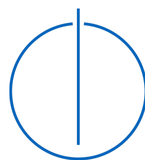
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Verification of Query Optimization Algorithms in Isabelle/HOL

Bernhard Stöckl



DEPARTMENT OF INFORMATICS

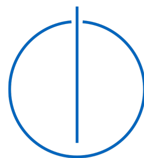
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Verification of Query Optimization Algorithms in Isabelle/HOL

Verifikation von Algorithmen zur Anfrageoptimierung in Isabelle/HOL

Author:	Bernhard Stöckl
Supervisor:	Prof. Tobias Nipkow
Advisor:	Lukas Stevens, M. Sc.
Submission Date:	March 15, 2022



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Langenbach, Bernhard Stöckl

Acknowledgments

I want to thank my supervisor, Prof. Tobias Nipkow, and my advisor, Lukas Stevens, for allowing me to work on this thesis without any prior experience of working with Isabelle. Furthermore, I want to thank Lukas for taking his time to provide me with regular feedback, answer questions, and give me helpful advice. I also want to thank my family for their support and proofreading this thesis.

Abstract

Query optimization is an important research area of database systems. This thesis formalizes some aspects of query optimization in the interactive theorem prover Isabelle/HOL. It includes a general framework consisting of the definitions of selectivities, query graphs, join trees, and cost functions. Furthermore, we implement the join ordering algorithm IKKBZ using these definitions. We use Isabelle/HOL to verify the correctness of these definitions and prove that IKKBZ produces an optimal solution within a restricted solution space.

Kurzfassung

Anfrageoptimierung ist ein wichtiges Forschungsgebiet im Bereich von Datenbanksystemen. Diese Arbeit formalisiert einige Aspekte der Anfrageoptimierung im interaktiven Theorembeweiser Isabelle/HOL. Es beinhaltet ein allgemeines Framework, das aus Definitionen von Selektivitäten, Query-Graphen, Join-Bäumen, und Kostenfunktionen besteht. Außerdem haben wir den Join-Optimierungsalgorithmus IKKBZ mit Hilfe dieser Definitionen implementiert. Wir benutzen Isabelle/HOL um die Korrektheit dieser Definitionen zu verifizieren und zu beweisen, dass IKKBZ eine optimale Lösung innerhalb eines eingeschränkten Lösungsraums findet.

Contents

Acknowledgments **v**

Abstract **vii**

Kurzfassung **ix**

1 Introduction **1**

2 Preliminiaries **3**
2.1 Related Work . . . 3
2.2 Isabelle/HOL . . . 3

3 Selectivities **7**

4 Query Graphs and Join Trees **9**
4.1 Query Graphs . . . 10
4.2 Join Trees . . . 12
4.2.1 Functions for Information Retrieval . . . 12
4.2.2 Functions for Correctness Checks . . . 14
4.2.3 Structures of Join Trees . . . 14
4.2.4 Creating Join Trees from Lists . . . 15
4.3 Combined Properties of Join Trees and Query Graphs . . . 15

5 Cost Functions **17**
5.1 Basic Cost Functions . . . 17
5.2 Properties of Cost Functions . . . 18

6 Extensions of Directed Trees **21**
6.1 General Additions to Directed Trees . . . 21
6.2 Dtree as an Algebraic Type for Directed Trees . . . 23
6.2.1 Well-Formed Dtrees . . . 24
6.2.2 Transformation from Dtree to Directed Tree . . . 25
6.2.3 Transformation from Directed Tree to Dtree . . . 26
6.2.4 Additional Dtree Functions . . . 28
6.3 List Dtrees . . . 30

7 IKKBZ **33**
7.1 IKKBZ Definitions and Correctness 33
7.1.1 IKKBZ-Sub 35
7.1.2 Complete IKKBZ 40
7.2 Optimality of IKKBZ 41
7.2.1 Conforming Sequences 41
7.2.2 Combining Preserves an Optimal Solution 45
7.2.3 Additional Invariants of IKKBZ-Sub 49
7.2.4 Optimality of IKKBZ-Sub and IKKBZ 55
7.3 Application of IKKBZ 60
7.3.1 A General Cost Function 60
7.3.2 Replacing List Based Input for Tree Queries 63
7.3.3 Application of IKKBZ on Simple Cost Functions 64
7.3.4 Application of IKKBZ on the C_{out} Cost Function 66

8 Conclusion **67**

Bibliography **69**

1 Introduction

Data in relational databases is often requested using query languages like SQL. These provide a high-level "declarative" interface to access the stored data without specifying a concrete execution of such a query [2, p. 34]. Therefore, an important problem of relational databases is query optimization. A major part of this research area is to find an optimal join order for a query. A join order refers to the order in which the relations of a query are combined to produce the desired result. Since different join orders can have different execution times while calculating the same query, this can have a significant impact on the performance. A join order is optimal if it has a minimal cost with respect to some cost function that measures the execution cost [8].

However, since the problem of finding an optimal join order is NP-complete in general, computing the optimal solution may not be feasible. Instead, there are heuristic algorithms (e.g. GOO) that generate results without any guarantees to their optimality. Other algorithms (e.g. IKKBZ) generate an optimal solution within a restricted solution space [3, 5, 7].

Since this topic has already been researched for several decades, a large variety of different algorithms has been developed. While their properties are usually proven, verifying these proofs can be a lot of work and making small errors is possible. Therefore, it is useful to use an interactive theorem prover like Isabelle[1] which machine-checks every step of a proof [13].

While the "Archive of Formal Proofs"[2] (AFP) already covers a wide range of different formalizations, query optimization is not one of them. Hence, the goal of this thesis is to implement the IKKBZ algorithm and a framework of basic query optimization theories that are necessary for this implementation.

Outline

The next chapter provides an overview of some related work and a short introduction to the notation of Isabelle/HOL.

In the Chapters 3 to 5 we define several general data structures which we need for query optimization. Chapter 3 covers selectivities and some related properties. In the following chapter, we develop definitions for query graphs and join trees. Based on these, we discuss cost functions in Chapter 5.

[1] `https://isabelle.in.tum.de/`
[2] `https://www.isa-afp.org/`

Chapter 6 provides some extensions to directed trees which we need to formalize IKKBZ. In Chapter 7, we define the IKKBZ algorithm, prove its optimality, and apply it to some example cost functions.

The last chapter summarizes the results of this thesis and gives a short overview of possible future extensions.

To keep this thesis short, we omit the concrete proofs in all of these chapters. Instead, we provide some key lemmas and an outline for the more interesting proofs. However, all definitions and their complete proofs can be found in their respective theory file[3]. We annotate the chapters with their corresponding theory files to the right of their title to make it easier to find these proofs. Furthermore, all definitions are focused on simplicity to make the proofs easier. However, they can be replaced by more efficient implementations by showing their equivalence.

[3] `https://gitlab.lrz.de/00000000014969F0/query-optimization`

2 Preliminiaries

In this chapter, we discuss some related work and explain the basic notation of Isabelle/HOL.

2.1 Related Work

Most of my knowledge about query optimization is taught in TUM's "Query Optimization" lecture [9] which was held by Neumann and Radke when I attended it. Especially the slides of chapter three [10] provide a good overview of join ordering techniques.

The paper "Building Query Compilers" [8] by Moerkotte also covers a variety of different topics related to query optimization.

"On the Optimal Nesting Order for Computing N-Relational Joins" [5] by Ibaraki and Kameda is one of the original papers that explain the IKKBZ algorithm. The other fundamental paper is "Optimization of Nonrecursive Queries" [7] by Krishnamurthy, Boral, and Zaniolo. The IKKBZ is named after the five authors of these two papers since they both led to this algorithm.

Of course, an important part of this thesis is Isabelle and its higher-order logic (HOL) library which was developed and extended by many different people [13].

The AFP's "Graph Theory" [12] entry by Noschinski extends the HOL library by basic graph-theoretic formalizations which we use for the graph-related definitions in this thesis.

The "Fast Diameter Estimation" [14] project developed by Stevens and Abdulaziz supplements this theory with further additions. This includes the definition of directed trees which we use for the IKKBZ algorithm.

2.2 Isabelle/HOL

Isabelle/HOL can be seen as a functional programming language that supports proofs about defined functions. It consists of a type system of base types, variables, functions, and type constructors. Definitions and proofs are stated using terms that are assigned such a type [11, p. 3].

New data types are defined by the **datatype** keyword. This may depend on type variables and requires one or multiple constructors for this new type. An example of such a datatype is the definition of a *list* which we will often use throughout this thesis. It uses the type variable *'a* and consists of two constructors [11, p. 8].

```
datatype 'a list = Nil | Cons 'a "'a list"
```

Furthermore, it introduces some additional operations that we omitted to keep this example simple. However, we will mostly use the syntactic sugar like "[]" for *Nil* and "#" for *Cons* when referring to lists [11].

Moreover, it is possible to use the **type_synonym** keyword to define aliases for a type. This can be used as an abbreviation or to give a more meaningful name to a type [11, p. 14].

To define functions, we will often use the **fun** keyword that allows recursive definitions and pattern matching for its input parameters. One restriction is that the termination of a function needs to be proven. However, for most functions, this is done automatically. For more complex preconditions or termination proofs, we use the **function** keyword which is quite similar to *fun*. For non-recursive definitions, there are the additional keywords **abbreviation** and **definition**. While there are some differences between these keywords, they are not important to understand this thesis [6, 11].

Proofs in Isabelle/HOL begin by stating the goal in a **lemma**, **theorem**, or **corollary**. We use the different names to indicate the importance of a proven statement, but there is no real difference between them. Often the goal is not some general statement but only holds if certain assumptions are satisfied. Therefore, Isabelle/HOL uses the "$\Longrightarrow$" symbol to separate goals and assumptions. Multiple assumptions can be chained together by these arrows. Furthermore, the symbols "$\llbracket$" and "$\rrbracket$" can be used to group assumptions and make the separation clearer. Moreover, the explicit keywords **assumes** and **show** can be used to make the distinction even more obvious. Another related keyword is **fixes** which allows to explicitly fix a free variable outside of the goal and assumptions. Furthermore, **defines** can be used to introduce a local abbreviation in the context of this proof [11].

While there are some small differences between the different ways of stating assumptions, these are again not relevant to understanding this thesis. Therefore, these three statements can be seen as equivalent [11]:

```
lemma sum_pos: "(n::int) > 0 ⟹ m ≥ 0 ⟹ n + m > 0"

theorem sum_pos: "⟦(n::int) > 0; m ≥ 0⟧ ⟹ n + m > 0"

corollary sum_pos:
  fixes n :: int and m
  defines "res ≡ n + m"
  assumes "n > 0" and "m ≥ 0"
  shows "res > 0"
```

Hence, it would be possible to commit to a single notation throughout this thesis. However, we still use these different notations to make definitions and proofs easily recognizable in their respective theory file. Since we omit the detailed proofs in this thesis, we also omit the explanations of how these work in Isabelle/HOL.

To deal with parametric theories, Isabelle uses **locale**s. A locale consists of fixed parameters and assumptions that can be used within all definitions and proofs in its context. Furthermore, it is possible to have a locale built on other locales by combining

them with a "**+**" symbol. The theorems defined in a locale's context can later be used by proving the assumptions and using the **interpretation** keyword [1].

