0)\}$

Fig. 7

Comparing them shows one key aspect of a joint call not already mentioned. Split over joint stores the argument checkpoint for h within the first run because it is the last time g is running and therefore the last moment to grab the arguments. Joint

over split does not require any more memory for its children because it will be called once again hence there is no need to store data that's getting reevaluated anyway. This detail seems small but considering multiple children and nested structures a joint call acts like a termination symbol for a depth first search recoding all data needed to reverse the root node. Thus a joint call splits the reversal of a tree into reversals of subtrees each with its own data context independent of the context of other subtrees. If I refer to the context of a subroutine I refer to the data context of the subtree the subroutine is reversed in.

2.2 Result Checkpointing

The previous section introduced argument checkpointing as a method to shrink the maximum amount of memory required to reverse a call tree to the cost of additional computation. This section introduces a converse method increasing the amount of memory to stall additional computation.

Considering a reversal without argument checkpointing every subroutine is called exactly once hence there is no potential to save computation time. Furthermore, this means computation time can only be reduced if argument checkpointing is used.

Figrure 8 shows the impact of nested subroutine argument checkpoints to the amount of repetitions running subroutines. The amount of additional reruns for a given subroutine is the number of argument checkpoints used by all parent calls. Applying this to the example yields

$$\mathrm{COST}(R_1) = \sum_{s \in \mathcal{T}_A} (c[s]) + \sum_{i=0}^{d} ((i+1) * c[g_i]) + (d+1) * c[h]$$

as total costs for the reversal. Note h is called $(d+1)$ times and every time with the same arguments resulting into the same outcome. A result checkpoint tries to overcome this massive amount of reevaluation by storing the result of a subroutine making additional calls unnecessary if the subroutine is not part of the reversed data context. In order to embed result checkpointing into the present notation additional definitions are necessary.

Definition 4. *Let $\mathcal{T} = (S, r, \chi)$ be a call tree. We call $\overline{r} = r[s]_{s \in S} \in \{0,1\}^{|S|}$ a result checkpointing scheme for $\mathcal{T}$ with $r[s] = 1$ iff the subroutine $s \in S$ shall be result checkpointed.*

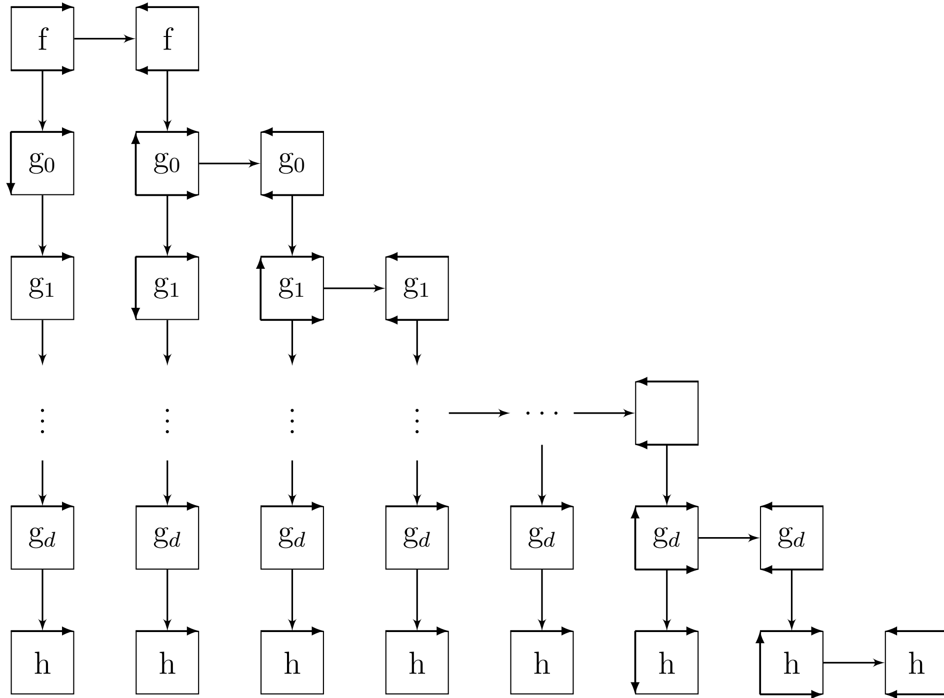


Fig. 8: Reversal of a call tree $\mathcal{T}_A$ of depth $d+2$ with $d \in \mathbb{N}$ and $R = \{(v, 1)|v \in V(\mathcal{T}_A)\}$

Fig. 9: Legend of arcs used to visualize result checkpointing within the compressed representation.

The result schema could have been mapped into the existing defintioin of a reversal schema however this would mix up argument and result checkpointing without any immediate dependency. Additionally, it makes it easier to compare reversals without and with result checkpointing by just adding another schema. If no result schema is explicitly given the reversal contains no result checkpoint by default.

Notice for every subroutine not within the context of the root node is called exactly two times if result checkponting is enabled, once to estimate the result and a second time to record the data flow. Figure 10 shows the previous example with result checkpointing for every subroutine with at least two parents. Calculating the preformed computation

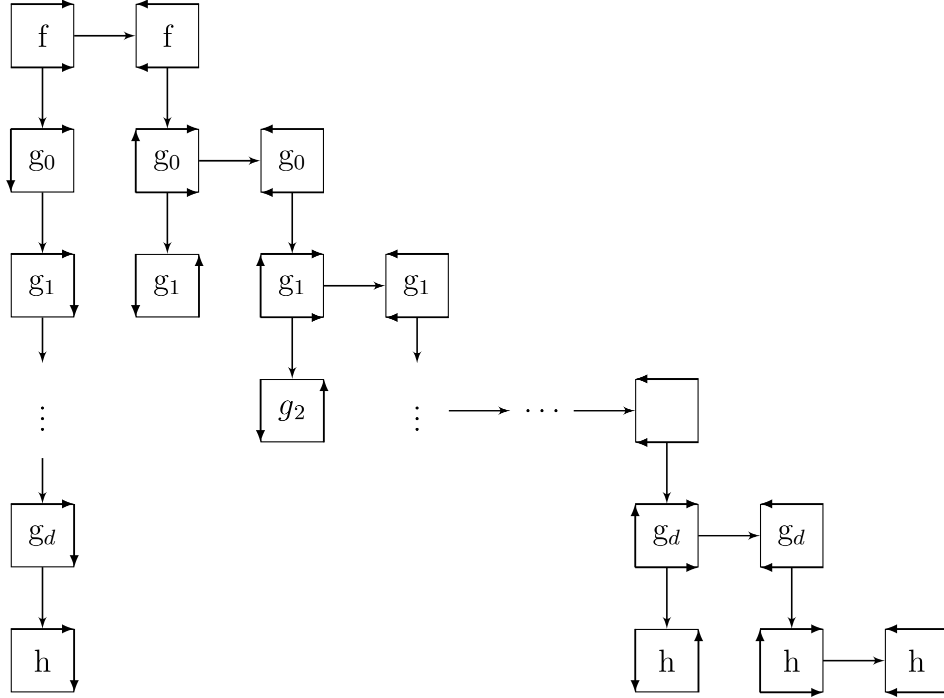


Fig. 10: Reversal of the call tree from figure 8 with result checkpointing.

$$\mathrm{COST}(R_2) = \sum_{s \in \mathcal{T}_A} (c[s]) + \sum_{i=0}^{d} (c[g_i]) + c[h]$$

estimates the total amount saved to

$$\mathrm{COST}(R_1) - \mathrm{COST}(R_2) = \sum_{i=1}^{d} (i * c[g_i]) + d * c[h]$$

using result checkpointing. Before the amount of additionally required memory can be expressed the characteristic of result sizes needs to be added to the underlying data type.

Definition 5. *An annotated call tree with result checkpoint sizes* $\mathcal{T}_R = (S, r, \chi, m_c, m_r, m, c)$ *is an expansion of an annotated call tree with* $m_r[s]$ *is the required tape size to store a result checkpoint for subroutine* $s \in S$.

So the size of memory required for all result checkpoints sums up to $\sum_{i=1}^{d}(m_r[g_i]) + m_r[h]$. Keep in mind this amount of memory is not added plain to the required

memory of a reversal because the memory peak has not to be at the start of reversing the root node and result checkpoints can be released during the reversal if not longer needed. A formal definition of the additional memory is within the IP formulation later. Also notice the memory for result checkpointing is fixed but the saved computation grows linear with the depth of nested joint calls.

2.3 Call Tree Reversal Problem

The previous sections explained the use of reversal schemas to reduce memory consumption of call tree reversals to the cost of additional computation. If enough memory is available this would not have any sense. However, for real world applications the required amount of memory is not viable. The call tree reversal problem is to find a feasible and optimal reversal schema for arbitrary hardware.

Definition 6. *The call tree reversal problem $CTR(\mathcal{T}_A, m_\top)$ is to find a valid reversal schema $\overline{y}$ for a given annotated call tree $\mathcal{T}_A$ and memory bound $m_\top \in \mathbb{R}$. The reversal schema must hold the constraint of memory bound with memory peak $MEMORY(\overline{y}) \leq m_\top$ and minimizes its cost $COST(\overline{y})$ with respect to all other possible schemas.*

Analog definition 6 could be defined utilizing result checkpointing, skipped here. CTR is proven to be NP-complete[3] consequently finding an optimal solution in reasonable time is unlikely for major sizes. This section introduces basic characteristics with a concluding example.

