Proceedings Paper:
Plump, DETLEF and ISMAILI ALAOUI, ZIAD (2026) Implementing Binary Search Trees in GP 2 (Extended Abstract). In: Proceedings 16th International Workshop on Graph Computation Models (GCM 2025). Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 6-12.

https://doi.org/10.4204/EPTCS.440.2

Implementing Binary Search Trees in GP 2 (Extended Abstract)

Ziad Ismaili Alaoui
University of Liverpool, United Kingdom

Detlef Plump
University of York, United Kingdom

1 Introduction

We present an approach to implement binary search trees in the rule-based graph programming language GP 2. (See [4] for a brief introduction to GP 2.) Our implementation uses GP 2's rooted graph transformation rules to be fast [1] and supports insertion, deletion and query operations. We argue that the worst-case runtime for each of the operations is $O(n)$ for a tree with n nodes. In addition, we expect that, on average, the operations run in time $O(\log n)$[1]. Hence the implementation would match the time complexity of binary search trees implementations in imperative languages (see, for example, [6]).

2 Binary Search Trees

A *binary search tree* (BST) is a binary tree in which each node is labelled with a distinct key (i.e. an integer value). For every node v, all keys in its left subtree are strictly smaller than the key of v, while all keys in its right subtree are strictly greater than the key of v.[2] Figure 1 shows an example of a binary search tree with 6 nodes. Binary search trees typically allow for three operations: insertion, querying,

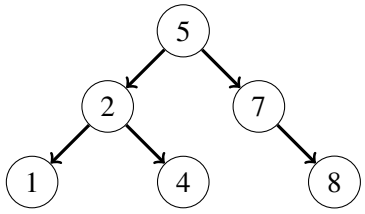

Figure 1: Example of a binary search tree.

and deletion. We provide a high-level explanation as to how these operations work.

Insertion. Inserting a node into a BST must preserve its properties. To insert a new key, we begin at the root and recursively traverse the tree: at each node, we move to the left child if the new key is less than or equal to the current node's key, or to the right child if it is greater. This process continues until we reach a null child position, where the new node is inserted. This ensures that the BST property remains valid after insertion.

[1] Here and throughout, $\log = \log_2$.

[2] Note this implies that BSTs do not have repeated entries.

L. Lambers and O. Semeráth (Eds.): GCM 2025
EPTCS 440, 2026, pp. 6–12, doi:10.4204/EPTCS.440.2

Querying. Searching for a key in a BST proceeds similarly to insertion. Starting at the root, we recursively compare the target key with the current node's key: if they match, the search is successful; if the target key is smaller, we search the left subtree; if larger, the right subtree. The search continues until the key is found or the search cannot proceed farther, indicating that the key is not present in the tree.

Deletion. Similarly to insertion, deleting a node from a BST must also preserve its properties. There are three possible cases to handle:
(1) If the node to be deleted is a leaf, it can simply be removed, leaving its parent with one less child.
(2) If it has one child, the node is removed and its child is connected directly to its parent.
(3) If it has two children, its key is replaced with the maximum key in its left subtree, and that replacement node (which always is a leaf or has one child) is then deleted. To find the maximum key in the left subtree, we start at the left child and repeatedly move to the right child until no further right child exists; this also ensures that the replacement node does not have two children, in case of duplicate elements. This operation naturally preserves the ordering of the elements of the tree.

In a conventional programming language, these operations can be implemented to run in time $O(n)$ in the worst case for a BST with n nodes. (This is when the tree degenerates into a linked list.) If the BST is *balanced*, that is, when the heights of the child subtrees of any node only differ by at most 1, these operations run in time $O(\log n)$ (see [6]).

3 The Program

The programming language GP 2 uses the notion of so-called *roots*, which are distinguished nodes that are stored independently from non-rooted ones. To avoid confusion, in this section, we refer to the root of the binary tree (i.e. the top element) as the *top*.