An example locale is the definition of directed trees in the "Fast Diameter Estimation" project [14]. It extends the locale *wf_digraph* by fixing a root that is in the vertices. The *wf_digraph* locale expresses directed graphs as sets of *verts* and *arcs*. Furthermore, it uses a *head* and a *tail* function to express which vertices are connected by the arcs. These functions must be well-formed in the sense that there may only be arcs between vertices that are contained in the *verts* set. Moreover, all vertices in a directed tree must be reachable via a unique *awalk* from the root. The *awalk* property is satisfied if the list of edges p represents a valid arc walk from the first to the last parameter. Note that we use the terms arcs and edges interchangeably throughout this thesis.

```
locale directed_tree = wf_digraph T for T +
  fixes root :: 'a
  assumes root_in_T: "root ∈ verts T"
      and unique_awalk: "v ∈ verts T ⟹ ∃!p. awalk root p v"
```

3 Selectivities

Selectivities.thy

Usually, the information available to a query optimizer is limited to base relations. Hence, we need a way to calculate the cardinalities of intermediate join results. For this purpose, we introduce *selectivities*. As an example, consider two relations R_1 and R_2 with the cardinalities 10 and 20. If we know that a join predicate $p_{1,2}$ of these two relations has a selectivity of 0.2, we can use this information to calculate the result's cardinality. It is computed by multiplying the base cardinalities and the selectivity value [8, p. 34]:

$$|R_1 \bowtie_{p_{1,2}} R_2| = 10 \cdot 20 \cdot 0,2 = 40$$

Formally, a selectivity $f_{i,j}$ assigns a real value in the range $[0,1]$ to a join predicate $p_{i,j}$. It can be combined with the cardinalities of the base relations to calculate the cardinality of a join result. Selectivities are defined as the result cardinality divided by the cardinality of the Cartesian product [5, p. 484][8, p. 34]:

$$f_{i,j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i| \times |R_j|}$$

If there is no join predicate between two relations, joining them results in a cross product of the input relations. This means that all possible combinations are computed and none are discarded. Therefore, it can be seen as a join with a selectivity of one. Hence, it is possible to assign a selectivity to all pairs of relations and we represent them by a function that maps two relations of an arbitrary type to a real value [8, p. 34]:

```
type_synonym 'a selectivity = "'a ⇒ 'a ⇒ real"
```

One important property of selectivities is that selectivities should be symmetric since $R_i \bowtie R_j = R_j \bowtie R_j$ holds for inner joins [8, p. 34]:

```
definition sel_symm :: "'a selectivity ⇒ bool" where
  "sel_symm sel = (∀x y. sel x y = sel y x)"
```

The second definition requires that all selectivities are in the range $(0,1]$. We exclude zero because it is required for some proofs. For example, the proof that a certain cost function satisfies the ASI property which we explain in Chapters 5.2 and 7.3 relies on positive cardinalities. Therefore, the selectivities between all relations must be positive as well [7, p. 132]. Furthermore, since a selectivity of zero would mean that the result is empty, computing a join order for these cases does not make sense anyway. Moreover, selectivities in real applications are only estimated, since correctly calculating them according to the previous definition would already require the output cardinality. Hence,

an estimated selectivity of zero can be replaced by a small positive value. Therefore, we do not lose any relevant cases by excluding zero [8, p. 34].

```
definition sel_reasonable :: "'a selectivity ⇒ bool" where
  "sel_reasonable sel = (∀x y. sel x y ≤ 1 ∧ sel x y > 0)"
```

For queries with more than two relations, it is necessary to compute the selectivity of joining compound relations that consist of already joined relations. Hence, we define *set_sel_aux* which computes the selectivity of joining a set of relations with a single relation and *set_sel* which computes the selectivities of joining two sets of relations. We also define an equivalent definition *set_sel′* which changes the order of recursion. However, since the definitions are quite similar we only include one of them here:

```
definition set_sel_aux :: "'a selectivity ⇒ 'a ⇒ 'a set ⇒ real" where
  "set_sel_aux sel x Y = (∏y ∈ Y. sel x y)"
```

```
definition set_sel :: "'a selectivity ⇒ 'a set ⇒ 'a set ⇒ real" where
  "set_sel sel X Y = (∏x ∈ X. set_sel_aux sel x Y)"
```

Since every relation that appears within a query should be joined exactly once, it is possible to define an equivalent definition based on distinct lists. This simplifies some proofs and has additional applications for IKKBZ since we only consider left-deep trees which are uniquely identified by a list of relations as we show in Chapter 4.2.4. Similar to the set-based selectivity calculation functions, we define two different sets of functions. However, this time we show the definitions for the other order of recursion:

```
fun list_sel_aux' :: "'a selectivity ⇒ 'a list ⇒ 'a ⇒ real" where
  "list_sel_aux' sel [] y = 1"
| "list_sel_aux' sel (x#xs) y = sel x y * list_sel_aux' sel xs y"
```

```
fun list_sel':: "'a selectivity ⇒ 'a list ⇒ 'a list ⇒ real" where
  "list_sel' sel x [] = 1"
| "list_sel' sel x (y#ys) = list_sel_aux' sel x y * list_sel' sel x ys"
```

While it would be possible to use some folding function to define these functions, we prefer these recursive definitions for their simplicity. Furthermore, by proving the equivalence to a possible folding-based definition we can use all the related proofs about these as well.

It follows by some simple inductions that the list- and set-based functions are equivalent if the inputs are distinct lists and their respective set representation. Furthermore, some important characteristics of these functions are that they preserve the symmetry and reasonability properties of their input selectivity:

```
lemma list_sel_reasonable:
  "sel_reasonable f ⟹ list_sel f x y ≤ 1 ∧ list_sel f x y > 0"
```

```
lemma list_sel_symm: "sel_symm f ⟹ list_sel f x y = list_sel f y x"
```

4 Query Graphs and Join Trees

QueryGraph.thy, JoinTree.thy

In this chapter, we cover query graphs which present a convenient representation of a query [8, p. 32]. Furthermore, we define join trees as a way of expressing different join orders [8, p. 33]. Throughout this chapter, we use the database schema shown in Table 1 to illustrate the definitions with some examples. It contains information about students and which lectures they attend. On this schema, we define the example query shown in Figure 1. The query finds all students that attend the same lecture as a student called "Fichte".

attend	
StudNr	LectureNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
25403	5022
29555	5022
29555	5001

students		
StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Table 1: Example relations based on the uni schema [4]

```sql
SELECT s2.*
FROM students s1, attend a1, attend a2, students s2
WHERE s1.Name = 'Fichte'
  AND s1.StudNr = a1.StudNr
  AND a1.LectureNr = a2.LectureNr
  AND a2.StudNr = s2.StudNr;
```

Figure 1: Example SQL query on the schema from Table 1

4.1 Query Graphs

A query graph is an undirected graph where the nodes are the relations of a query and the edges are join predicates [8, p. 32]. Figure 2 shows the query graph for the example query shown in Figure 1. In this query graph, we can see that every relation in the *FROM* clause has its own node labeled with its name, identifier, and cardinality. For example, the lower right node represents the *students* relation with identifier *s2* and a cardinality of eight. Furthermore, all predicates are represented as edges annotated with their corresponding predicate and an approximated selectivity.

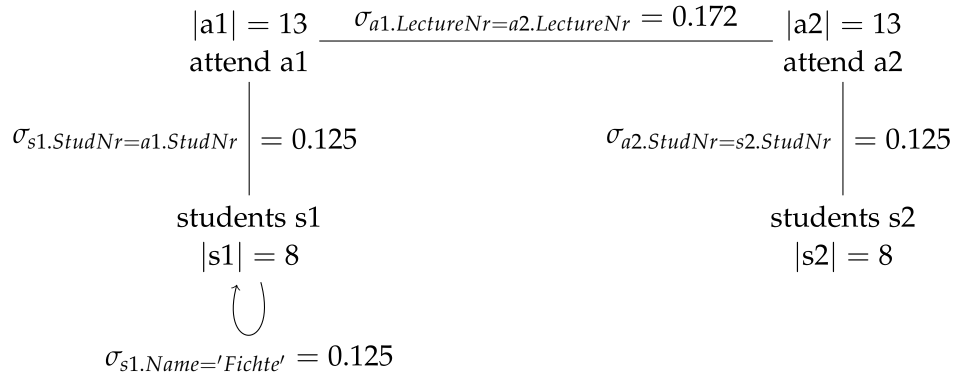


Figure 2: Query graph of the example query from Figure 1

However, we do not need all of that information for the problem of finding a join order: It is sufficient to label the nodes with their identifier and cardinality. Similarly, the only information we need about the edges is their selectivity. Furthermore, selections (i.e. self-edges) can be pushed down and processed before join ordering. Hence, it is sufficient to consider a simplified query graph as shown in Figure 3 [8].

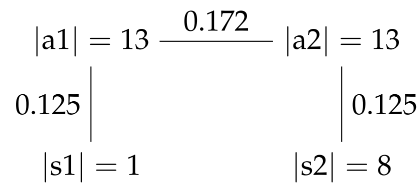


Figure 3: Simplified query graph

In our implementation, we use the AFP's graph theory [12] as a basis for query graphs. Additionally, we fix a weight function that annotates the edges with their selectivity and a cardinality function that assigns each relation to its cardinality. This way we have all the information necessary for join ordering stored in the query graphs. For relations, we use the arbitrary type *'a* to keep the definition general. A possible instantiation would be a string since identifiers in queries are represented by strings. However, it is often the case that integers are used instead for performance reasons.

```
locale query_graph = graph +
  fixes sel :: "'b weight_fun"
  fixes cf :: "'a ⇒ real"
  assumes sel_sym:
    "⟦tail G e1 = head G e2; head G e1 = tail G e2⟧ ⟹ sel e1 = sel e2"
      and not_arc_sel_1: "e ∉ arcs G ⟹ sel e = 1"
      and sel_pos: "sel e > 0"
      and sel_leq_1: "sel e ≤ 1"
      and pos_cards: "x ∈ verts G ⟹ cf x > 0"
```

The *sel_sym* condition is necessary to ensure the symmetry of the selectivities since undirected graphs are represented as bidirected graphs in the underlying graph theory. The second condition *not_arc_sel_1* requires that non-existing arcs have a selectivity of one since that means that they are interpreted as a cross product. The last three conditions ensure that the selectivities and cardinalities have reasonable values as discussed in Chapters 3 and 4.2.2.