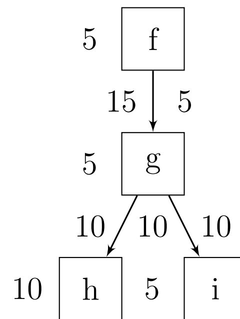


Fig. 11: Illustration of an annotated call tree with given left to right and depth-first ordering. The memory size of subroutine argument checkpoints is left to the corresponding subroutine, drawn as squares. Tape sizes for execution are drawn left, right or in between of edges indicating the call of a callee from its caller.

Global joint reversal designates the reversal with every subroutine reversed in its own context, written $R_1(\mathcal{T}_A)$ and $R_0(\mathcal{T}_A)$ is the reversal schema without argument

checkpointing called global split reversal. These reversals have an important meaning because the global joint reversal minimizes the amount of memory required while maximizing the computational cost. On the other hand, global joint maximizes memory requirements but minimizes computation. As a result

$$\begin{aligned} \text{MEMORY}\Delta(\mathcal{T}_A) =& \text{MEMORY}(R_0(\mathcal{T}_A)) - \text{MEMORY}(R_1(\mathcal{T}_A)) \\ \text{COST}\Delta(\mathcal{T}_A) =& \text{COST}(R_1(\mathcal{T}_A)) - \text{COST}(R_0(\mathcal{T}_A)) \end{aligned}$$

describes the possible range for characteristics of any arbitrary reversal. Furthermore, it bounds the solution space for a given CTR to quickly evaluate whether a CTR is infeasible or can be used to rate a given solution.

$$\begin{aligned}
R_1 =& \{(f,g,1),(g,h,1),(g,i,1)\} && \text{MEMORY} = 225, \text{COST} = 830 \\
R_1 =& \{(f,g,1),(g,h,1),(g,i,0)\} && \text{MEMORY} = 225, \text{COST} = 780 \\
R_2 =& \{(f,g,1),(g,h,0),(g,i,1)\} && \text{MEMORY} = 285, \text{COST} = 630 \\
R_3 =& \{(f,g,1),(g,h,0),(g,i,0)\} && \text{MEMORY} = 295, \text{COST} = 580 \\
R_4 =& \{(f,g,0),(g,h,1),(g,i,1)\} && \text{MEMORY} = 225, \text{COST} = 550 \\
R_5 =& \{(f,g,0),(g,h,1),(g,i,0)\} && \text{MEMORY} = 225, \text{COST} = 500 \\
R_6 =& \{(f,g,0),(g,h,0),(g,i,1)\} && \text{MEMORY} = 285, \text{COST} = 350 \\
R_0 =& \{(f,g,0),(g,h,0),(g,i,0)\} && \text{MEMORY} = 300, \text{COST} = 300
\end{aligned}$$

Fig. 12: Listing of all possible reversals for the call tree from figure 11 with their memory peak and computation cost.

The number of different reversal schemas for a given call tree is $(|S|-1)^{|\sigma|}$ as a result figure 12 already listens a bunch of reversals for a small example. If using only argument checkpointing $|\sigma| = |\{0,1\}| = 2$ but with result checkpointing the reversal encoding needs to be adjusted to $|\sigma| = |\{0,1\} \times \{0,1\}| = 4$, like mentioned earlier. Selecting a memory bound lets us select an optimal solution, figure 13 shows an example.

For bigger call trees the number of reversal gets unmanageable and leads to the problem how to value a given reversal because it is not possible to compare it to an optimal solution in decent time. With the given boundary of memory and computation cost it is possible to rate a reversal based on its relative position within the solution space. Unfortunately, this can lead into distortions as covered later in analyzation.

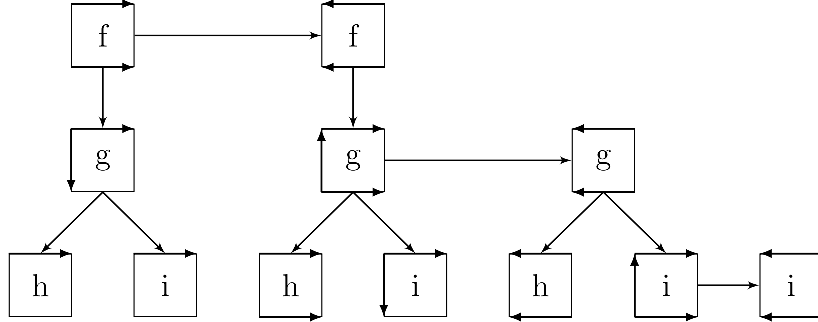


Fig. 13: Visualization of the optimal reversal R_5 with memory bound $m_\top = 250$.

Another possibility is a comparison to other solutions considered to be relatively good with application in mind. One way to calculate useful reversals for comparison is the usage of heuristics.

2.4 Heuristics

This section introduces three heuristics taken from [2]. Two of them result into the same solution but differentiate in runtime characteristics. In order to formulate those heuristics, it is necessary to sort subroutines based on their tape size required for recording. Therefore, a mapping SORT is defined which is basically a permutation of subroutines ordered by a given predicate applied to tape sizes.

Data: call tree $\mathcal{T}_A$, memory bound $m_\top$
Result: reversal schema $\overline{y}$
$I = \mathrm{SORT}(S(\mathcal{T}_A), <);$
$y[I] = 1;$
$i = |I|;$
while $i > 1$ **do**
  $y[I[i]] = 0;$
  if $MEMORY(y) > m_\top$ **then**
    $y[I[i]] = 1;$
  end
  $i = i - 1;$
end

Algorithm 1: Largest-Memory-Increase-First($R_{\mathrm{LMI}}(\mathcal{T}_A)$)

The Largest-Memory-Increase-First heuristic, shown in algorithm 1, starts with a global joint reversal and tries to switch every subroutine call to split mode. This leads to an increased memory peak and may makes the reversal infeasible, if so the

change is reverted. The algorithm loops through an ordered set of subroutines hence it starts with the subroutine requiring the most memory for recording and continues with the second most and so forth.

Data: call tree $\mathcal{T}_A$, memory bound $m_\top$
Result: reversal schema $\overline{y}$
$I = \mathrm{SORT}(\mathcal{T}_A, <)$;
$y[I] = 0$;
$i = |I|$;
while $i > 1$ **do**
  $y[I[i]] = 1$;
  if $MEMORY(y) < m_\top$ **then**
    break;
  end
  $i = i - 1$;
end

Algorithm 2: Largest-Memory-Decrease-First($R_{\mathrm{LMD}}(\mathcal{T}_A)$)

Using a reversed ordering defines the Smallest-Memory-Increase-First heuristic which results into the same solution as using the Largest-Memory-Decrease-First heuristic but is expected to reach the approximate solution faster according to [1]. The Largest-Memory-Decrease-First heuristic starts with a global split reversal to obtain a feasible solution and switches subroutine calls to split mode increasing the memory peak until the reversal is infeasible.

Data: call tree $\mathcal{T}_A$, memory bound $m_\top$
Result: reversal schema $\overline{y}$
$I = \mathrm{SORT}(S(\mathcal{T}_A), >)$;
$y[I] = 1$;
$i = |I|$;
while $i > 1$ **do**
  $y[I[i]] = 0$;
  if $MEMORY(y) > m_\top$ **then**
    $y[I[i]] = 1$;
  end
  $i = i - 1$;
end

Algorithm 3: Smallest-Memory-Increase-First($R_{\mathrm{SMI}}(\mathcal{T}_A)$)

The given heuristics are originally taken by [2] and further used to value solutions obtained by the established integer program formulation within the analyzation part.

3 Integer Program Formulation

In this section two IP formulations are derived solving the call tree reversal problem, once with and once without result checkpointing. Therefore, the reversal is reviewed in more detail to work out a universal formula for the resulting cost of a reversal schema and its' peak memory consumption. Further these formulas are used to express the constraints of bounded memory and optimization of computational cost. First the version without result checkpointing is derived and later enhanced to include result checkpointing.

3.1 Basic Formulation

In order to evaluate peak memory consumption for a given reversal schema $R = \overline{y}$ of call tree $\mathcal{T}_A = (S, r, \chi, m_c, m, c)$ every step within the reversal needs to be considered which could possibly lead to a new peak. Therefore, it is necessary to reflect every memory allocation and its data context.

For a fully split reversal this is straight forward because all data is recorded in one context before it is reversed. On the other hand, every subroutine argument checkpoint creates a new data context. Although an argument checkpoint characteristics creates its own context its data flow is not independent of its parent context because in order to guarantee a correctly reversed data flow there is at least some part of the parent embracing context reversed after. As a consequence, the only data context with no previous data on the stack is the root context. Before the amount of prepending data on the stack can be evaluated it is necessary to calculate the total amount of memory a context requires to reverse. To do so the context must include all recorded data of subroutines wihtin the context plus the data needed to restore every sub context at the time its needed, which are the argument checkpoints. The following formula expresses exactly this relationship in a mathematical way.

$$M^f[s] = m[s] + \sum_{s_c \in \chi[s]} (1 - y[s_c]) \cdot M^f[s_c] + y[s_c] \cdot m_c[s_c]$$

$(1 - y[s_c])$ evaluates to 0 if the context of the subroutine is encapsulated by an argument checkpoint thus $y[s_c] = 1$ and the context includes $m_c[s_c]$ to restore the subroutines' context later. Otherwise $y[s_c] = 0$ and $m_c[s_c]$ is not included because the current context completely occupies the context of its subroutine and $M^f[s_c]$ is added recursively.