The program `bst` (Figures 2 and 3) implements a binary search tree in GP 2 whose top is pointed at by a green node, which acts as a parent to the top, and follows the specification below.

Input: A linked list of unmarked nodes and edges such that

1. each node label represents exactly one user instruction (insertion, query and deletion);
2. nodes labelled `"i":`n for some `int` n represent an instruction to insert n into the BST;
3. nodes labelled `"s":`n for some `int` n represent an instruction to search for a node labelled n, should one exist, in the BST and create a dashed edge from the instruction node to the found node in the BST;
4. nodes labelled `"d":`n for some `int` n represent an instruction to turn a node labelled n in the BST, should one exist, into a garbage node[3];
5. the first instruction is the head of the linked list and rooted; and
6. there exists an edge from u to v if and only if u denotes the ith instruction and v, the $(i+1)$th.

Output: A graph consisting of the linked list provided by the user, where the tail is rooted instead, a green node, and a binary search tree whose nodes are grey, edges are unmarked, and whose top is pointed at by the green node and constructed in accordance with the order of instructions specified by the user and the procedures described in Section 2.

[3] A *garbage node* is a non-instruction node that is not reachable from the green node.

```
Main   = make_root; (try insert then Insert else (
                     try search then Search else (
                     try delete then Delete else fail));
                     try next_op else break)!
Insert = try root  then ({go_right1, go_left1}!;
                   if match then break else add_leaf) else add_root
Search = try root  else break; {go_right1, go_left1}!; try match
Delete = try root  else break; {go_right1, go_left1}!;
         try Case1 else (try Case2 else Case3)
Case1  = try delete_leaf else fail
Case2  = try delete_midl else fail
Case3  = save_node; go_left2; if go_right2
         then (go_right2!; {swap1, swap2, swap3, swap4})
         else {swap5, swap6}; try Case1 else Case2
```