Since the selectivity depends on the edges which makes it a bit impractical, we define a function to determine an equivalent selectivity using the type introduced in Chapter 3. We define it by using the *THE* operator to apply *sel* on the unique edge between two relations. If no such edge exists, we know that there is no predicate between the nodes. Hence, the result must be a cross product with a selectivity of one.

```
definition match_sel :: "'a selectivity" where
  "match_sel x y =
    (if ∃e ∈ arcs G. (tail G e) = x ∧ (head G e) = y
    then sel (THE e. e ∈ arcs G ∧ (tail G e) = x ∧ (head G e) = y)
    else 1)"
```

We transfer the locale assumptions by introducing the notion of a *matching_sel*. A selectivity f matches the *sel* function if they produce the same result for all pairs of arcs and the connected nodes. Furthermore, all pairs without arcs need to have a selectivity of one.

```
definition matching_sel :: "'a selectivity ⇒ bool" where
  "matching_sel f = (∀x y.
      (∃e. (tail G e) = x ∧ (head G e) = y ∧ f x y = sel e)
      ∨ ((∄e. (tail G e) = x ∧ (head G e) = y) ∧ f x y = 1))"
```

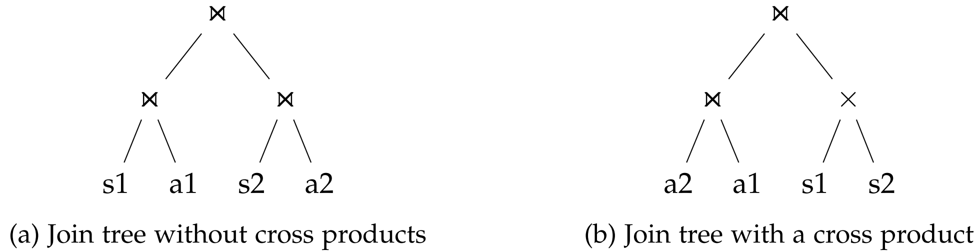
With this definition, we can show that any function that satisfies this property fulfills the basic symmetry and reasonability constraints. Furthermore, since all matching selectivities need to agree on all inputs, it follows that all of them must be equal. Finally, we apply the results to *match_sel* by showing that it satisfies the *matching_sel* property.

```
corollary match_sel_symm: "sel_symm match_sel"

corollary match_sel_reasonable: "sel_reasonable match_sel"
```

4.2 Join Trees

Join trees are binary trees in which the leaves are relations and inner nodes are joins or cross products. They represent possible join orderings by following them bottom-up. Figure 4 shows some example join trees for the usual query. While the first one is free of cross products, the second one does have a cross product since there is no join predicate between *s1* and *s2* [8, p. 33].



(a) Join tree without cross products

(b) Join tree with a cross product

Figure 4: Example join trees for the query from Figure 1

We define join trees as a new datatype that only represents the structure while cardinalities and selectivities are handled separately for more flexibility. This allows us to use the cost and selectivity functions we defined for query graphs.

```
datatype (relations:'a) joinTree
  = Relation 'a | Join "'a joinTree" "'a joinTree"

type_synonym 'a card = "'a ⇒ real"
```

4.2.1 Functions for Information Retrieval

One function that is included in the datatype definition is the *relations* function. It is of type *'a joinTree ⇒ 'a set* and returns the set of all relations in a join tree. While the *inorder* function provides a list of relations created by an inorder traversal of a join tree, *revorder* returns the reverse of that list. The last operation that includes all relations in a join tree is the *relations_mset* function which returns a multiset containing every relation as often as it appears in its input join tree.

```
fun inorder :: "'a joinTree ⇒ 'a list" where
  "inorder (Relation rel) = [rel]"
| "inorder (Join l r) = inorder l @ inorder r"

fun relations_mset :: "'a joinTree ⇒ 'a multiset" where
  "relations_mset (Relation rel) = {#rel#}"
| "relations_mset (Join l r) = relations_mset l + relations_mset r"
```

Even though it is possible to define some of these collections based on others, these direct definitions are sometimes simpler to use. Therefore, we define them

by the above recursive functions and show the equivalences by additional lemmas like *reorder_eq_rev_inorder*:

```
lemma revorder_eq_rev_inorder: "revorder t = rev (inorder t)"
```

We use the *card* function to calculate the cardinality of the result of joining all relations as given by a join tree. As input it requires a cardinality function of type *'a card* which maps all relations to their cardinality. Furthermore, it takes a *'a selectivity* as defined in Chapter 3 and a *'a joinTree* to calculate its cardinality according to this formula [8, p. 35]:

$$|T_1 \bowtie T_2| = \left(\prod_{R_i \in T_1, R_j \in T_2} f_{i,j} \right) |T_1||T_2|$$

Figure 5 shows an example calculation of the cardinalities for the join tree that is shown in Figure 4b. It repeats the join tree on the left and shows the cardinalities on the right. The calculations according to the above formula are also included in the right tree. The cardinalities of the base relations and the selectivities are taken from the query graph shown in Figure 3.

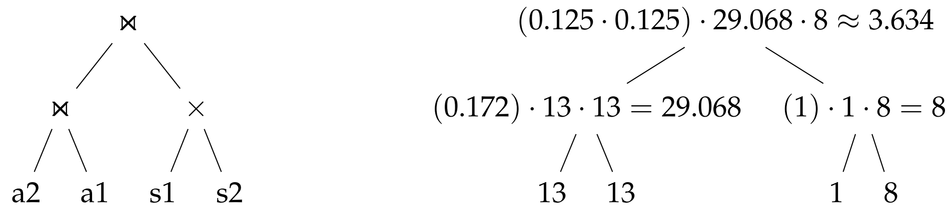


Figure 5: Example cardinality calculation for the join tree shown on the left

Since it requires the cardinalities of both join inputs to calculate the resulting cardinality, we use a recursive definition. Furthermore, we use *list_sel* to calculate the product of selectivities that is displayed in parentheses in the previous calculations.

```
fun card :: "'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ real" where
  "card cf f (Relation rel) = cf rel"
| "card cf f (Join l r) =
    list_sel f (inorder l) (inorder r) * card cf f l * card cf f r"
```

Since it is sometimes useful to know the height or number of relations in a join tree, we define the two functions *height* and *num_relations* to calculate these values. Finally, the *first_node* function returns the left-most node in a join tree. This node is the first node that is joined in left-deep trees which is why it is useful to be able to extract it from a join tree.

```
fun first_node :: "'a joinTree ⇒ 'a" where
  "first_node (Relation r) = r"
| "first_node (Join l _) = first_node l"

lemma first_node_eq_hd: "first_node t = hd (inorder t)"
```

4.2.2 Functions for Correctness Checks

The definitions in this section are used to check if a join tree satisfies certain properties. The *reasonable_cards* function checks whether all cardinalities are positive. Additionally, it requires all calculated cardinalities to be less or equal to the cardinality of the cross product. While it is clear that cardinalities should be non-negative numbers, the value zero is excluded for simplicity and because it would produce an empty result similar to a selectivity of zero. The second condition follows from the reasonability for subtrees combined with a valid selectivity which is less or equal to one for all pairs of relations.

```
fun reasonable_cards :: "'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ bool"
where
  "reasonable_cards cf f (Relation rel) = (cf rel > 0)"
| "reasonable_cards cf f (Join l r) = (let c = card cf f (Join l r) in
    c ≤ card cf f l * card cf f r ∧ c > 0
    ∧ reasonable_cards cf f l ∧ reasonable_cards cf f r)"
```

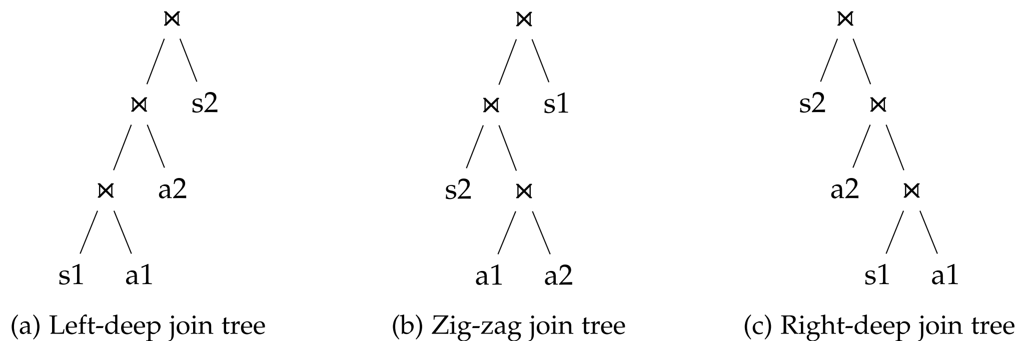
Another correctness property is that every relation should have a unique identifier that only appears once in a join tree. We define it using the *inorder* function but equivalent definitions based on *revorder* and *mset* are included as lemmas.

```
definition distinct_relations :: "'a joinTree ⇒ bool" where
  "distinct_relations t = distinct (inorder t)"
```

4.2.3 Structures of Join Trees

Join trees can have certain structures that are necessary for some join ordering algorithms. For example, the IKKBZ algorithm only considers left-deep trees in its solution space. Left-deep trees require that every inner node has a leaf as its right child while right-deep trees need the left child to be a leaf. Zig-zag trees are slightly less restrictive since the leaf nodes may be on either side of an inner node [8, p. 33].

Figure 6 shows examples of such join trees. All of them are zig-zag trees, but only the tree in Figure 6a is left-deep. Similarly, only the last join tree is right-deep.



(a) Left-deep join tree (b) Zig-zag join tree (c) Right-deep join tree

Figure 6: Example join trees for the query from Figure 1

While there exists another structure called "bushy", this is a property satisfied by all join trees. Therefore, a definition of bushy trees is superfluous. The following is the definition of left-deep join trees but all other functions are defined analogously.

```
fun left_deep :: "'a joinTree ⇒ bool" where
  "left_deep (Relation _) = True"
| "left_deep (Join l (Relation _)) = left_deep l"
| "left_deep _ = False"
```

4.2.4 Creating Join Trees from Lists

The functions in this section are used to create join trees from lists. The first function *create_rdeep* creates a right-deep join tree by recursion on the list of input relations. Similarly, *create_ldeep* constructs a left-deep tree from a given list of relations. However, since join trees can not be empty, they are both undefined for empty lists. Moreover, the *create_ldeep* definition uses an auxiliary function that works on reversed lists to create a tree with correct inorder traversal.

```
fun create_rdeep :: "'a list ⇒ 'a joinTree" where
  "create_rdeep [] = undefined"
| "create_rdeep [x] = Relation x"
| "create_rdeep (x#xs) = Join (Relation x) (create_rdeep xs)"

fun create_ldeep_rev :: "'a list ⇒ 'a joinTree" where
  "create_ldeep_rev [] = undefined"
| "create_ldeep_rev [x] = Relation x"
| "create_ldeep_rev (x#xs) = Join (create_ldeep_rev xs) (Relation x)"

definition create_ldeep :: "'a list ⇒ 'a joinTree" where
  "create_ldeep xs = create_ldeep_rev (rev xs)"
```

With these definitions, we can show that left-deep trees are uniquely represented by sequences of relations. To do so, we show that the *inorder* and *create_ldeep* functions are inverse operations. The only exception is that we need to exclude empty lists since join trees can not be empty. Of course, the same holds for right-deep trees as well.

```
lemma create_ldeep_order: "xs ≠ [] ⟹ inorder (create_ldeep xs) = xs"

lemma create_ldeep_inorder:
  "left_deep t ⟹ create_ldeep (inorder t) = t"
```

4.3 Combined Properties of Join Trees and Query Graphs

Finally, we show the relationship between query graphs and join trees by adding some definitions in the context of a query graph. First, we define the notion of *matching_rels*

which determines whether all relations in a join tree are contained in the query graph or not. This allows us to reason over such join trees with inductive proofs that would not be possible if equivalence was required.

```
definition matching_rels :: "'a joinTree ⇒ bool" where
  "matching_rels t = (relations t ⊆ verts G)"
```

With this definition, we can show that a join tree has reasonable cardinalities when using the fixed cardinality function and the matching selectivity.

```
corollary match_reasonable_cards:
  "matching_rels t ⟹ reasonable_cards cf match_sel t"
```

Moreover, we define *valid_tree* to determine if a join tree computes the query represented by a query graph. This means that every relation of the query graph has to appear exactly once in the join tree. Note that this does not forbid duplicate relations like *students* but only duplicates of concrete instances like *students s1* and *students s2*.

```
definition valid_tree :: "'a joinTree ⇒ bool" where
  "valid_tree t = (relations t = verts G ∧ distinct_relations t)"
```

Finally, we define a function to decide if a join tree is free of cross products. This is useful because some algorithms only consider join trees without cross products. For example, IKKBZ guarantees to find an optimal solution within left-deep join trees without cross products as we prove in Chapter 7.