For a fully joint reversal the number of different context is equal to the number of routines and every routine is reversed within its own encapsulated data flow. Therefore,

it is necessary to evaluate the current amount of memory on the stack every time a subroutine is reversed. This leads to a special relationship between caller and callee and how the data required for the reversal of a callee is embedded into the caller's context based on the chosen run mode. To analyze this relation figure 14 shows a generic example used to further derive the required formula.

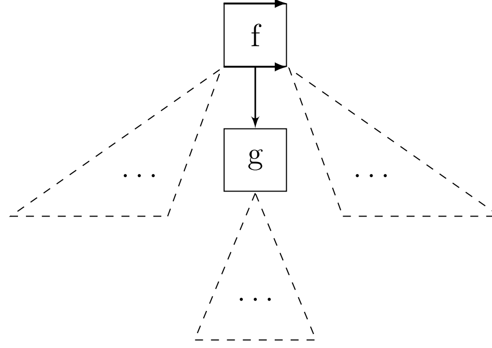


Fig. 14: Highlighted part of a reversal with a nested context of g and embracing subtrees displayed as dashed triangles. Note g has no specified run mode because it is considered to be variable.

In order to only specify memory peaks, the formula is extended by intermediate redundant steps describing the state of the stack right before one of its subroutines is called. $M^r[s, i]$ is the amount of memory on the stack during the reversal of s before the i-th subroutines' data is pushed not including a conditional checkpoint. $M^r[s, |\chi[s]| + 1]$ is the size of the stack including the entire data context of s and all data of parent context not reversed yet. Figure 15 visualizes the different states of the stack and the according notation using M^r.

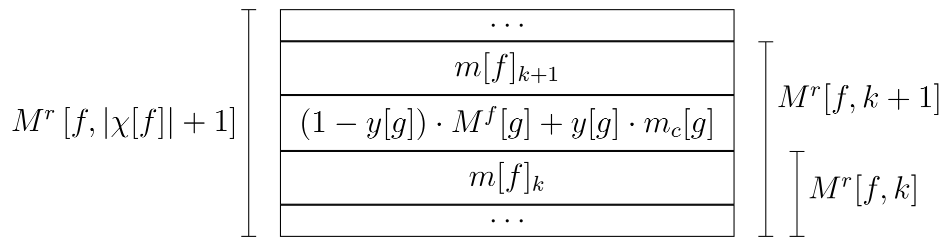


Fig. 15: Visualization of different states of the stack during the reversal of f according to figure 14. g is assumed to be the k-th child of f.

Basically $M^r[f, k]$ denotes the data on the stack which is needed to be hold during the reversal of g. For the actual reversal of g its' data context needs to be pushed

which leads to the equation

$$M^r[g, |\chi[g] + 1|] = M^r[f, k] + M^f[g]$$

, with

$$M^r[f, k] = M^r[f, k+1] - ((1 - y[g]) \cdot M^f[g] + y[g] \cdot m_c[g] + m[f]_{k+1})$$

describing the amount of memory on the stack right before g is reversed. The equation additionally clarifies the redundancy of $M^r[f, k]$ because $M^r[f, k] < M^r[f, k] + M^f[g]$ with $M^f[g] > 0$.

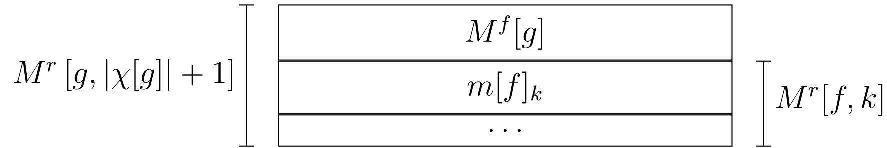


Fig. 16: Visualization of the stack right before g is reversed, according to figure 14

For the root node r no caller exists hence no data is on the stack before r is recorded which yields

$$M^r[r, |\chi[r]| + 1] = M^f[r]$$

as initial amount of memory required for reversal and terminates the recursive amount of prepending memory for all subroutines. Finally, constraints are added to guarantee the chosen reversal does not overstep a given memory bound $m_\top$.

$$M^r[s, |\chi[s]| + 1] < m_\top : s \in S$$

To record the data context of a subroutine it needs to be executed before. As a result, every argument checkpoint results into a reevaluation of the entire subtree. The cost of such reevaluation is given by

$$C[s] = c[s] + \sum_{s_c \in \chi[s]} C[s]$$

which yields additional computation costs of

$$C^r[s] = y[s] \cdot C[s] + \sum_{s_c \in \chi[s]} C^r(s_c)$$

for the reversal of given subroutine and

$$C^r[r] \to \min$$

is the objective to be minimized. Here the cost function is defined recursive showing the nested relation between the chosen run mode of a subroutine and its influence to all its direct and indirect children. However, unfolding C^r leads to an alternative definition

$$C^r[r] = \sum_{s \in S} y[s] \cdot C[s]$$

as objective. The call tree is constant at the time the IP formulation is built thus $C[s]$ can be precomputed to a vector $\overline{c}_r = (C[s])_{s \in S}$ minimizing the objective definition to the final version with a single scalar multiplication.

$$C^r[r] = \overline{c}_r^T * \overline{y} \to \min$$

This finalizes the first IP formulation. The next section will extend this formulation to include result checkpointing. Unfortunately, this will result to a nonlinear cost function and influences the performance of commonly used branch and cut algorithm dramatically.

3.2 Result Checkpointing Extension

To support result checkpointing it is necessary to choose independently whether a result checkpoint is stored for a given subroutine. Therefore, the variables are extended by a result checkpointing schema $\overline{r} = (\{0, 1\})_{s \in S}$ and previous definitions are adjusted taking care of the new schema, indicated by the index r.

First the memory requirement to store the context of a subroutine is adjusted, previously referenced as M^f. Contrary to previous definitions result checkpoints per definition are stored for subroutines not within the same context. If anything result checkpoints are preferred for deeply nested subroutines as mentioned within the preliminaries. Thus adding the size of a result checkpoint to a subroutine context efforts additional patience.

$$M^{\text{res}}[s] = \sum_{s_c \in \chi[s]} r[s_c] \cdot m_r[s_c] + M^{\text{res}}$$

M^{res} denotes the size of utilized result checkpoints for a subroutine recursively but does not include the result checkpoint of the subroutine itself.
This comes by the nature of result checkpoints. Notice the reversal in figure 17 is fully joint and hence consists of four different contexts, one for each subroutine. Further when g is recorded the result checkpoint of h is used the last time although h is not part of g's context. Generally spoken a result checkpoint is utilized the last time

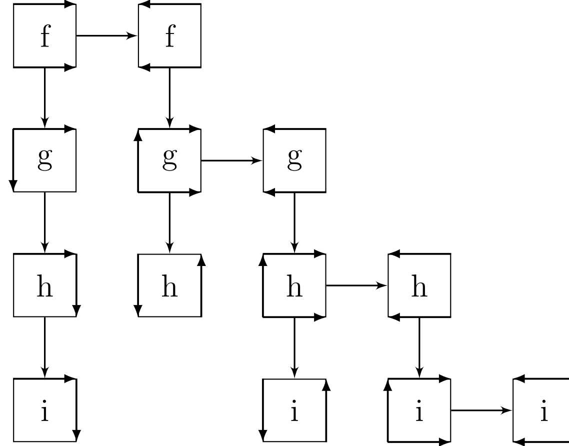


Fig. 17: A small example of call tree reversal utilizing result checkpointing.

when the parent context is recorded. Additionally, this example shows why $M^{\text{res}}[h]$ does not include the result checkpoint of h because the time h is checkpointed it will be run anyway next time to record the internal data flow. Thus the time h gets argument checkpointed the corresponding result checkpoint can be released. Keeping this in mind M^f can be extended to

$$M_r^f[s] = M^f[s] + y[s] \cdot M^{\text{res}}[s]$$

conditionally including result checkpoints if the corresponding subroutine is not part of the subroutine's context. M^r needs no further adjustments because it just works on the abstraction of a context and it was possible to express result checkpoints as part of the context of a subroutine.

Finally, the cost definition needs to be renewed. Here we benefit from the previous recursive definition C^r. The actual cost for reevaluating the context of given subroutine were given by

$$C[s] = c[s] + \sum_{s_c \in \chi[s]} C[s]$$

and needs to be modified to take result checkpoints into account. Therefore, subroutine costs are only added iff no result checkpoint is present or the subroutine is part of the current context and needs to be recorded. To express the summation of all reevaluations used to record the subroutines context a new formula

$$C_r^f[s] = c[s] + \sum_{s_c \in \chi[s]} (1 - y[s_c]) \cdot C_r^f(s_c) + y[s_c] \cdot C_r[s_c]$$

is defined. For subroutines not within the reversed context C_r is added which is not defined yet.

$$C_r[s] = r[s] \cdot \left(c[s] + \sum_{s_c \in \chi[s]} C_r[s_c] \right)$$

Notice C_r utilizes not only result checkpoints for the children of s because it is possible to store an argument checkpoint and use an existing result checkpoint at the same time. Finally, C^r can be updated to

$$C_r^r[s] = y[s] \cdot C_r^f[s] + \sum_{s_c \in \chi[s]} C_r^r(s_c)$$

without taking any initial cost into account to store all result checkpoints because they are running anyway while recording the root node and C_r^r only implies additional costs.