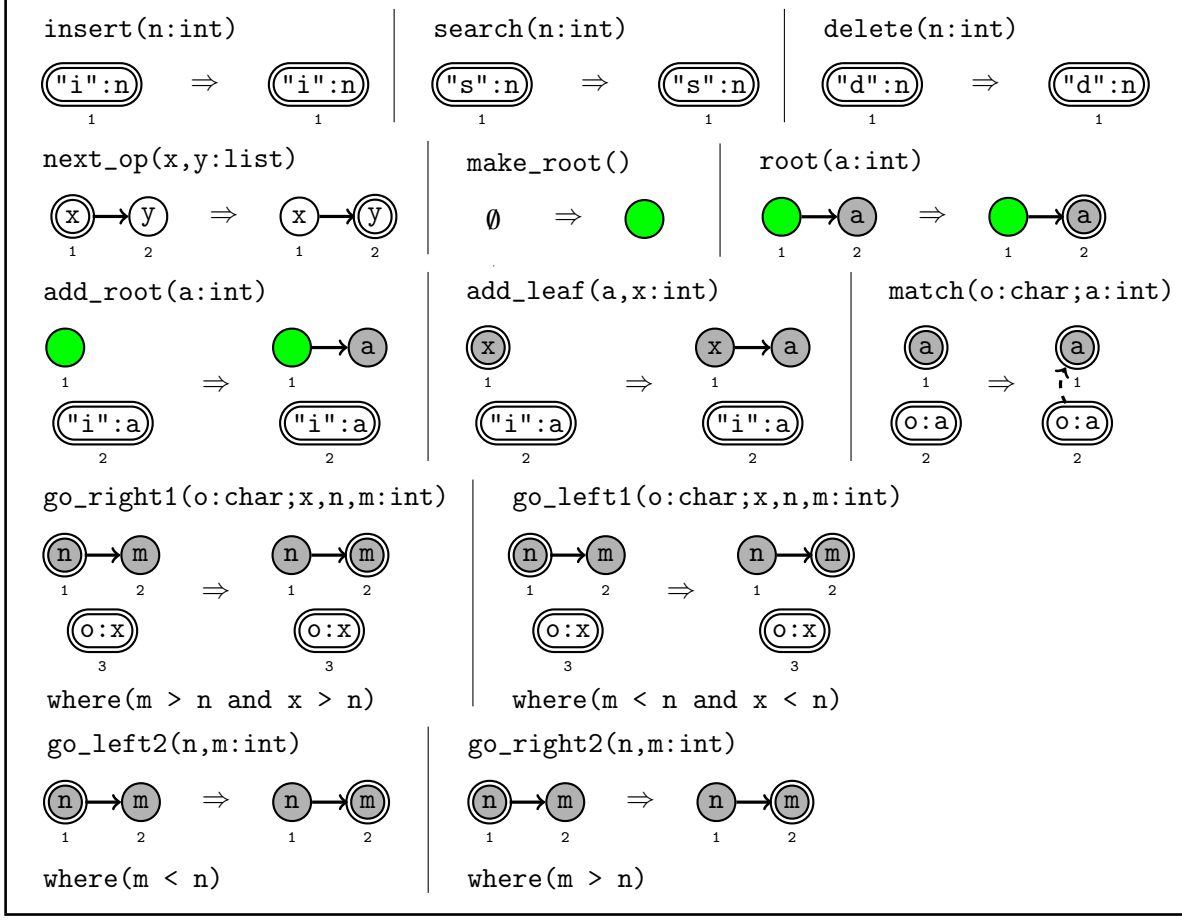

Figure 2: The program `bst`; the rest of the rules are graphically depicted in Figure 3.

We refer to the nodes specified by the user as *instruction nodes*. The program first applies the rule `make_root` to create a green dummy node whose function is to serve as a parent to the top of the BST;

this technique allows for simpler rule design. The program iterates through the user-given instructions by performing a traversal of the linked list provided as input. Throughout the execution of the program, an instruction node whose instruction is currently in execution is rooted; otherwise, it is not. We describe the behaviour of the program for each operation, modularised as procedures (`Insert`, `Search` and `Delete`), below.

Insertion (`Insert`). The program first attempts to root the top of the BST by calling the rule `root`. If none exists (i.e. the tree is empty), the rule fails to apply and `add_root` is applied instead, which creates a top to the tree. If `root` successfully applies, meaning that at least one node is in the tree, the rule set `{go_right1, go_left1}` is called as long as possible.[4] Essentially, each rule moves the root down the tree until none can apply, which happens if and only if the root reaches a node of strictly less than two children, and the new key in the rooted instruction node can be added as a child to the rooted node in the BST. The insertion is done by the application of the rule `add_leaf`. Given that the BST cannot hold duplicate elements, the rule `match` is tried each time the root is moved at some position and terminates the loop as soon as a node of existing key exists in the tree.

Querying (`Search`). This procedure works in a way analogous to `Insert`. The program first attempts to root the top of the BST by calling `root`. If that rule fails, the procedure terminates as there are no nodes in the tree. The traversal remains similar; as soon as none of the rules in the rule set `{go_right1, go_left1}` is applicable, either the element was found or a dead end was reached. The rule `match` is then called and creates a dashed edge from the rooted instruction node to the rooted node in the BST holding the sought-for key should they match.

Deletion (`Delete`). Similarly, an application of the rule `root` is attempted; a failure simply terminates the procedure as there would be no node to delete. Then, the node holding the key to be delete, should it exist, is rooted by performing a search analogous to that of previous procedures down the tree. Should that node be found, three different cases of deletion apply, as described in Section 2. If the node to be deleted is a leaf, the rule `delete_leaf` applies. If it is a one-child node, the rule `delete_midl` applies. Otherwise (i.e. the node has two children), the rule `save_node` is applied to keep track of that node; then, the root is moved to the left child, and go downward to the right child as long as possible. Eventually, the rooted node in the tree contains the largest key of the child left subtree (LKCLS) of the node to be deleted. The keys are swapped by reconnecting the edges; this is done by the rules prefixed `swap` (there are 6 distinct cases depending on many levels down the LKCLS is), and one of the other two cases now applies to the node to be deleted. Note that nodes are not actually deleted from the tree; indeed, they only become disconnected from the BST, that is, garbage nodes. The insertion of a previously deleted key does not affect garbage nodes (i.e. there could be multiple garbage nodes of the same key).