Since a cross product occurs when joining two relations without a join predicate and these are represented by edges in the query graph, a cross product occurs when there is no edge between the relations of two joined subtrees. Therefore, a join tree without cross products always has to contain an edge from the left to the right subtree since query graphs are undirected.

```
fun no_cross_products :: "'a joinTree ⇒ bool" where
  "no_cross_products (Relation rel) = True"
| "no_cross_products (Join l r) =
    ((∃x ∈ relations l. ∃y ∈ relations r. x →G y)
    ∧ no_cross_products l ∧ no_cross_products r)"
```

A consequence of this property is that all relations in a join tree are reachable in the query graph by only visiting relations that are contained in the join tree. However, we need the additional condition *matching_rels* to ensure that it holds for join trees consisting of a single relation as well.

```
lemma no_cross_awalk:
  "⟦matching_rels t; no_cross_products t;
      x ∈ relations t; y ∈ relations t⟧
    ⟹ ∃p. awalk x p y ∧ set (awalk_verts x p) ⊆ relations t"
```

5 Cost Functions

CostFunctions.thy

Cost functions associate join trees with a real value that indicates the cost of executing a query in the given join order. This chapter explains some exemplary cost functions and proofs of certain relevant properties.

5.1 Basic Cost Functions

The first cost function is the C_{out} function which sums up all output cardinalities of the joins in a join tree. It is defined by the following recursive equation [8, p. 35]:

$$C_{out}(T) = \begin{cases} 0 & \text{if } T \text{ is a single relation} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

The definition in Isabelle/HOL is directly derived from this equation and uses the *card* function described in Section 4.2.1 to calculate the cardinality of a join:

```
fun c_out :: "'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ real" where
  "c_out _ _ (Relation _) = 0"
| "c_out cf f (Join l r) =
    card cf f (Join l r) + c_out cf f l + c_out cf f r"
```

While the C_{out} function relates output cardinality to the execution cost, the remaining definitions in this section are designed to capture the cost of specific implementations of a join. However, they all have in common that the cost of a complete join tree is determined by the sum of the costs of all joins occurring in it [8, p. 36].

As the name suggests, the C_{nlj} function estimates the cost of nested loop joins. Therefore, the cost of a single join is defined by the product of the cardinalities of its input [8, p. 36]:

$$C_{nlj}(e_1 \bowtie_p e_2) = |e_1||e_2|$$

```
fun c_nlj :: "'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ real" where
  "c_nlj _ _ (Relation _) = 0"
| "c_nlj cf f (Join l r) =
    card cf f l * card cf f r + c_nlj cf f l + c_nlj cf f r"
```

The third cost function aims to provide an estimation for hash joins. It assumes a constant cost to find all matches of an element of the left input in the right input. Therefore, it assumes that the right input is already stored in a hash table and is only viable for left-deep trees. Otherwise, intermediate results would need to be precomputed

which is not realistic. Furthermore, we assume that the average length of a collision chain is 1.2 which leads to the following definitions [8, p. 36]:

$$C_{hj}(e_1 \bowtie_p e_2) = 1.2|e_1|$$

```
fun c_hj :: "'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ real" where
  "c_hj _ _ (Relation _) = 0"
| "c_hj cf f (Join l r) = 1.2 * card cf f l + c_hj cf f l + c_hj cf f r"
```

Another possible join implementation is the use of a sort merge join which operates by sorting both inputs and then performing a linear scan of both sorted inputs. Hence the cost for large inputs is bounded by the cost of sorting which leads to these definitions [8, p. 36]:

$$C_{smj}(e_1 \bowtie_p e_2) = |e_1|log(|e_1|) + |e_2|log(|e_2|)$$

```
fun c_smj :: "'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ real" where
  "c_smj _ _ (Relation _) = 0"
| "c_smj cf f (Join l r) =
    card cf f l * log 2 (card cf f l) + card cf f r * log 2 (card cf f r)
    + c_smj cf f l + c_smj cf f r"
```

5.2 Properties of Cost Functions

In this section, we discuss some additional properties of cost functions. The first property is **symmetry** which is satisfied if a cost function returns the same result when the join partners are swapped [8, p. 38]:

```
definition symmetric :: "('a joinTree ⇒ real) ⇒ bool" where
  "symmetric f = (∀x y. f (Join x y) = f (Join y x))"
```

All of the example cost functions except for C_{hj} fulfill this property. However, the C_{out} function requires that the selectivity function is symmetric. Since this should be the case for valid selectivities anyway, it is not a real restriction [8, p. 38].

```
lemma c_out_symm: "sel_symm f ⟹ symmetric (c_out cf f)"

lemma c_nlj_symm: "symmetric (c_nlj cf f)"

lemma c_smj_symm: "symmetric (c_smj cf f)"
```

The second property is the **adjacent sequence interchange (ASI)** property. Intuitively, it means that the relationship of the cost of two sequences is completely represented by another function that only depends on smaller subsequences. As the name suggests, the representation only needs to hold if these subsequences are adjacent and exchanging them transforms one complete sequence into the other one. This allows us to argue about the optimality of a complete sequence by only using local information of smaller

parts. Formally, the ASI property is satisfied by a function f if there exists a rank function r_f such that $f(AS_1S_2B) \leq f(AS_2S_1B)$ holds for any sequences A, B and any nonempty sequences S_1, S_2 if and only if $r_f(S_1) \leq r_f(S_2)$ [5, p. 495].

The definition in Isabelle/HOL has some additional requirements that are necessary to prove that a function satisfies the ASI property. However, valid sequences always fulfill these conditions. Therefore, they do not restrict the applicability of the property. The first additional requirement is that the sequence needs to be distinct which is already required for a sequence to be valid. The second condition is that if the root r is contained in the sequence, it needs to be the first relation. Since we choose the root as the first element this is satisfied for valid sequences as well.

```
definition asi :: "('a list ⇒ real) ⇒ 'a ⇒ ('a list ⇒ real) ⇒ bool"
where
  "asi rank r c = (∀A U V B. distinct (A@U@V@B) ∧ U ≠ [] ∧ V ≠ []
      ∧ (r ∉ set (A@U@V@B)
        ∨ (take 1 (A@U@V@B) = [r] ∧ take 1 (A@V@U@B) = [r]))
      ⟶ (c (rev (A@U@V@B)) ≤ c (rev (A@V@U@B))
        ⟷ rank (rev U) ≤ rank (rev V)))"
```

Since this property operates on list-based cost functions, we need list-based functions that are equivalent for left-deep trees. In Chapter 7.3 we define such a general cost function called *c_list* and prove that it satisfies this ASI property. Furthermore, we show how to instantiate the parameters of this function such that we get equivalent list-based definitions for the C_{out}, C_{nlj}, and C_{hj} cost functions. Hence, these three example cost functions satisfy the ASI property.

6 Extensions of Directed Trees

The IKKBZ algorithm relies on a so-called precedence graph which is a query graph interpreted as a directed tree by choosing one vertex as the root [8, p. 50][5, p. 494]. To formulate the algorithm in Isabelle/HOL, we use the notion of a directed tree as defined in the "Fast Diameter Estimation" project [14]. This chapter introduces some additions to these directed trees that are used in Chapter 7 to define and prove some properties of IKKBZ.

6.1 General Additions to Directed Trees

Directed_Tree_Additions.thy

To formalize IKKBZ, we need additional definitions and lemmas for directed trees. First of all, we define finite directed trees as a combination of the *directed_tree* and *fin_digraph* locales. The second locale requires that the vertices and arcs are both finite sets.

```
locale finite_directed_tree = directed_tree + fin_digraph T
```

On the one hand, we need this constraint for the algebraic type *dtree* that we introduce in Section 6.2. On the other hand, we do not lose any expressiveness concerning *graph*s since they are always finite per definition in the AFP's graph theory. Therefore, this condition is satisfied by all *query_graph*s and is not a restriction.

To execute the IKKBZ algorithm, we need to transform a graph into a directed tree. However, since defining such a transformation algorithm is outside the scope of this thesis, we assume that such a conversion function exists. A possible implementation could be breadth-first search (BFS). Hence, we create a *bfs_tree* locale which assumes that a directed tree T is a subgraph of a graph G. Furthermore, T should contain exactly those vertices that are reachable from the vertex that was chosen as the root.

```
locale bfs_tree = directed_tree T root + subgraph T G for G T root +
  assumes root_in_G: "root ∈ verts G"
      and all_reachables: "verts T = {v. root →*G v}"
```

We use this locale to assume the existence of such a conversion function. Its parameters are a well-formed directed graph and a vertex that should be chosen as the root.

```
locale bfs_locale =
  fixes bfs :: "('a, 'b) pre_digraph ⇒ 'a ⇒ ('a, 'b) pre_digraph"
  assumes bfs_correct:
      "⟦wf_digraph G; r ∈ verts G; bfs G r = T⟧ ⟹ bfs_tree G T r"
```

Since IKKBZ works on acyclic query graphs, we define undirected trees as connected graphs in which $|arcs| \leq 2 \cdot (|verts| - 1)$ holds. This is based on the fact that the number of edges in a tree is the number of vertices minus one. However, since graphs are modeled as bidirected graphs, the number of arcs is doubled. Furthermore, we weaken the condition to less or equal since it allows instantiating this locale more easily. We can assume that our query graph is connected because IKKBZ does not consider cross products which means that the query graph must be connected [8, p. 49].

```
locale undirected_tree = graph +
  assumes connected: "connected G"
      and acyclic: "card (arcs G) ≤ 2 * (card (verts G) - 1)"
```

We combine this locale with the transformation from a graph to a directed tree to prove that our *undirected_tree* is really an acyclic graph.

```
locale undir_tree_todir = undirected_tree G + bfs_locale bfs
  for G :: "('a, 'b) pre_digraph"
  and bfs :: "('a, 'b) pre_digraph ⇒ 'a ⇒ ('a, 'b) pre_digraph"
```

In this context, we create an abbreviation that instantiates the first parameter of the *bfs* function with the fixed undirected graph:

```
abbreviation dir_tree_r :: "'a ⇒ ('a, 'b) pre_digraph" where
  "dir_tree_r ≡ bfs G"
```