4 Call Tree Generation

In order to argue about internal structures and their influence on particular characteristics of the solver it is necessary to deduce a system viable to proof the existence of internal structures without the loss of abstraction in declaration. Additionally, predefined structural requirements may lead to a non-uniform distribution of objects using randomization which may distort result analysis if it leads to unexpected distribution within the anticipated objects. Therefore, the description of requirements should prohibit such declarations and underlie a mathematical model which is capable to express topology properties. The following sections will derive a capable system from a minimalistic model and extend this until we come up with a set of tools to further work on a higher level of abstraction.

4.1 Constructor

If the model should be capable to describe internal structures of the generate data type it needs to adapt this type in a related way and therefore it is manifest to start with a tree structure.

```haskell
data BinaryTree t = Leaf | Branch (BinaryTree t) t (BinaryTree t)
```

Code 1: A binary tree definition in haskell.

Code example 1 shows a common definition of a binary tree in haskell. Notice the recursive definition of subtrees combined into a higher order tree using a branch. You can think of a branch from two perspectives. First from the higher order tree containing subtrees. In this case the tree defines an abstraction of structure, subtrees are embedded in. Alternatively, from a subtrees' perspective the internal structure is not changed and the branch builds just an application of themselves. Within the next steps more complex structural combinations are built from bottom up throug applications.

4.1.1 Application

Definition 7. *For a given directed ordered graph $G = (V, E, \prec)$ we call the tuple $(v, e_1, e_2) \in V \times E^2$ a branch if $\beta(v, e_1, e_2) = (\beta_1 \vee \beta_2) \wedge \gamma$ holds with:*

$$\begin{aligned}
\beta_1 &:= \exists v_1 v_2 (v_1 \neq v_2 \wedge e_1 = (v, v_1) \wedge e_2 = (v, v_2) \wedge e_1 \prec e_2) \\
\beta_2 &:= e_1 = (v, v) \wedge e_2 = (v, v) \\
\gamma &:= \nexists v_3 (e_1 \neq (v, v_3) \wedge e_2 \neq (v, v_3) \wedge E(v, v_3))
\end{aligned}$$

If γ the branch is a leaf. Additionally, G can be extended by a relation $B := \{v \in V | \exists e_1, e_2(\beta(v, e_1, e_2))\}$ containing all vertices which are part of a branch within the graph.

Note definition 7 of a branch allows circles. At this point neglect the option of circles except for the loop indicating leaves.

Lemma 1. *For every $v \in V$ there is at most one branch $b = (v, e_1, e_2)$ for any $e_1, e_2 \in E$.*

Proof. Let $v \in V$ be any vertex and $b_1 = (v, e_1, e_2), b_2 = (v, e_3, e_4)$ two valid branches with $b_1 \neq b_2$. v is both times the same hence edges differ. If $e_1 = e_4 \wedge e_2 = e_3$ either $e_1 = e_2 \Rightarrow b_1 = b_2 ↯$ or $e_1 \neq e_2 \underset{\beta_1}{\Rightarrow} e_1 \prec e_2 \wedge e_3 \prec e_4 \Rightarrow e_1 \prec e_2 \prec e_1 ↯$. Otherwise $e_1 \neq e_4 \vee e_2 \neq e_3$ injures γ. Lemma 1 is proven by contraposition. □

Definition 8. *Let $G = (V, E, \prec)$ be a directed ordered graph. G is called fully branch-decomposable iff $\forall v(Bv)$ and $\mathfrak{B}$ names the class of all fully branch-decomposable directed ordered graphs. For any $B \in \mathfrak{B}$ $\beta : V \mapsto V \times E^2$ maps every vertex to its unique branch by lemma 1.*

To build a subset of $\mathfrak{B}$ through application at least one primitive is required, previous referenced as leaf.

Corollary 1. *$B_\perp = (\{v\}, \{(v, v)\}, \emptyset) \in \mathfrak{B}$ is the fully branch-decomposable graph in $\mathfrak{B}$ with at least on vertex, minimizing the number of vertices.*

With a strict definition of fully branch-decomposable graphs an application can be defined.

Definition 9. *An application is a function $\Delta : (\mathfrak{B} \times V)^2 \mapsto (\mathfrak{B} \times V)$ defined for valid tuples $(B, v) \in \mathfrak{B} \times V$ with $v \in V(B)$. For the result of $\Delta((B_1, v_1), (B_2, v_2)) := (B_0, v_0)$, B_0 needs to hold the application integrity property for B_1 and B_2. Furthermore, v_1, v_2 must be reachable from v_0 within B_0. A graph G_0 holds the application integrity property for another graph G iff G is a subgraph of G_0 without any edge leaving the subgraph in G_0.*

Basically an application guarantees the newly created graph contains two subgraphs untouched as already mentioned. The next step is the definition of a valid application.

Definition 10. *Δ_B is called branch application with $\Delta_B((B_1, v_1), (B_2, v_2)) = ((V_0, E_0, \prec_0), v_0)$ and*

$$V_0 := V(B_1) \sqcup V(B_2) \cup \{v_0\}$$
$$E_0 := E(B_1) \sqcup E(B_2) \cup \{(v_0, v_1), (v_0, v_2)\}$$
$$\prec_0 := \prec (B_1) \sqcup \prec (B_2) \cup \{((v_0, v_1), (v_0, v_2))\}$$

You will find applications using $B_\perp$ instead of a tuple as parameter. This notation is a shorthand because there is only one vertex to choose from $B_\perp$.

Corollary 2. Δ_B *is a valid application.*

With all this in mind it is possible to define any binary tree by nesting branch applications like in figure 18, not considering the loop at every leaf. Unfortunately, there is nothing more what could be expressed by this system. A reasonable extension would be the construction of any arbitrary tree. One might think a legit solution is an abstraction of a branch application allowing an adjustable amount of children but this would lead to a graph breaking the property of being fully branch-decomposable. Obviously there is the need of fundamental changes because in fact every graph containing a vertex with more than two children is not fully branch-decomposable.

Notice the root for a created binary tree is unambiguous. This is not naturally given by the definition of an application because it let us choose any vertex we want rather from anonymous nesting as part of the bottom up construction.

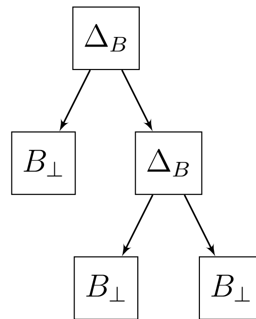


Fig. 18: Visualization of a binary tree created by nested application using $\Delta_B(B_\perp, \Delta_B(B_\perp, B_\perp))$.

4.1.2 Contraction

To overcome the previous problem, the creation of a graph splits into two different phases, characterization and contraction. During characterization the graph is likely built as before with some minor changes to store information for the following contraction phase.

Definition 11. *A contraction is a function* $\alpha : \mathcal{G} \times E \mapsto \mathcal{G}$ *with*

$$\alpha((V,E),\{v,w\}) := (V \backslash \{w\} \cup \{v\}, E \backslash \{\{u,w\} | u \in V\} \cup \{\{u,v\} | \{u,w\} \in E \wedge u \neq v\})$$

Implicitly contraction prohibit loops hence there is no need to pay further attention for them within the contracted graph. Furthermore, the order of contraction does not vary the resulting graph. To determine edges to be contracted a relation $C \subseteq E$ is added to the graph structure containing those.

$$\alpha_C(G) = \begin{cases} G & \text{if } |C| = 0, \\ \alpha(\alpha_{C \setminus e}(G), e) & \text{for any } e \in C \end{cases}$$

$\alpha_C(G)$ is the contraction phase. From now on every graph creation carries the required relation. Therefore, the branch application needs to be updated.

Definition 12. *The right hand side branch application Δ_B^R works like the normal branch application but marks its' second edge (v_0, v_2) for contraction.*

$$C_0 := C(B_1) \sqcup C(B_2) \cup \{(v_0, v_2)\}$$

Up to this point every application was defined using the internal structure of the graph. From now on the right hand side branch application is used to build more complex structures. An introducing example is a definition of the previous application Δ_B shown in figure 19.

Before concluding contraction there are some aspects deserved closer attention. First only a right hand side branch application is defined. This is sufficient because every construction using Δ_B^R only differs in the ordering of its' children from a potentially left hand side constructor. Thus which application is chosen does not matter for expressiveness. I prefer the right hand side constructor because it leads to a nested formula with the first child always on the left in the most outer bracket. Second every right hand side application results in exactly one new vertex within the resulting graph. Third figure 19 shows λ naming a vertex. This notation will further occur more often. It is used for implicit created vertices with no impact on the contracted graph thus they are anonym and only passed through nesting.

Finally, the graph construction using nested application and following contraction needs a formal definition.