Claim 1 *The insert, delete and query operations run in time* $\mathrm{O}(n)$*, where n is the number of BST nodes.*

The claim above follows from the fact that all rules match and apply in constant time, and that the BST is acyclic so that the loop `{go_right1, go_left1}!` always terminates after at most n iterations, where n is the number of BST nodes.

[4] The edges of the binary search tree need not be explicitly labelled, as each edge's direction (left or right) can be inferred directly by comparing the values of its endpoint nodes. This does not affect the complexity stated in Claim 1.

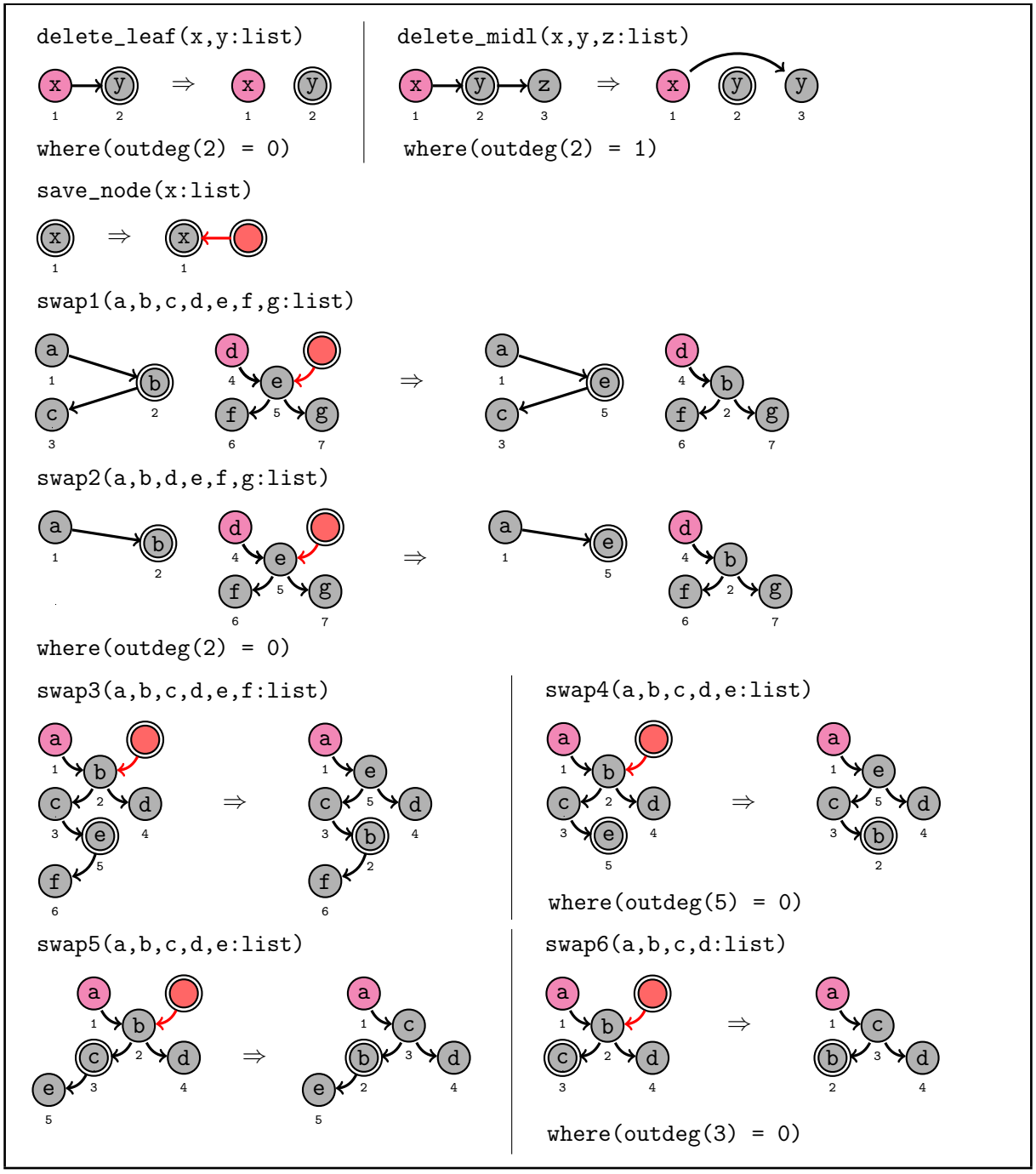

Figure 3: Remaining rules of program bst.

4 Runtime Measurements

We provide runtime measurements of the program `bst`[5] in Figures 4 and 5 (obtained using a quad-core Intel i5 clocked at 2.4 GHz, with 16 GB RAM, running 64-bit Ubuntu 22.04). In Figure 4, only degenerate trees (that is, linked lists) are considered, and the time required to perform a single operation is recorded. For each tree size, the operation was executed 300 times, and the results were averaged. Each operation takes longer on a linked list because performing any update or search