With these definitions we first prove that the generalization in the *acyclic* assumption implies the equality of both sides. Furthermore, we prove that the arcs of a directed tree combined with the reverse arcs are equal to the set of all edges in a graph:

```
lemma arcs_compl_un_eq_arcs:
  "r ∈ verts G ⟹
    {e2 ∈ arcs G. ∃e1 ∈ arcs (dir_tree_r r).
        head G e2 = tail G e1 ∧ head G e1 = tail G e2}
      ∪ arcs (dir_tree_r r) = arcs G"
```

Moreover, we show that there exists a unique path between every pair of arcs in the graph and that every path that starts at a vertex r is contained in the directed tree rooted at r. These lemmas can be seen as proof that the definition of undirected trees is correct.

```
lemma unique_apath: "⟦u ∈ verts G; v ∈ verts G⟧ ⟹ ∃!p. apath u p v"
```

```
lemma apath_in_dir_if_apath_G:
  "apath r p v ⟹ pre_digraph.apath (dir_tree_r r) r p v"
```

Since IKKBZ combines nodes during the normalization phase, it makes sense to use lists of relations to represent these compound nodes [7, p. 134]. However, converting a query graph into a directed tree results in a tree with relations as vertices. Therefore, we introduce a definition that transforms a directed tree into a tree that consists of singleton lists. We do this by using the image operation on the vertices and wrapping the head

and tail functions in anonymous functions. The image operation is represented by a backtick ` and applies a function to all elements of a set.

```
definition to_list_tree :: "('a list, 'b) pre_digraph" where
  "to_list_tree =
    (|verts = (λx. [x]) ` verts T, arcs = arcs T,
      tail = (λx. [tail T x]), head = (λx. [head T x])|)"
```

Furthermore, as we discuss in more detail in Section 6.3, some operations of IKKBZ require additional assumptions to preserve well-formedness. These are that all vertices are pairwise disjoint and nonempty. Therefore, we fix these in a locale that extends finite directed trees. It follows directly from the definitions that a finite directed tree converted with the *to_list_tree* function satisfies all conditions of this locale.

```
locale fin_list_directed_tree =
  finite_directed_tree T for T :: "('a list,'b) pre_digraph" +
  assumes disjoint_verts:
    "⟦u ∈ verts T; v ∈ verts T; u ≠ v⟧ ⟹ set u ∩ set v = {}"
      and nempty_verts: "v ∈ verts T ⟹ v ≠ []"
```

6.2 Dtree as an Algebraic Type for Directed Trees

Dtree.thy

This section defines an algebraic type that is equivalent to a finite directed tree. Having this type simplifies the definitions and proofs required for the IKKBZ algorithm. Even though it is equivalent only for **finite** directed trees, this is sufficient for our purposes since a *query_graph* is always finite per definition. *Dtree*s are defined as a recursive datatype with a root of type *'a* and a finite set (*fset*) of sucessors or children. The set consists of pairs of a *('a,'b) dtree* and an *'b* arc that connects this subtree to its parent *root*:

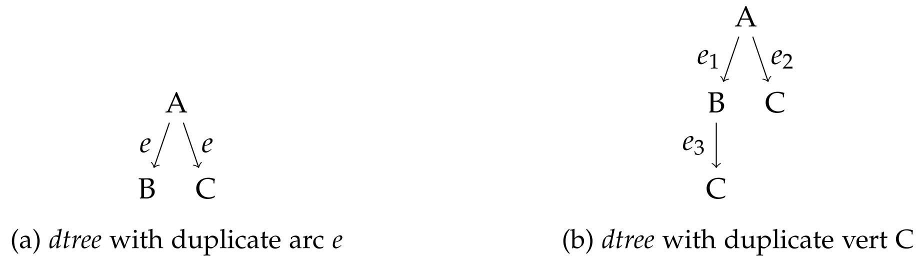
```
datatype (dverts:'a, darcs: 'b) dtree
  = Node (root: 'a) (sucs: "(('a,'b) dtree × 'b) fset")
```

The definition directly contains the functions *dverts* and *darcs* that return the set of all vertices or arcs contained in a *dtree*. Furthermore, the functions *root* and *sucs* provide access to the components of a *dtree*. Note that even though the definition requires the use of an *fset*, we will often convert them to "normal" sets because they have nicer syntax and are more convenient to use. While, the *fset* function converts a *fset* into a *set*, the *Abs_fset* does the inverse transformation. However, to properly use *Abs_fset*, its input must be a finite set. In general, *fset* operations are represented as the *set* counterparts surrounded by vertical lines (e.g. |`| instead of `).

Since recursive functions are required to terminate in Isabelle/HOL, we introduce a lemma called *dtree_size_decr* which automates the termination proof. It states that the size of a successor is less than the size of its parent node.

6.2.1 Well-Formed Dtrees

An arbitrary *dtree* may not be transformable into a valid directed tree. Therefore, we introduce additional constraints that a well-formed *dtree* has to satisfy. The first condition is that each arc can only appear once in a *dtree*. Otherwise, it would be unclear where an arc should lead to. For example "*Node A {(Node B {},e), (Node C {},e)}*" which is the *dtree* shown in Figure 7a has the arc *e* leading from *A* to both *B* and *C*. Therefore, it would not be possible to define a head function which is necessary for a directed tree.



(a) *dtree* with duplicate arc *e*

(b) *dtree* with duplicate vert C

Figure 7: Examples of *dtree*s that are not well-formed

Hence our first condition is that all arcs should be unique within a *dtree*. We express this by building the multiset of arcs and requiring that every arc appears exactly once. Isabelle/HOL uses the "#" symbol to distinguish between multiset and set notation.

```
fun darcs_mset :: "('a,'b) dtree ⇒ 'b multiset" where
  "darcs_mset (Node r xs) = (∑(t,e) ∈ fset xs. {#e#} + darcs_mset t)"

definition wf_darcs :: "('a,'b) dtree ⇒ bool" where
  "wf_darcs t = (∀x ∈# darcs_mset t. count (darcs_mset t) x = 1)"
```

Similarly, the vertices of a *dtree* need to be unique as well. Otherwise, the *dtree* shown in Figure 7b would be valid even though it violates the condition that every vertex should be reachable by a unique path from the root. The implementation also contains equivalent recursive definitions that do not use multisets which simplifies most proofs. However, we omit these here since they are not important to understand the definitions.

```
fun dverts_mset :: "('a,'b) dtree ⇒ 'a multiset" where
  "dverts_mset (Node r xs) = {#r#} + (∑(t,e) ∈ fset xs. dverts_mset t)"

definition wf_dverts :: "('a,'b) dtree ⇒ bool" where
  "wf_dverts t = (∀x ∈# dverts_mset t. count (dverts_mset t) x = 1)"
```

Finally, we summarize these conditions in a locale of well-formed *dtree*s:

```
locale wf_dtree =
  fixes t :: "('a,'b) dtree"
  assumes wf_arcs: "wf_darcs t"
      and wf_verts: "wf_dverts t"
```

6.2.2 Transformation from Dtree to Directed Tree

To transform a *dtree* into a directed tree, we need sets of vertices and arcs as well as two functions that map arcs to their tail and head. Since we already have the sets given by *dverts* and *darcs*, we only need to define functions that correctly map edges to a head and tail. If an arc is included in the successors of a node, the tail should be the root of this node. Otherwise, we recursively proceed in the subtree which contains this edge. If an arc should not be contained at all, we map it to the value of some default function which we will assume as a parameter. We use the *ffold* function to iterate over the *fset* of successors. It works similar to folding functions for lists and has three parameters: The first is a function *f* which is applied to an element of the set and the accumulator. The second parameter is an initial value for this accumulator and the final one is an *fset* to iterate over.

```
fun dtail :: "('a,'b) dtree ⇒ ('b ⇒ 'a) ⇒ 'b ⇒ 'a" where
  "dtail (Node r xs) def = (λe. if e ∈ snd ` fset xs then r
    else (ffold (λ(x,e2) b.
      if (x,e2) ∉ fset xs ∨ e ∉ darcs x ∨ ¬wf_darcs (Node r xs)
      then b else dtail x def) def xs) e)"
```

Note that the anonymous function which is used as the first argument of the *ffold* only proceeds recursively if the input is contained in the set of children *xs* and the arcs are well-formed. The first condition may seem redundant, but it is necessary because the termination proof does not use the information that recursive calls are only done for children of this node. Moreover, well-formedness is required since the lemmas about the *ffold* function are only available for functions whose functional composition is commutative:

```
locale comp_fun_commute =
  fixes f :: "'a ⇒ 'b ⇒ 'b"
  assumes comp_fun_commute: "f y ∘ f x = f x ∘ f y"

lemma dtail_commute:
  "comp_fun_commute (λ(x,e2) b.
        if (x,e2) ∉ fset xs ∨ e ∉ darcs x ∨ ¬wf_darcs (Node r xs)
        then b else dtail x def)"
```

We define the *dhead* function analogously by mapping an arc to the root of the subtree it forms a pair with. The additional conditions allow us to prove commutativity for this function as well.

```
fun dhead :: "('a,'b) dtree ⇒ ('b ⇒ 'a) ⇒ 'b ⇒ 'a" where
  "dhead (Node r xs) def = (λe. (ffold (λ(x,e2) b.
      if (x,e2) ∉ fset xs ∨ e ∉ (darcs x ∪ {e2}) ∨ ¬wf_darcs (Node r xs)
      then b else if e=e2 then root x else dhead x def e) (def e) xs))"
```

Finally, we put all four functions together to obtain a directed tree. Since both *dtail* and *dhead* require a function with default values for arcs not contained in the *dtree*, we set these as parameters. One possible instantiation of these default functions is to map everything to the root of the *dtree* that should be converted.

```
abbreviation from_dtree
  :: "('b ⇒ 'a) ⇒ ('b ⇒ 'a) ⇒ ('a,'b) dtree ⇒ ('a,'b) pre_digraph"
where
  "from_dtree deft defh t ≡
    (|verts = dverts t, arcs = darcs t,
      tail = dtail t deft, head = dhead t defh|)"
```

In the context of *wf_dtree* we can show that this conversion does indeed produce a finite directed tree rooted at the root of the *dtree*:

```
theorem from_dtree_fin_directed:
  "finite_directed_tree (from_dtree dt dh t) (root t)"
```

6.2.3 Transformation from Directed Tree to Dtree

The definitions and proofs in this section are all in the context of *finite_directed_tree* since we need the finiteness to be able to transform a directed tree into an equivalent *dtree*. For our transformation we use an auxiliary function that converts the subtree rooted at a vertex r into an equivalent *dtree*. While the root of the node is set to the vertex r, we define the successors as the outgoing arcs of r and the subtrees rooted at the head of these arcs.

```
function to_dtree_aux :: "'a ⇒ ('a,'b) dtree" where
  "to_dtree_aux r = Node r (Abs_fset {(x,e).
    (if e ∈ out_arcs T r then x = to_dtree_aux (head T e) else False)})"
```

Since this function is recursive, we need to show its termination. We use the sets of reachable vertices to do so: Since all vertices reachable in the subtree of a child of r are reachable from r as well, no new elements are added to the set. Furthermore, r itself is not reachable from its child. Otherwise, T would not be a directed tree. Therefore, the cardinality of the set of reachable vertices decreases from parent to child. While it may seem equivalent to use "$P \wedge Q$" instead of an "`if P then Q else False`" construct, they actually generate different goals for the termination proof. Hence, we need this if-clause to be able to prove the termination of this function. Otherwise, the assumption that e is an outgoing arc of r is missing and we could not use this lemma:

```
lemma child_card_decr:
  assumes "e ∈ out_arcs T r"
  shows "Finite_Set.card {x. (head T e) →*T x}
        < Finite_Set.card {x. r →*T x}"
```