Definition 13. *The class of fully branch-decomposable graphs $\mathfrak{B}_{\Delta_B^R} \subseteq \mathfrak{B}$ created by applications of leaves is defined by induction:*

(i) $(B_\perp, v) \in \mathfrak{B}_{\Delta_B^R} \times V, \{v\} = V(B_\perp)$

(ii) $(B_1, v_1), (B_2, v_2) \in \mathfrak{B}_{\Delta_B^R} \times V \Rightarrow \Delta_B^R((B_1, v_1), (B_2, v_2)) \in \mathfrak{B}_{\Delta_B^R} \times V$

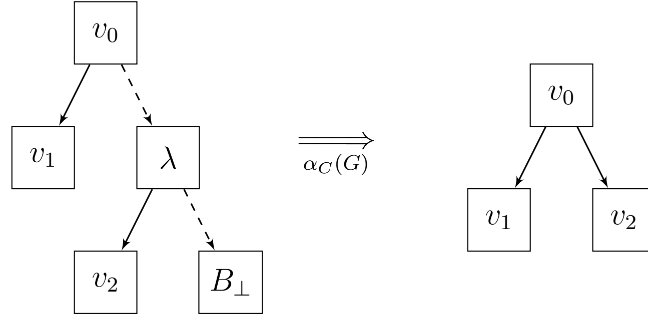


Fig. 19: Visualization of an equivalent resulting graph for Δ_B using Δ_B^R with $(G, v_0) = \Delta_B((B_1, v_1), (B_2, v_2)) = \Delta_B^R((B_1, v_1), \Delta_B^R((B_2, v_2), B_\perp)$. The left hand side shows the configuration and the right hand side the obtained graph after contraction. Dashed edges are marked for contraction.

(iii) $\mathfrak{B}_{\Delta_B^R} := \{B | (B, v) \in \mathfrak{B}_{\Delta_B^R} \times V\}$

Lemma 2. *The class of graphs* $\mathfrak{G}_T := \{\alpha_{C(B)}(B) | B \in \mathfrak{B}_{\Delta_B^R}\}$ *is equal to the class of trees.*

Proof. By structural induction over a tree starting with leaves. Siblings are combined using Δ_B and a parent is the created vertex from an application of its' children. □

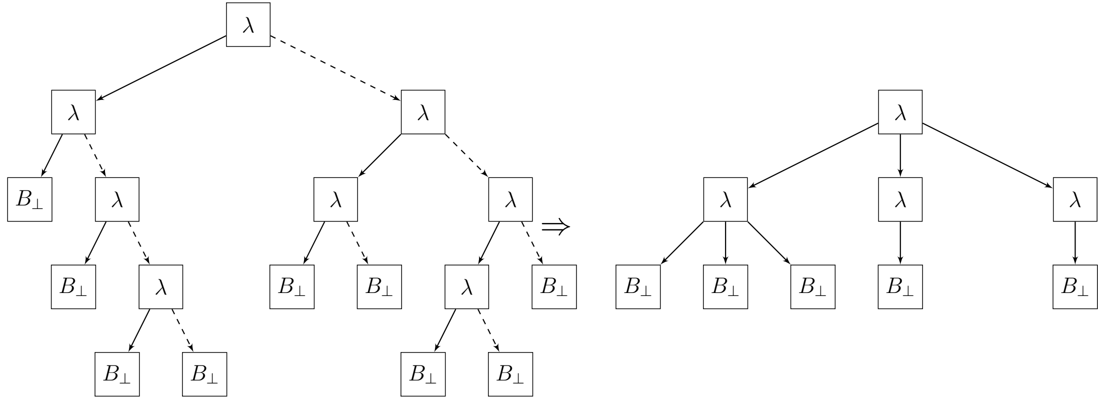


Fig. 20: Example of a more complex tree(right) created by a nested function of applications(left).

4.1.3 Recursion Constructor

The previous system is strong enough to build any tree thus the following sections will relax the system to the sake formality to improve practical usage. However, the

reference to this system is preserved and features added carefully. The sequentially improved structure is called recursion constructor and is initially equal to the right hand side branch application.

First the ordering constraint is removed mixing left and right hand side branch application up. Thus the recursion constructor has two edges marked for contraction and embracing the unmarked edge. This will not improve expressiveness because the arbitrary ordering can always be mapped back to a left or right sided ordering.

Second a recursion constructor has an integer property n. Before the tree is generated the recursion constructor unfolds into a sequence of recursion constructors of length n. Each of this constructors are a copy of itself without this extension with its middle edge set to the next constructor in the sequence. The last constructor keeps its edge to the originally target. By default, n is equal to one.

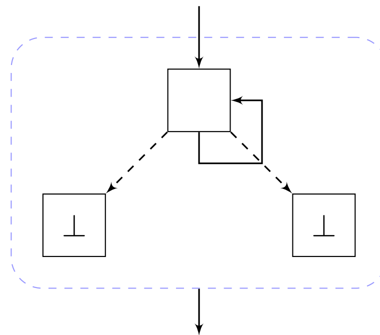


Fig. 21: Visualization of a recursion constructor. The looped edge indicates the created sequence of length n and $B_\perp$ is simplified to $\perp$.

Third the middle edge can be marked individually if required. Notice this will cause the sequence to have its edge marked as well but this can be bypassed by making three recursion constructors in chain thus an implementation could offer to set both individually. Precisely this relaxation makes it possible to choose for every edge to be marked in particular. However, this does not break backward compatibility to the previous system because constructors with fully marked edges can be contracted with no impact and for multiple unmarked edges an intermediate constructor can be defined as already mentioned.

Fourth constructors can be shared to multiple parent as long as the resulting structure

is acyclic. This is the most useful extension. To retain backward compatibility cycles are forbidden. Therewith a breadth search creates copies for all visited constructors resulting in a tree with the same structural nature as if constructors were shared.

Nested recursion constructors are now quite similar to call graphs but they prohibit circular references to guarantee termination and the contraction phase gives the possibility to hook into structures or reorder them without touching the internal structure. In fact an contracted edge does not result in an edge within the resulting graph but from now on we stick more to the idea of an extended call graph namely and use edge/call and vertex/sub-routine interchangeable.

4.1.4 Obtaining Call Trees

The recursion constructor defines a powerful tool for tree generation but misses costs, tape sizes and argument and result checkpoint sizes in order to obtain a valid call tree. To create those, every vertex is extended by a corresponding value set to zero by default. However, this cannot create tape sizes between calls. This is where contraction comes into account.

For all attributes the contraction of two nodes adds both values plain together except for tape sizes. Tape size are relatively added to the edge position of the caller not counting calls marked for contraction. As a result, to place tape sizes between calls it is necessary set a contraction call in between pointing to a vertex with the desired tape size. The tape size of a vertex with children is always added at front.

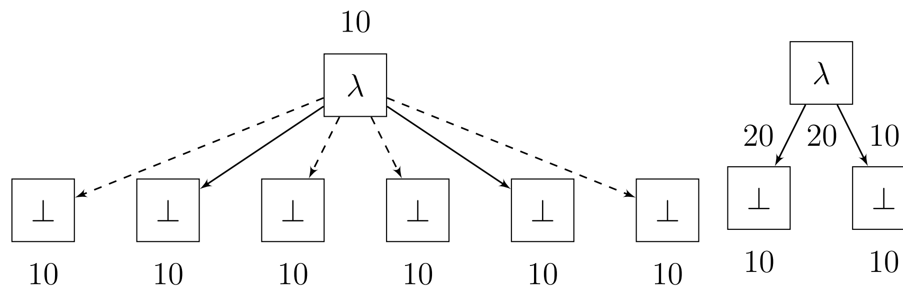


Fig. 22: Example of a contraction leading to tape sizes between calls. On the left the tree before contraction and on the right after.

4.2 Randomization

To create call trees randomly two cases must be considered.

First the characteristics used to obtain a call tree must be chosen randomly. Unfortunately, it is not enough to choose corresponding random values for each recursion constructor because a shared constructor could lead to multiple routines all with the same value. Therefore, the values are replaced by function which will return a new appropriate random value each time it is called. During creation according function are then called every time the constructor results into a new vertex determine its values.

Second the created structure must be randomized. In order to create constructors and combinations of them randomly I preferred another solution choosing the number a recursion constructor calls itself randomly each time it unfolds. Therefore, the actual value is replaced with a function just as the other characteristics. Notice this allows conditional calls by returning a zero out of the function.

Further recursion constructors use this technique to make randomization available for all generated call trees.

4.3 Generator And Configuration

To analyze the impact of arbitrarily properties customizing the output of a generated call tree it is necessary to distinguish between a generator and a configuration.

Definition 14. *A call tree generator is a tuple* $(G, \mathcal{B})$. G *is a acyclic graph of recursion construtors and* $\mathcal{B} = P_1 \times \ldots \times P_k$ *the set of all possible configurations with properties* P_1 *to* P_k *called build matrix.*

Properties are basically functions which can change attached values of recursion constructors before they are used to generate the call tree without changing the structure of the graph. To analyze the impact of those properties the generation is called multiple times for every possible combination of properties. This results into a matrix of generated call trees with the same dimension as the corresponding build matrix.

This technique is extensively used within the analyzation to inspect the behavior of varying loop sizes or sequence depths for example. Additionally, using a combination of multiple properties makes it possible to slice the resulting matrix to see the impact of growing depth and breadth at the same time. Notice there is an implicit property. The seed of a random generator used during generation need to be always the same for

determinism. On the other hand, choosing different seeds as part of the build matrix makes it possible to improve the stability generated call trees and more accurate analysis.