requires traversal of all existing vertices. In contrast, on a balanced tree of the same size, such operations require visiting at most $\log n$ vertices, which is significantly fewer. This difference explains why operations are much faster in the second plot. In Figure 5, only fully balanced trees are considered. Each measurement records the time required to insert an element into a leaf of the tree, search for it, and subsequently delete it. As the runtime of each individual operation on balanced trees was too small to measure reliably, we consider the three operations together to obtain a more stable and representative measurement, and as with the degenerate trees, each sequence of operations was repeated 300 times, and the results were averaged.

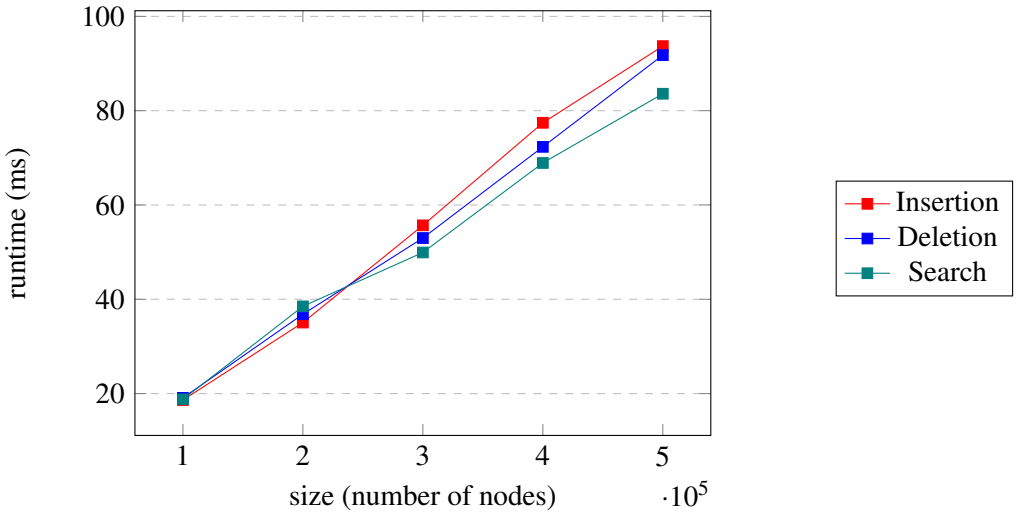

Figure 4: Measured runtime per operation on degenerated trees.

5 Conclusion and Future Work

We have shown how to implement binary search trees in the rule-based language GP 2 such that searching, inserting, and deleting entries requires linear time in the worst-case. The next step will be to establish, both experimentally and analytically, that the operations run in time $O(\log n)$ on average.