To transform a complete directed tree, we only need to call this auxiliary function with the *root* of the tree:

```
definition to_dtree :: "('a,'b) dtree" where
  "to_dtree = to_dtree_aux root"
```

We show that the auxiliary function produces a well-formed *dtree* by verifying the conditions *wf_darcs* and *wf_dverts*. Since *to_dtree* only uses this auxiliary function, we know that it produces a well-formed *dtree* as well.

```
theorem wf_to_dtree: "wf_dtree to_dtree"
```

Moreover, we prove that the *dverts* set of a *dtree* created by *to_dtree_aux r* is the set of reachable vertices if r is a vertex of T. Since the *root* is always a vertex and all vertices are reachable from the root, the set of *dverts* of a converted tree is exactly the set of vertices.

```
lemma dverts_eq_reachable:
  "r ∈ verts T ⟹ dverts (to_dtree_aux r) = {x. r →*T x}"

lemma dverts_eq_verts: "dverts to_dtree = verts T"
```

Finally, we show that the sets *darcs* and *arcs* are equal as well by using that every arc is an outgoing arc of some vertex and all outgoing arcs of a vertex r are in the *darcs* of *to_dtree_aux r*. By combining these two lemmas with the previous result that all vertices are contained in the dtree, we conclude that all arcs are contained in the *darcs* as well. Since every element contained in *darcs* is an outgoing arc and all of these are arcs, we can prove that the other direction holds as well. Therefore, both sets are equal:

```
lemma darcs_eq_arcs: "darcs to_dtree = arcs T"
```

With these lemmas, we can show that transforming from a *wf_dtree* to a directed tree and back results in the original *dtree*:

```
interpretation T: finite_directed_tree "from_dtree dt dh t" "root t"

theorem to_from_dtree_id: "T.to_dtree dt dh = t"
```

Furthermore, we define an abbreviation in the context of *finite_directed_tree* which uses the original *tail* and *head* functions as default values. This way, the values of *dtail* and *dhead* are guaranteed to be equal to the original functions even if they are called with an arc that is not contained in T. This allows us to show the identity of transforming from a directed tree to a *dtree* and then back to a directed tree.

```
abbreviation from_dtree :: "('a,'b) dtree ⇒ ('a,'b) pre_digraph" where
  "from_dtree t ≡ Dtree.from_dtree (tail T) (head T) t"

lemma from_to_dtree_eq_orig: "from_dtree (to_dtree) = T"
```

6.2.4 Additional Dtree Functions

In this subsection, we explain some other basic functions that we need to formalize IKKBZ. The first one is called *is_subtree* and determines whether one *dtree* occurs within another one. Based on these, we also define strict subtrees as subtrees that are not equal.

```
fun is_subtree :: "('a,'b) dtree ⇒ ('a,'b) dtree ⇒ bool" where
  "is_subtree x (Node r xs) =
    (x = Node r xs ∨ (∃(y,e) ∈ fset xs. is_subtree x y))"
```

We use this function in Chapter 7 to express properties that would otherwise require recursive definitions. The relation it represents is a partial order. This means that it is reflexive, transitive, and antisymmetric.

The next function, *num_leaves*, calculates the number of leaves in a tree. We will use it to prove the termination of some IKKBZ related functions in Chapter 7.

```
fun num_leaves :: "('a,'b) dtree ⇒ nat" where
  "num_leaves (Node r xs) =
    (if xs = {||} then 1 else (∑(t,e) ∈ fset xs. num_leaves t))"
```

Furthermore, we define a *max_deg* function which tells us the maximal (outgoing) degree of a *dtree*. Its recursive definition takes the maximum of the degrees of its children and its own successor cardinality. It uses the *Max* function which returns the maximum of a set of values and the *max* function which returns the maximum of two input values.

```
fun max_deg :: "('a,'b) dtree ⇒ nat" where
  "max_deg (Node r xs) =
    (if xs = {||} then 0
    else max (Max (max_deg ` fst ` fset xs)) (fcard xs))"
```

One possible alternative definition combines *max* and *Max* to a single *Max* call without case distinctions by inserting *fcard xs* into the set. Another equivalent nonrecursive definition is given by mapping the set of subtrees to their cardinality.

We can use this function to decide whether a *dtree* is a chain: If the maximum degree in a *dtree* is less than or equal to one, every vertex has at most one successor. Therefore, the *dtree* is actually a chain. This allows us to express such a *dtree* as a list. Hence, we introduce transformation functions that convert a *dtree* into a list and vice versa.

Since we only want to convert *dtree*s for which *max_deg t* ≤ 1 holds, we simplify our definition such that nodes that do not have exactly one child are treated as nodes without successors. Furthermore, we store edges and vertices in the list so that we can reconstruct a *dtree*. However, since the number of arcs is less than the number of vertices, we exclude the root of the *dtree*. This way the number of arcs and vertices is equal and every vertex can be stored as a pair with its incoming arc.

```
function dtree_to_list :: "('a,'b) dtree ⇒ ('a×'b) list" where
  "dtree_to_list (Node r {|(t,e)|}) = (root t,e) # dtree_to_list t"
| "∀x. xs ≠ {|x|} ⟹ dtree_to_list (Node r xs) = []"
```

To transform from list to *dtree*, we use the *dtree_from_list* function. Since the root is excluded from the conversion to a list, we need it as an additional parameter for this function. The list is transformed by setting the element of the node as the root parameter and the first pair as the only successor of this node:

```
fun dtree_from_list :: "'a ⇒ ('a × 'b) list ⇒ ('a,'b) dtree" where
  "dtree_from_list r [] = Node r {||}"
| "dtree_from_list r ((v,e)#xs) = Node r {|(dtree_from_list v xs, e)|}"
```

We show that these functions correctly preserve the sets of vertices and arcs. Of course, it is necessary to assume that the input of *dtree_to_list* is a chain since it will otherwise not process the complete *dtree*. The following are the lemmas for the arc sets, but the ones for vertices are quite similar.

```
lemma dtree_to_list_eq_darcs:
  "max_deg t ≤ 1 ⟹ snd ` set (dtree_to_list t) = darcs t"

lemma dtree_from_list_eq_darcs:
  "darcs (dtree_from_list r xs) = snd ` set xs"
```

Furthermore, we show the identity of the transformation that uses both functions and prove that *dtree_to_list* always produces a *dtree* with a maximum degree of less than or equal to one:

```
lemma dtree_from_to_list_id:
  "max_deg t ≤ 1 ⟹ dtree_from_list (root t) (dtree_to_list t) = t"

lemma dtree_to_from_list_id: "dtree_to_list (dtree_from_list r xs) = xs"

lemma dtree_from_list_deg_le_1: "max_deg (dtree_from_list r xs) ≤ 1"
```

To take advantage of list-based operations, we need to formulate the well-formedness properties of *wf_dtree*s for lists. This allows us to prove whether a function that alters a list preserves the well-formedness or not. We define the two properties using a recursive function for each of them. However, they are both equivalent to a definition that uses the *distinct* function.

```
fun wf_list_arcs :: "('a × 'b) list ⇒ bool" where
  "wf_list_arcs [] = True"
| "wf_list_arcs ((v,e)#xs) = (e ∉ snd ` set xs ∧ wf_list_arcs xs)"

fun wf_list_verts :: "('a × 'b) list ⇒ bool" where
  "wf_list_verts [] = True"
| "wf_list_verts ((v,e)#xs) = (v ∉ fst ` set xs ∧ wf_list_verts xs)"
```

We prove the correctness of these functions by showing that these properties hold if and only if the corresponding well-formedness property holds for a created *dtree*:

```
lemma wf_darcs_iff_wf_list_arcs:
  "wf_list_arcs xs ⟷ wf_darcs (dtree_from_list r xs)"
```

```
lemma wf_dverts_iff_wf_list_verts:
  "r ∉ fst ` set xs ∧ wf_list_verts xs
 ⟷ wf_dverts (dtree_from_list r xs)"
```

Finally, we show that these conditions hold initially when a well-formed *dtree* is transformed into a list:

```
lemma wf_list_arcs_if_wf_darcs:
  "wf_darcs t ⟹ wf_list_arcs (dtree_to_list t)"

lemma wf_list_verts_if_wf_dverts:
  "wf_dverts t ⟹ wf_list_verts (dtree_to_list t)"
```

6.3 List Dtrees

List_Dtree.thy

As mentioned before, IKKBZ works on directed trees with compound nodes that we represent by lists. However, when combining two nodes, our basic conditions of well-formedness are not sufficient to be preserved. For example, the *dtree* shown in Figure 8a satisfies all properties of the *wf_dtree* locale. Even so, combining the nodes *B* and *C* by appending their elements results in the *dtree* shown in Figure 8b. This clearly has a duplicate node *[B, C]* and is, therefore, not well-formed.

(a) *dtree* that satisfies all properties of *wf_dtree* (b) *dtree* after combining the nodes *B* and *C*

Figure 8: Example of a *dtree* that does not preserve well-formedness

The problem with the first *dtree* is that even though the nodes themselves are unique, the contents are duplicated. Moreover, during the execution of IKKBZ, we do not want duplicated relations. Therefore, this section explains stricter conditions for list-based *dtree*s. First, we define a function that returns the "real" set of elements in a list-based *dtree*. In this section, we use it to formulate our well-formedness condition, but we will also use it in Chapter 7 to show the correctness of properties concerning the set of all relations.

```
fun dlverts :: "('a list,'b) dtree ⇒ 'a set" where
  "dlverts (Node r xs) = set r ∪ (⋃x ∈ fset xs. dlverts (fst x))"
```

We prefer this recursive definition because it simplifies some proofs. However, it is equivalent to computing the union of all sets of *dverts*. In the well-formedness condition, we require pairwise disjointness of the *dlverts* of children and the content of their parent element. Additionally, we disallow empty lists, since they interfere with *wf_dverts* and should not appear during IKKBZ execution anyway.

```
abbreviation disjoint_dlverts :: "(('a list, 'b) dtree × 'b) fset ⇒ bool"
where
  "disjoint_dlverts xs ≡
    (∀(x,e1) ∈ fset xs. ∀(y,e2) ∈ fset xs.
      dlverts x ∩ dlverts y = {} ∨ (x,e1) = (y,e2))"
```

```
fun wf_dlverts :: "('a list,'b) dtree ⇒ bool" where
  "wf_dlverts (Node r xs) =
    (r ≠ [] ∧ (∀(x,e1) ∈ fset xs.
      set r ∩ dlverts x = {} ∧ wf_dlverts x) ∧ disjoint_dlverts xs)"
```

Since we excluded empty lists, two elements can only be equal if their sets are not disjoint. Hence, we can show that *wf_dlverts* implies *wf_dverts*:

```
lemma wf_dverts_if_wf_dlverts: "wf_dlverts t ⟹ wf_dverts t"
```