5 Analyzation

This section will analyze an implementation of the IP formulation using CMPL, focused on a comparison of both versions, with and without result checkpointing. Therefore, primitive structures are analyzed on its own and later combined together to more complex ones. The analyzation will focus on structural argumentation and neglect the reference to programming features or call graph relations.

5.1 Sequence Structure

As already mentioned within the preliminaries the amount of reevaluations for a single subroutine grows linear with the number of parent context which is the number of direct and indirect parents being argument checkpointed. Hence to utilize result checkpointing it is necessary to first have a growing depth and second enforce the reversal to have as much joint run modes as possible.

The first part is quite simple using a sequence of operations created by a recursion terminating after calling itself d times. The resulting call tree is $\mathcal{T}_d = (S, r, \chi, m_c, m_r, m, c)$ with $|S| = d$ and all costs and memory sizes are set to ten. To be specific the step sizes for all internal nodes is set to five which result in a fully tape size of ten because they have exactly one child and hence two step sizes. $s_i \in S$ denotes the subroutine with depth i.

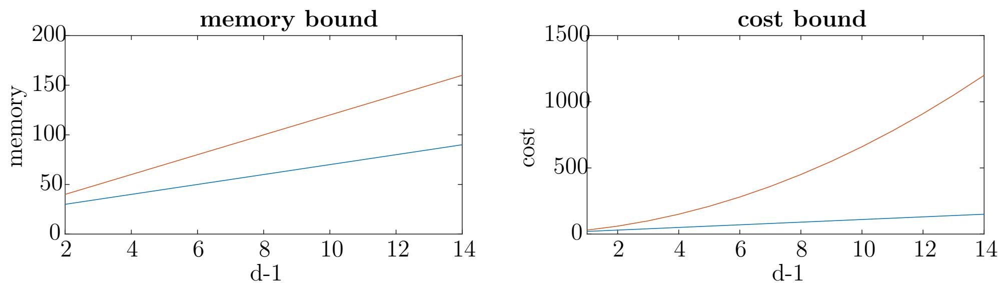


Fig. 23: Exposure of the memory and cost bound of $\mathcal{T}_d$ for growing d.

Figure 23 shows the resulting boundaries for $\mathcal{T}_d$. It is worth to notice the cost upper bound grows exponential with d while others grow linear. Also notice the memory bound span does not grow very much because the amount of memory saved by a split operation within a sequence is only the remaining part of all parents.

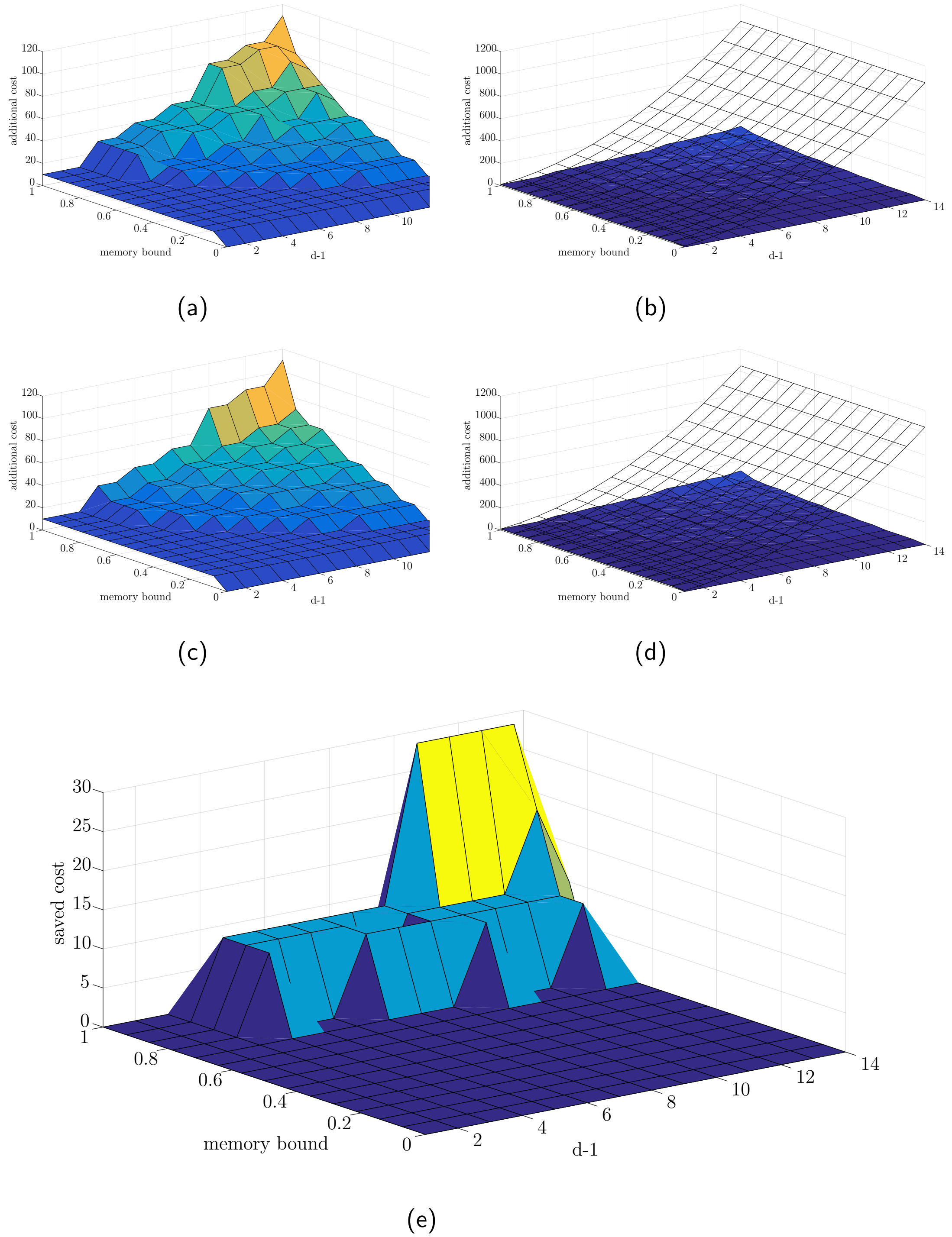


Fig. 24: Analytics of $\mathcal{T}_d$ for growing d and shrinking memory mound.

Using both versions of the IP formulation for different memory bounds chosen by a linear interpolation yields figure 24. The memory bound is interpolated from zero(maximum) to one(minimum) within the interval given by global split and joint reversals. 24a shows the additional cost from a reversal computed with the standard IP formulation and 24b the same but with the maximum additional cost possible by a global joint reversal laid over. Figure 24c and 24d present the same respectively with result checkpointing. The saved amount of computation cost using result checkpointing is displayed by figure 24e.

Figure 24b shows that the problem description fails to force the standard IP to fully split the reversal. A closer look into the reversal for a shrinking memory bound and fixed depth, here eight, shows what happens.

	$y[s_i]$							
m	1	2	3	4	5	6	7	8
0.0	0	0	0	0	0	0	0	0
0.2	0	0	0	0	0	0	0	1
0.4	0	0	0	0	0	0	1	0
0.6	0	0	0	0	0	1	0	0
0.8	0	0	0	0	0	1	0	1
1.0	0	0	0	0	1	0	1	1

Fig. 25: Tabular representation of schemas chosen by the standard IP with shrinking memory bound for $\mathcal{T}_8$.

One argument checkpoint climbs the sequence up and frees the remaining memory from its parents for its children thus the memory peak is at the root node holding the argument checkpoint and its remainder before reversing. Then a point is reached where climbing higher is not an option because the deepest subroutines reaches the memory bound. Therefore, the IP makes a second argument checkpoint for the leaf, what makes sense because this costs at least. The last step is the most interesting because the root and the leaf reaches the memory bound and therefore one checkpoint climbs up and additionally a new one is set.

Comparing the structure of raising costs for both IPs shows the IP with result checkpointing is very much smoother. Considering a reversal where a result checkpoint is placed under an argument checkpoint results into one call for each routine above the argument checkpoint and two for every below. As a result, setting the argument checkpoint one step higher results into the cost of one additional call. This is exactly

what happens and explains why the level of cost perfectly align to multiples of ten until free space is squeezed out and no memory left to store result checkpoints.

However, the additional cost is relatively low even for the tightest memory bound. To increase the level and number of joints it is necessary to relocate the memory peak to the top subroutines. To do so the balance of prepending and remaining tape sizes is changed. Now both are always the same but a greater remaining tape size at the top could force the reversal to choose those because they can free up more space for their children. Generally, four cases are distinguished shown by figure 26. Case 26a were already probed and 26d has no effect because it just increases the overall memory bound. 26b has the best effect shown in figure 24b. Case 26c looks promising unfortunately it is not an behaves like 26a. Figure 27 uses the formula $m[s_i, 2] = (d - i) * 5$ for remaining tape sizes and delivers the best result of distributions I have researched so far. Actually I tried to use a scalar factor k and visualized the standard IP solution for the lower memory bound in dependency to k shown in figure 28 with a negative effect for $k > 1$.