Another topic for future work is to implement some form of *balanced* binary search trees, such as red–black trees, where the operations preserve balance. For such a structure, the query, insert, and delete operations are guaranteed to run in time $O(\log n)$. An implementation of red–black trees in the graph transformation language PROGRES is discussed in [2], where the height of the left and right subtrees of a node are allowed to differ up to a factor of 2. We also mention [3] as a graph transformation approach to formally verify the insertion operation of red–black trees.

[5]Concrete syntax available at: `https://github.com/UoYCS-plasma/GP2/blob/master/programs/bst.gp2`.

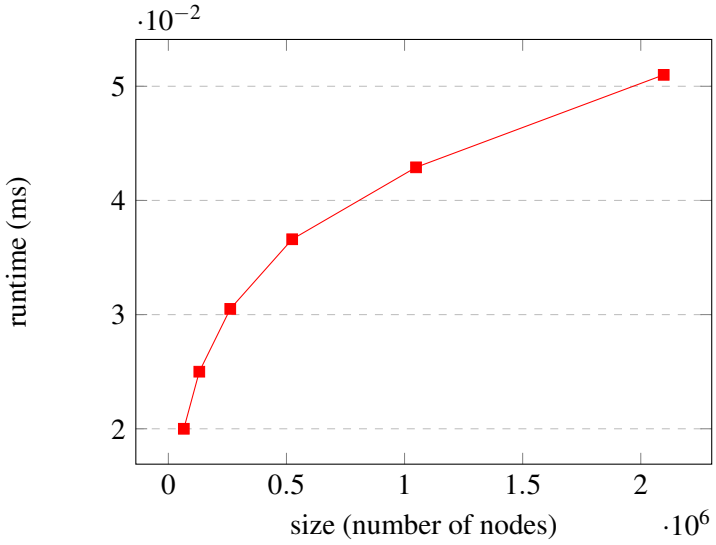

Figure 5: Measured averaged runtime for an insertion, followed by a search and deletion, on balanced trees.

An alternative approach to data structures in GP 2 could be to efficiently implement them in an imperative language such as C and provide access operations in GP 2. However, we argue that for graph structures such as binary search trees, the code of operations is easier to read and understand in a rule-based graph transformation language. Moreover, writing both application programs and data structure implementations in GP 2 allows formal program verification with, for example, the methods described in [5, 7].

References

[1] Ziad Imaili Alaoui & Detlef Plump (2025): *Rule-Based Graph Programs Matching the Time Complexity of Imperative Algorithms*. Technical Report, University of York, doi:10.48550/ARXIV.2501.09144. 40 pages.

[2] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr & Gabriele Taentzer (1999): *Graph Transformation for Specification and Programming*. *Science of Computer Programming* 34(1), pp. 1–54, doi:10.1016/S0167-6423(98)00023-9.

[3] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König & Vitali Kozioura (2005): *Verifying Red-Black Trees*. In: *Workshop on Verification of Concurrent Systems with Dynamic Allocated Heaps (COSMICAH 2005)*, Queen Mary University of London, pp. 1–15. Informal proceedings.

[4] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs*. *Science of Computer Programming* 214, p. 102727, doi:10.1016/j.scico.2021.102727.

[5] Christopher M. Poskitt & Detlef Plump (2023): *Monadic Second-Order Incorrectness Logic for GP 2*. *Journal of Logical and Algebraic Methods in Programming* 130, p. 100825, doi:10.1016/j.jlamp.2022.100825.

[6] Steven S. Skiena (2020): *The Algorithm Design Manual*, third edition. Springer, doi:10.1007/978-3-030-54256-6.

[7] Gia Wulandari & Detlef Plump (2021): *Verifying Graph Programs with Monadic Second-Order Logic*. In: *Proc. 14th International Conference on Graph Transformation (ICGT 2021)*, *Lecture Notes in Computer Science* 12741, Springer, pp. 240–261, doi:10.1007/978-3-030-78946-6_13.