Finally, we use this new condition combined with the *wf_darcs* property to set up a new locale and show that they form a subset of *wf_dtree*s:

```
locale list_dtree =
  fixes t :: "('a list,'b) dtree"
  assumes wf_arcs: "wf_darcs t"
      and wf_lverts: "wf_dlverts t"
```

```
sublocale list_dtree ⊆ wf_dtree
```

We can now show equivalence to the *fin_list_directed_tree* locale by proving that the transformation functions called in the context of one locale result in a valid tree of the other locale. More concretely, this means that *list_dtree*s can be converted into *fin_list_directed_tree*s and vice versa.

```
theorem list_dtree_to_dtree: "list_dtree to_dtree"
```

```
theorem from_dtree_fin_list_dir:
  "fin_list_directed_tree (root t) (from_dtree dt dh t)"
```

Furthermore, similar to the previous section, we define the *wf_dlverts* property for lists as well, such that we can properly use our list transformations.

```
fun wf_list_lverts :: "('a list × 'b) list ⇒ bool" where
  "wf_list_lverts [] = True"
| "wf_list_lverts ((v,e)#xs) =
    (v ≠ [] ∧ (∀v2 ∈ fst ` set xs.
      set v ∩ set v2 = {}) ∧ wf_list_lverts xs)"
```

As in the previous section, we prove its correctness by showing the relationship between *wf_list_lverts*, *wf_dlverts*, and the two transformation functions. However, we need to add the condition for the root separately since the root node is not contained in the list used by the transformations.

```
lemma wf_list_lverts_if_wf_dlverts:
  "wf_dlverts t ⟹ wf_list_lverts (dtree_to_list t)"

lemma wf_dlverts_iff_wf_list_lverts:
  "(∀v ∈ fst ` set xs. set r ∩ set v = {}) ∧ r ≠ [] ∧ wf_list_lverts xs
    ⟷ wf_dlverts (dtree_from_list r xs)"
```

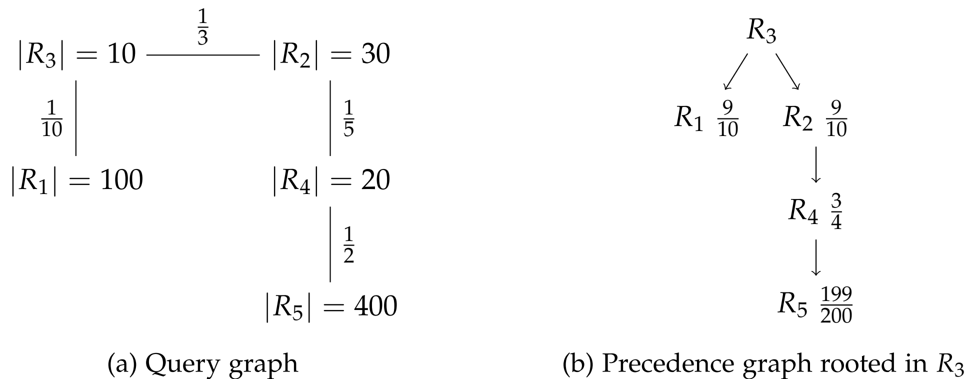
7 IKKBZ

In this chapter, we define an implementation of IKKBZ in Isabelle/HOL and prove its correctness. Moreover, we show that it produces an optimal solution within a restricted set of join trees. Finally, we show how to apply it to some example cost functions from Chapter 5. The definitions and proof ideas build on the corresponding chapter of the "Query Optimization" lecture [9, 10] and the "Building Query Compilers" paper [8]. Of course, these are based on the original IKKBZ papers [5, 7].

7.1 IKKBZ Definitions and Correctness

IKKBZ.thy

The general idea of the IKKBZ algorithm is to operate on the query graph interpreted as a directed tree by directing all edges away from a relation that is chosen as its root. Furthermore, the nodes are annotated with a rank that corresponds to the cost function according to the ASI property. Figure 9b shows such a so-called precedence graph rooted in R_3 for the example query graph shown in Figure 9a. While the annotated ranks correspond to the C_{out} cost function, their concrete calculation is not important right now and is explained in Section 7.3 [5, p. 497][7, p. 134].



(a) Query graph

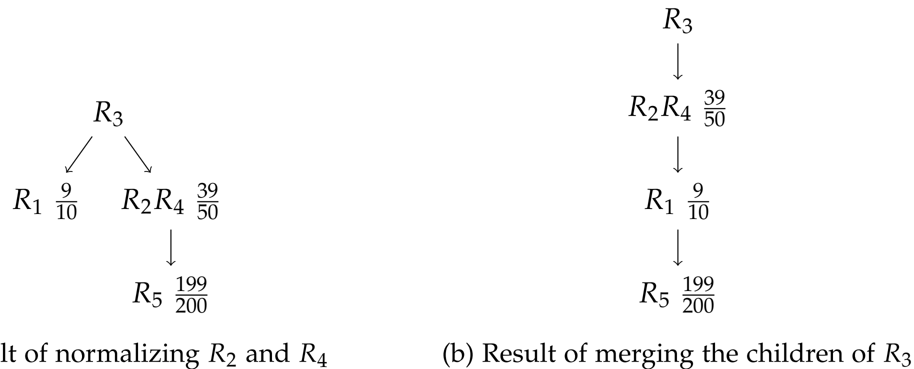
(b) Precedence graph rooted in R_3

Figure 9: Precedence graph rooted in R_3 for an example query graph

Inside such a precedence graph, IKKBZ looks for a node with more than one child and only chains as children. This means that all of the nodes in the subtrees of these children must have at most one successor. A node that satisfies both of these conditions is called a wedge. In our example, we can see that the root R_3 is a wedge since all of its (strict) subtrees are chains and it has two children. In these chains, we look for contradictory sequences that occur if the rank of a node is smaller than the rank of its

parent. In Figure 9b, we can see that there are contradictory sequences in the right chain since R_2 has a larger rank than R_4 but needs to be before R_4 according to the precedence graph [5, p. 497][7, p. 134].

After finding such contradictory sequences, IKKBZ combines the two nodes into a new node by appending their contents. The idea behind this combination is that we would achieve a lower cost by swapping the two nodes because of the ASI property. However, we are not allowed to do so since we want to follow the structure of the precedence graph. Hence, it makes sense to have the two nodes be adjacent since putting a node between these two would allow to exchange it without increasing the cost. In our example, we can see that the only node that could be between R_2 and R_4 is R_1 if we follow the structure of the precedence graph. However, since the rank of R_1 is equal to the rank of R_2, swapping the two nodes does not increase the cost according to the ASI property. Therefore, we combine the two nodes which results in the graph shown in Figure 10a. This so-called normalization is repeated until there are no contradictory sequences left. In our example, we can see that there are no contradictory sequences in the left subtree since it consists of a single node. Furthermore, the right subtree does not have any contradictions either since the rank of R_2R_4 is lower than the rank of R_5. Hence, the normalization is complete for this wedge [5, p. 497][7, p. 134].



(a) Result of normalizing R_2 and R_4

(b) Result of merging the children of R_3

Figure 10: Example execution of IKKBZ-Sub on the precedence graph rooted at R_3

Once there are no more contradictions left, the algorithm merges the subtrees of the wedge in increasing order according to our rank function. In Figure 10b we can see the result of merging the two subtrees from our previous graph. IKKBZ repeats this process of finding a wedge, normalizing, and merging until only a single chain is left. In our example, this is the case after this first iteration. Therefore, the execution of this process which we call IKKBZ-Sub terminates by unfolding (denormalizing) the nodes. Hence, IKKBZ-Sub produces the sequence $R_3R_2R_4R_1R_5$ if we start its execution with the precedence graph rooted at R_3. Since we do not know which relation to pick as the root, IKKBZ simply tries all relations and chooses the solution with minimal cost. However, we omit the other four IKKBZ-Sub executions and the final result for our example since they are quite similar [5, p. 497][7, p. 134][8, p. 54].

7.1.1 IKKBZ-Sub

We start by defining the IKKBZ-Sub algorithm based on *list_dtree*s and later using the transformation functions from the previous chapters to formulate the complete algorithm. Since we need a rank function to execute this algorithm, we define a *ranked_dtree* locale which fixes such a rank function. Additionally, it fixes a comparator for pairs consisting of a list of *'a* relations and an *'b* edge. Furthermore, this comparator must only return that two pairs are equal if either the intersection of the sets of lists is nonempty or the edges are equal. We need this comparator to define the merging function as we will explain in the next subsection.

```
locale ranked_dtree = list_dtree t for t :: "('a list,'b) dtree" +
  fixes rank :: "'a list ⇒ real"
  fixes cmp :: "('a list × 'b) comparator"
  assumes cmp_antisym:
    "⟦v1 ≠ []; v2 ≠ []; compare cmp (v1,e1) (v2,e2) = Equiv⟧
      ⟹ set v1 ∩ set v2 ≠ {} ∨ e1 = e2"
```

While it would be possible to compare two lists by assuming either *'a::linorder* or *'b::linorder*, fixing a comparator is more general. Comparators assign each pair of elements a value of type *comp*. This datatype consists of the three values *Less*, *Equiv*, and *Greater*. Furthermore, a comparator needs to be reflexive, transitive, and the *Greater* and *Less* results must be symmetric to each other. This means that *cmp a b* must return *Less* if and only if *cmp b a* returns *Greater*. Therefore, a possible definition of *cmp* based on *'a::linorder* could look like this:

```
lift_definition cmp :: "('a list × 'b) comparator" is
  "(λx y. if hd (fst x) < hd (fst y) then Less
    else if hd (fst x) > hd (fst y) then Greater else Equiv)"
```

Merging

First, we define an auxiliary merging definition that assumes that all children of the input *dtree* are normalized chains. It turns each chain into a list and utilizes the *Sorting_Algorithms.merge* function from the mergesort algorithm to merge these lists. This function merges two lists in ascending order according to a comparator. However, it does not change the order of elements within the same list. Hence, merging $[1,2,7]$ and $[4,8,5]$ according to the usual ordering for natural numbers results in $[1,2,4,7,8,5]$. Therefore, it requires the sortedness of both of its inputs to produce a sorted output list. Furthermore, we use the *ffold* function to iterate over all children and combine them all into a single list. This has the advantage that we do not need to pick two specific chains to merge in each step. In the end, we transform this list back into a *dtree* which is now a new chain. To keep the definition and lemmas more compact, we introduce an abbreviation *merge_f* for the input function of *ffold*.

```
abbreviation merge_f :: "'a list ⇒ (('a list, 'b) dtree × 'b) fset
  ⇒ ('a list, 'b) dtree × 'b ⇒ ('a list × 'b) list ⇒ ('a list × 'b) list"
where
  "merge_f r xs ≡
    λ(t,e) b. if (t,e) ∈ fset xs ∧ list_dtree (Node r xs)
      ∧ (∀(v,e') ∈ set b.
        set v ∩ dlverts t = {} ∧ v ≠ [] ∧ e' ∉ darcs t ∪ {e})
  then Sorting_Algorithms.merge cmp' (dtree_to_list (Node r {|(t,e)|})) b
  else b"
```

```
definition merge :: "('a list,'b) dtree ⇒ ('a list,'b) dtree" where
  "merge t1 ≡ dtree_from_list (root t1)
    (ffold (merge_f (root t1) (sucs t1)) [] (sucs t1))"
```