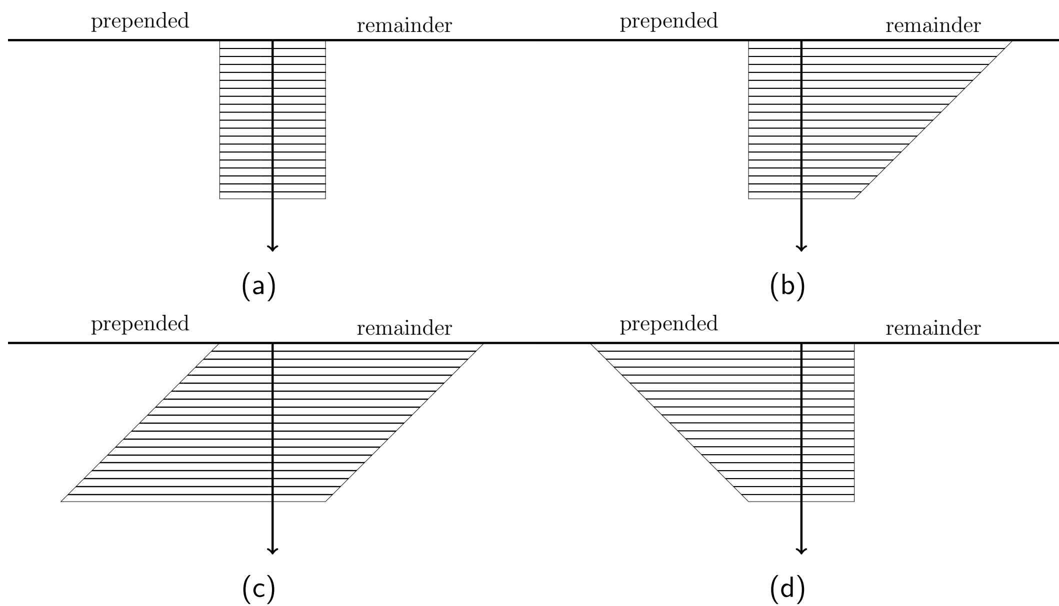


Fig. 26: Visualization of different balances of prepending and remaining tape sizes dependent on the subroutines depth.

Unfortunately, promising concepts like growing the argument size in depth or shrinking the result size yield no worthy results. Even hybrid solutions using multiple

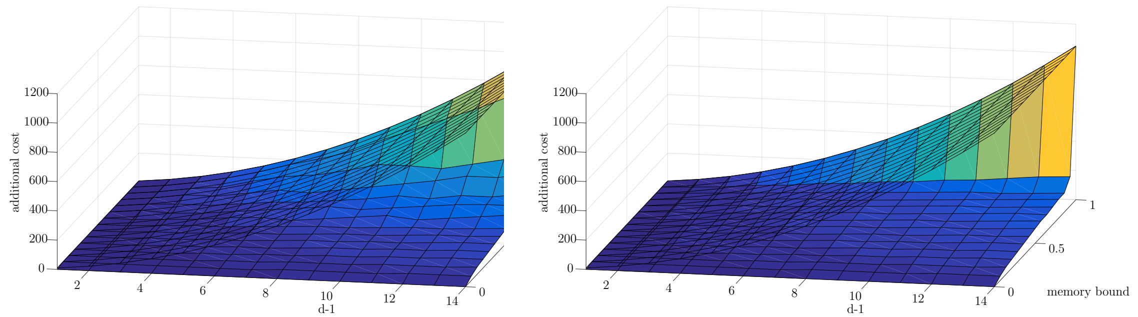


Fig. 27: Analytics for shrinking remainder size with subroutines depth. On the left the additional cost for a reversal schema obtained by the standard IP and on the right the IP with result checkpointing.

factors for shrinking remainder and argument checkpoint size were dominated by the examples introduced. Thus I stick to the four cases of tape size balances, shortly summarized.

- Case 26a
  The overall costs are relatively low. Result checkpointing utilization is limited by the argument checkpoint position.
- Case 26b
  Using the standard IP results into a fully joint reversal for tight memory bounds. Result checkpointing utilizes spacing and delivers the best results possible for memory bounds close to the minimum.
- Case 26c
  The standard IP results also into a fully joint reversal but the memory space of parents is squeezed out delivering no notable saving with result checkpointing.
- Case 26d
  No important effect.

Although the analysis for now only applies to sequences this insight is the main contribution the hole analysis. To close this section up the conclusion is shortly summarized.

The saved amount of computation for one result checkpoint grows linear with the number of direct and indirect parents being argument checkpointed. In order to increase this number, it is necessary to unbalance the prepending and remaining tape size. Larger remaining tape sizes on parents with a minor depth result in a reversal

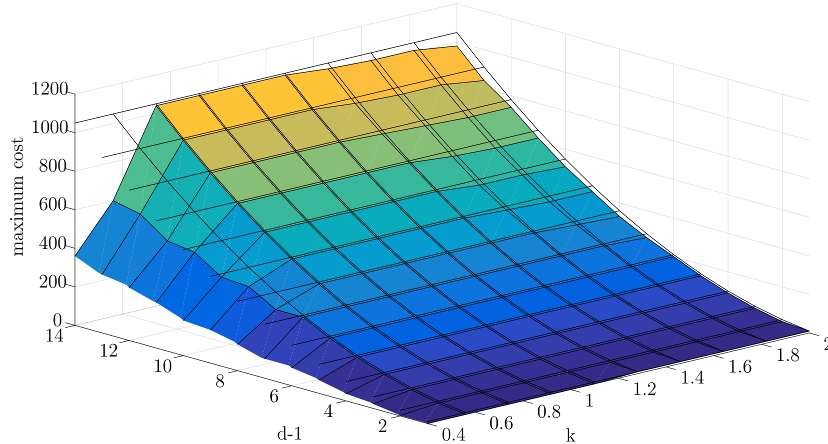


Fig. 28: Visualization of maximum cost for a reversal obtained from the standard IP in relation to the sequence depth and a scalar factor k managing the step size balance.

with argument checkpoints for top level nodes and thus more potential utilization of result checkpointing. Using this unbalance memory distribution minimizing memory bounds let the resulting reversal converge to a global split reversal. Slightly relaxing the memory bound results in free memory spaces during the reversal which can be used to store result checkpoints. The number of result checkpoints which can be stored depends on the result sizes and the amount of additional memory available. This circumstance will always repeat itself. This means even if the memory distribution is unbalanced the memory peak only relocate to deeper nodes but all parent nodes must be able to hold the result checkpoints. As a result, the potential costs saved with result checkpointing depends on the number of nested argument checkpoints but is limited by unutilized memory during the reversal of the parent with the closest peak to the memory bound.

5.2 Fork Structure

The previous sections introduced cases with different impact on utilization of result checkpointing but the example of a single sequence has only rare matchings to real world problems. To be comparable to more applications loops need to be considered but loops do not increase the depth of the tree required for nested argument checkpoints. Thus plain loops are ignored. Instead loops of sequences are analyzed.

Therefore, the previous created call tree $\mathcal{T}_d$ is extended to $\mathcal{T}_{d,k}$ with k is the number of times a new root node calls the sequence $\mathcal{T}_d$. Choosing the closest memory bound and standard IP formulation yields figure 29.

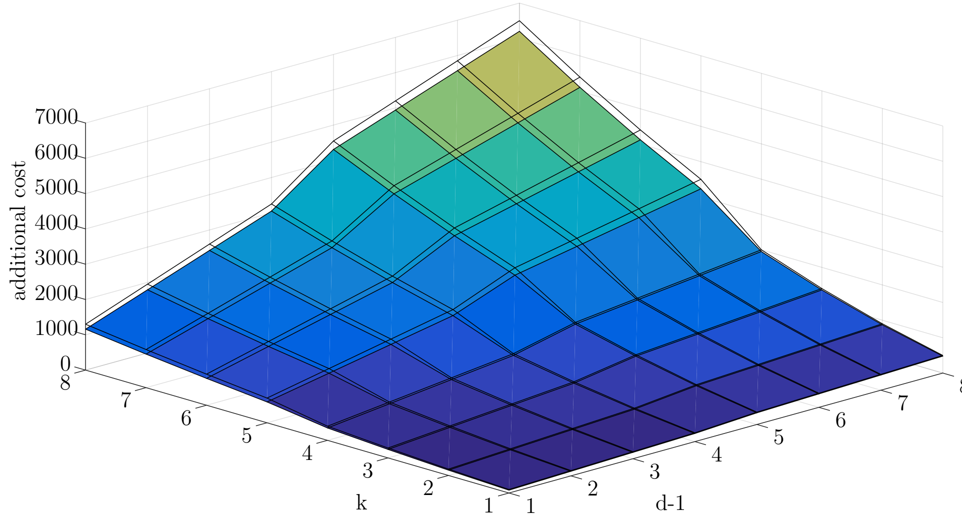


Fig. 29: Visualization of costs resulting from the standard IP applied to $\mathcal{T}_{d,k}$ with closest memory bound.

First notice the maximum amount of additional costs slightly differs because the root node itself is not argument checkpointed. This can be bypassed by increasing the argument sizes for all children. Second both parameters d and k result in a linear growth. A plot of saved computation using result checkpointing is omit because both IP formulations result into the same reversal, thus saved computation is zero and result checkpointing is not used. This is caused by the nature of result checkpoints hold by the parent for all of its children and those are also hold while reverseing the most outer child of the root. As a result, the memory peak is propagated up and the root has no memory left to store result checkpoints because it must hold all other argument checkpoints of previous children as well. This situation is kind of similar to the previous sequence but increasing the remainder has counter intuitive effects.

For too large remainder sizes, required in order to save result checkpoints for all children, argument checkpoints of subroutines within the sequence are dropped because the previous released remainder is sufficient to record the entire data flow of the sequence. This is the same observation from sequences with permissive memory bounds.

To conclude result checkpointing can give very good computational savings for sequences but are not adaptable for loops. This is caused by the propagation of result checkpoints to parents which will have a memory peak during reversal and loops causes this peaks to be closer to the actual memory bound leaving no space for result checkpoints which need to be hold by all parents.

6 Included Software

Beside the written part this thesis includes a rich set of software I developed. Every module of the software is split into single parts. Dependencies between different modules are preserved by a build system enforcing reevaluation if a dependency changed. In fact, this thesis is a module of the project itself dependent on other modules like the generator implementation. For example, some of the figures within this thesis are generated using the generator and a change of the generator or its configuration would lead to an update of all dependent figures.

To accomplish this the entire project is designed with Cmake, a cross-platform tool to generate build systems. Further self-developed software is packed into applications offering a command line interface to either work with them by hand or embed them into Cmake. Nevertheless, applications are mostly interfaces only and relevant functionality is built into multiple shared libraries.

Each library comes with code documentation and a set of tests. The entire stack of libraries consists of fifty pages of code documentation and over seventy test cases for critical parts of the software. The code documentation itself is part of the toolchain, thus if you integrate the generated code documentation into your IDE you can write on it and straightforwardly use it within your IDE help suite. However, this section will focus more on applications rather than library functionality. If you want to read more about the libraries refer to the developer guide or code documentation on the CD.

The importer initializes the database used internally to share call tree dependent data between apps. Therefore, every application includes a common interface to connect to databased by command line or reading the configuration from a created cache file. Beside database initialization the importer offers an extensible import routine to load call trees from arbitrary file formats into the database. The standard importer comes with the support of a specific csv format.

The application *generate* is responsible to generate problem descriptions, call trees and seeds. First seeds should be generated used later to extend the build matrix. To generate call trees the application accepts a configuration file with a specified build matrix and a path to a javascript file. This javascript file is then called within an engine provided by the application for every possible configuration and the resulting call trees are uploaded into the database. The engine has an interface for c interoperability called packages and the software comes with two predefined, *ccalltree* and *crandom*. *ccalltree* is a low level interface to create a call tree within the surrounding

application. *crandom* extends the random functions that comes with javascript to a set of more reliable distributions. Furthermore, *crandom* guarantees to be correctly seeded to the corresponding configuration seed value. *ccalltree* is very low level and generally only consists of push and pop operations. In order to work properly with the generator then engine includes a javascript library named *calljs*. This libary implements a generator as described within this thesis and maps the created call tree to the c interface. Furthermore, the interface is exchangable and *calljs* can also print the resulting call tree, contracted or uncontracted, into a latex file. This is very handy for test or debugging purposes. To use the empowered javascript engine without the need to upload the generated call tree into the database the engine can be started seperately using the application *jsconsole*.

Generated call tree can then be solved using the *solve* application. This application uses heuristics and cmpl to solve problem descriptions loaded from the database. To do so it saves the problem description into da file and then starts cmpl for every IP formulation passed. Afterwards the results are uploaded back into the database. The software includes two cmpl implementations for the standard IP formulation and the extended formulation with result checkpointing.

For analyzation the application *analyze* is used. This application loads all call trees and arrange them into the build matrix they were created from. In order to save the entire call tree within each cell the application prepares a set of statistics which can be extended by an interface. If a problem descriptions and solutions exist for a call tree, its corresponding cell is extended by a matrix with two dimensions. One for the used solver and another for the chosen memory bound with cells containing statistics of the solving step. The entire set of matrices can then be either stored into a matlab file or the application itself starts matlab as an interactive shell. Additionally, the software comes with a light set of matlab functions called *callmat* containing useful functions to plot relevant data.

The entire software stack uses multiple abstraction layers making it easy to hook into every part wihtout the need of a deep understanding of every single class. Moreover, the automated test suite and toolchain will give instant feedback for every customization and preserved dependencies will automatically keep the state of each module consistent. Regardless the software is written with this thesis in mind thus applications may be a bit tailored but this is the minor part of the software and major components are shared by libraries making it simple to implement own applications. Actually there is a library to fastly implement custom applications called *runtime*.

This section gives only a short introduction but further information an explanation

can be obtained by the user/developer guide and code documentation on the attached CD.

7 Conclusion

The Call Tree Reversal Problem, short CTR, is a special case of the DAG reversal problem applied to the call tree structure and proven to be NP-complete[3] with respect to the DAG reversal problem. This thesis introduces a formulation of CTR using integer programs. The result is more customizable then custom heuristics and delivers a good inside view into the different states during a reversal. Thus the formulation not only delivers an alternative problem description further it explains dependencies of structural properties of call trees and eases a deeper understanding of call tree reversals. Additionally, pitfalls are elaborated and solved using result checkpointing. Both developed versions are analyzed with respect to expectations within elaboration resulting in a continuous transition from theoretical composition to practice. Further an implementation of the IP formulation for CTR were analyzed showing fundamental characteristics of result checkpointing. Unfortunately using result checkpointing slows the solver down to an unacceptable performance. However, result checkpointing shows promising results under certain circumstances but suffer from memory peaks of parent nodes.

The analyzation focus on very primitive data structures and shows dependencies between different characteristic and their influence on result checkpoint utilization. This yields results which can be further adopted to a wide range of more complex structures to build more in depth researches or inspect application specific structures and works fine with the call tree construction. The call tree constructions showed a mathematical concept of building arbitrary call trees using a unified structure which can be repeatedly nested into each other. Due to the primitive example within analyzation this was not extensively used but makes it easier to integrate the used structures into custom abstractions and analyze the impact with respect to the prepared characterization. Additionally, call tree generation comes with a stack of techniques used to adopt real world applications and map existing program characteristics like call graphs or low and high level programming features.

The included toolchain makes it fluent to customize the research in this thesis to custom needs at any level. Interleaving the set of provided tools result in a fully automated system preserving dependencies and giving instant feedback for any change. The usage of four different languages(c++, javascript, sql, matlab) and extensive use of abstraction layers makes it hard to understand the entire stack in depth. To overcome this the project includes a vagrant file to setup a fully feature enabled virtual machine, except commercial licensed software, able to build the entire software stack including this thesis and a user plus developer guide. On the other hand, the mix of several languages and tools makes it much easier to focus on the part necessary using

a well suited language for the focused problem.

8 Outlook

This section lists topics I discovered being out of scope for this thesis and gives impulses for future related work.

Call Tree Generation

Grammars

One common approach to randomly generate structured objects is the usage of a grammar definition. I neglect this option because I wanted to have a more application related representation but this representation could be used to be adapted by grammar. This would open a wide range of theoretic- and practically resources.

Extraction

Every construction within this thesis introduces an own configuration based on observations and analyzation of real world applications or tools used to create those. Another approach could be to take existing structures and algorithmic extract a construction configuration. Those structure could then be used and especially mixed up with user defined abstractions to create more practical related data or analyze application specific circumstances.

Minors

The developed construction structure with contraction operation comes with minors in mind. Minors can be used to find and express graph characterizations but the thesis does not utilize this in analyzation. Thus it would be interesting how available characterizations like claw-free graphs influences out coming results or how new application specific characterizations could be defined to guarantee desired behavior. For example, structures worked out in this thesis could guarantee a good utilization of result checkpointing or at least used to help the IP to find those faster.

IP Formulation

Meantime Result Checkpointing

The actual implementation always assumes to store all result checkpoints from the beginning but the analysis showed for a growing number of joints the memory peak moves up building a bottleneck. Therefore, result checkpoints could be recorded after bottlenecks are passed and maybe released before the according subroutine is reached.

This would enable result checkpointing for subtrees and maybe counter the problem of loops.

Hybrid Solutions

Both IP formulations are strictly separated within this thesis but a mixed solution could try to find a balance between performance and result checkpointing utilization. Additionally, heuristics could be used internally to obtain a basic solution.

Solver Parameter

This thesis heavily focuses on the outcome and utilization of result checkpointing but neglects performance improvements using specific solvers and parameters. A field of interest might be the usage of analyzation to choose convenient parameters enhancing the result and performance.

Just In Time Generation

The IP implementation is currently static but it may be suitable to use characteristics and analyzations obtained by heuristics, matlab and so forth to generate an adapting problem definition. For example, result checkpointing could be enabled only for subtrees with predefined properties or the graph could transform into a simpler one.

Analyzes

Random Inspection

This thesis uses a conceptual approach analyzing structures. However, random structures could be created and tested against properties of interest. Passing call trees could then be used to take statistical inference.

References

[1] Uwe Naumann Johannes Lotz, Sumit Mitra, *A 0,1-programming approach to the call tree reversal problem*, Ph.D. thesis, 2016.

[2] Uwe Naumann, *The art of differentiating computer programs: An introduction to algorithmic differentiation*, Software, Environments, and Tools, SIAM.

[3] ———, *Call tree reversal is np-complete*, Tech. Report AIB-2007-18, Department of Computer Science, RWTH Aachen, December 2007.