The comparator *cmp′* compares elements according to their rank and uses *cmp* as a tie-breaker to ensure a unique ordering of all nodes. This is accomplished by the disjointness requirement enforced within the condition of the if-clause combined with the locale assumption regarding *cmp*.

```
lift_definition cmp' :: "('a list × 'b) comparator" is
  "(λx y. if rank (rev (fst x)) < rank (rev (fst y)) then Less
    else if rank (rev (fst x)) > rank (rev (fst y)) then Greater
    else compare cmp x y)"
```

Combined with the other conditions of the if-clause of *merge_f*, this ensures that the merging operation is commutative. This allows us to use lemmas regarding *ffold* which we need to prove certain properties of *merge*.

```
lemma merge_commute: "comp_fun_commute (merge_f r xs)"
```

By using this commutativity, we prove the correctness of this operation by induction on the set of successors. The first proofs show that the sets of *dverts*, *dlverts*, and *darcs* are preserved if all children are chains. While the equality for *dlverts* follows directly from the unchanged *dverts* set, we prove the other two preservations using auxiliary lemmas that induct on the argument of *ffold*.

```
lemma dverts_merge_eq:
  assumes "∀t ∈ fst ` fset (sucs t). max_deg t ≤ 1"
  shows "dverts (merge t) = dverts t"
```

```
lemma darcs_merge_eq:
  assumes "∀t ∈ fst ` fset (sucs t). max_deg t ≤ 1"
  shows "darcs (merge t) = darcs t"
```

Furthermore, we show that this operation preserves all the invariants. Since the added assumption of *ranked_dtree* does not change, we only need to show that the *wf_darcs* and *wf_dlverts* properties are preserved. This follows again by using the *ffold* lemmas. Hence, the *merge* operation preserves all the invariants of the relevant locales.

```
theorem merge_list_dtree: "list_dtree (merge t)"
```

```
corollary merge_ranked_dtree: "ranked_dtree (merge t) cmp"
```

Moreover, we prove that it preserves the distinctness of the nodes and that the head of the root of the input *dtree* does not change:

```
lemma distinct_merge:
  assumes "∀v ∈ dverts t. distinct v" and "v ∈ dverts (merge t)"
  shows "distinct v"
```

```
lemma merge_hd_root_eq: "hd (root (merge t1)) = hd (root t1)"
```

Finally, we define a function that recursively finds a wedge and uses the *merge* function to turn it into a chain. However, instead of finding only one wedge, we recursively descend into all successors. As with *merge*, this simplifies the definition since we do not need to pick an arbitrary element. Furthermore, doing so does not affect the result since all wedges need to have their children merged at some point.

```
fun merge1 :: "('a list,'b) dtree ⇒ ('a list,'b) dtree" where
  "merge1 (Node r xs) = (
    if fcard xs > 1 ∧ (∀t ∈ fst ` fset xs. max_deg t ≤ 1)
    then merge (Node r xs)
    else Node r ((λ(t,e). (merge1 t,e)) |`| xs))"
```

With the lemmas of the *merge* function it is quite simple to prove that the *merge1* function preserves all the same properties as *merge*.

Normalization

For the normalization, we first define the function *normalize1* which combines the root with its child if they are contradictory sequences. Otherwise, it descends recursively similar to *merge1*.

```
function normalize1 :: "('a list,'b) dtree ⇒ ('a list,'b) dtree" where
  "normalize1 (Node r {|(t1,e)|}) =
    (if rank (rev (root t1)) < rank (rev r)
    then Node (r @ root t1) (sucs t1)
    else Node r {|(normalize1 t1,e)|})"
| "∀x. xs ≠ {|x|} ⟹
    normalize1 (Node r xs) = Node r ((λ(t,e). (normalize1 t,e)) |`| xs)"
```

Then we write the *normalize* function which calls *normalize1* until it reaches a fixpoint. This means that there are no more contradictions in the *dtree* which was the goal of this function.

```
fun normalize :: "('a list,'b) dtree ⇒ ('a list,'b) dtree" where
  "normalize t1 =
    (let t2 = normalize1 t1 in if t1 = t2 then t2 else normalize t2)"
```

To prove the termination of this function, we show that the size of the result of *normalize1* is smaller than its input if there was some change:

```
lemma normalize1_size_decr:
  "normalize1 t1 ≠ t1 ⟹ size (normalize1 t1) < size t1"
```

Similar to the *merge* function, we prove that these functions preserve the invariants and sets. However, this time we can only show equivalence for the *dlverts* set since arcs are removed and vertices altered when combining nodes.

```
lemma normalize_darcs_sub: "darcs (normalize t1) ⊆ darcs t1"
```

```
lemma normalize_dlverts_eq: "dlverts (normalize t1) = dlverts t1"
```

Instead of normalizing only below wedges, we normalize the complete *dtree* since that generalizes and, therefore, simplifies some properties. One property we can prove this way is that any subtree of a normalized *dtree* has a rank less than or equal to its child if it only has a single successor:

```
lemma normalize_sorted_ranks:
  "⟦is_subtree (Node r {|(t1,e1)|}) (normalize t)⟧
    ⟹ rank (rev r) ≤ rank (rev (root t1))"
```

Denormalization

The last component that we need is the *denormalize* function. This one simply appends all the nodes to create a list that is in the correct order. Since we only care about the cases where the input *dtree* is a chain, we treat all cases where the successor set is not a singleton as if it were a leaf node.

```
function denormalize :: "('a list, 'b) dtree ⇒ 'a list" where
  "denormalize (Node r {|(t,e)|}) = r @ denormalize t"
| "∀x. xs ≠ {|x|} ⟹ denormalize (Node r xs) = r"
```

We prove its correctness by showing that it results in a list whose set is equal to the *dlverts* set of a *dtree* if it is a chain. Moreover, we prove that the well-formedness combined with only distinct lists as nodes results in a distinct list with the head of the root as the first element:

```
lemma denormalize_set_eq_dlverts:
  "max_deg t1 ≤ 1 ⟹ set (denormalize t1) = dlverts t1"
```

```
lemma denormalize_distinct:
  "⟦∀v ∈ dverts t1. distinct v; wf_dlverts t1⟧
    ⟹ distinct (denormalize t1)"
```

```
lemma denormalize_hd_root_wf:
  "wf_dlverts t ⟹ hd (denormalize t) = hd (root t)"
```

Full IKKBZ-Sub

Finally, we combine the *normalize* and *merge1* functions in the definition of *ikkbz_sub* which repeats these two steps until the *dtree* is a single chain.

```
function ikkbz_sub :: "('a list,'b) dtree ⇒ ('a list,'b) dtree" where
  "ikkbz_sub t1 =
    (if max_deg t1 ≤ 1 then t1 else ikkbz_sub (merge1 (normalize t1)))"
```

To prove its termination, we show that *merge1* decreases the number of leaves if *max_deg* is greater than one. Since normalize does not increase the number of leaves either, we can show that each iteration of *ikkbz_sub* decreases the number of leaves.

```
lemma ikkbz_num_leaves_decr:
  "max_deg t1 > 1 ⟹ num_leaves (merge1 (normalize t1)) < num_leaves t1"
```

From the previous lemmas about *merge1* and *normalize* we can conclude that all the same properties hold for *ikkbz_sub* as well. Additionally, it follows directly from the termination condition that the resulting *dtree* consists of a single chain. Combining these results with the *denormalize* lemmas leads to the correctness of the *ikkbz_sub* function:

```
corollary denormalize_ikkbz_eq_dlverts:
  "set (denormalize (ikkbz_sub t)) = dlverts t"

corollary distinct_denormalize_ikkbz_sub:
  "∀v ∈ dverts t. distinct v ⟹ distinct (denormalize (ikkbz_sub t))"

corollary denormalize_ikkbz_sub_hd_root:
  "hd (denormalize (ikkbz_sub t)) = hd (root t)"
```

So far, the execution of *ikkbz_sub* started with a list-based *dtree*. However, we want to start IKKBZ-Sub on a (finite) directed tree rooted at some relation *root*. Therefore, we define the *precedence_graph* locale which extends *finite_directed_tree*s by *rank* and *cmp* parameters which we also added in the *ranked_dtree* locale. Since we want to use these to instantiate this locale, the comparator needs to satisfy the same assumption that is required in the *ranked_dtree* locale. Additionally, we fix a cost function and assume that it satisfies the ASI property. We will use this in Section 7.2 to prove the optimality of IKKBZ.

```
locale precedence_graph = finite_directed_tree +
  fixes rank :: "'a list ⇒ real"
  fixes cost :: "'a list ⇒ real"
  fixes cmp :: "('a list × 'b) comparator"
  assumes asi_rank: "asi rank root cost"
      and cmp_antisym:
    "⟦v1 ≠ []; v2 ≠ []; compare cmp (v1,e1) (v2,e2) = Equiv⟧
      ⟹ set v1 ∩ set v2 ≠ {} ∨ e1 = e2"
```

In this context, we define a transformation to a list-based *dtree* by combining two of our transformation functions:

```
definition to_list_dtree :: "('a list, 'b) dtree" where
  "to_list_dtree = finite_directed_tree.to_dtree to_list_tree [root]"
```

Moreover, we use our tranformation lemmas to show that this produces a *ranked_dtree* and define a new *ikkbz_sub* function which executes the *ranked_dtree*'s *ikkbz_sub* function on this result.

```
interpretation t: ranked_dtree to_list_dtree
```

```
definition ikkbz_sub :: "'a list" where
  "ikkbz_sub = denormalize (t.ikkbz_sub to_list_dtree)"
```

Finally, we combine our results for the *ranked_dtree*'s *ikkbz_sub* function with the lemmas of our transformation functions to prove that the new *ikkbz_sub* produces the desired results. This means that it contains every vertex of T exactly once and the root is the first node in the sequence.

7.1.2 Complete IKKBZ

Since we want to run the complete IKKBZ algorithm on an acyclic query graph, we combine the *query_graph* and *undir_tree_todir* locales in the *tree_query_graph* locale. Given that the *undir_tree_todir* locale includes the transformation from graph to directed tree, we can use this to generate a precedence graph from the query graph.

```
locale tree_query_graph = undir_tree_todir G + query_graph G for G
```

Furthermore, we extend this locale by fixing a cost function *cost* which we want to optimize and the comparator that we need for our *precedence_graph* locale. We split this definition into separate locales since we want to start execution based on *cmp_tree_query_graph* and derive the additional assumptions from a proper cost function.

```
locale cmp_tree_query_graph = tree_query_graph +
  fixes cmp :: "('a list × 'b) comparator"
  assumes cmp_antisym:
        "⟦v1 ≠ []; v2 ≠ []; compare cmp (v1,e1) (v2,e2) = Equiv⟧
          ⟹ set v1 ∩ set v2 ≠ {} ∨ e1 = e2"
```

```
locale ikkbz_query_graph = cmp_tree_query_graph +
  fixes cost :: "'a joinTree ⇒ real"
  fixes cost_r :: "'a ⇒ ('a list ⇒ real)"
  fixes rank_r :: "'a ⇒ ('a list ⇒ real)"
  assumes asi_rank: "r ∈ verts G ⟹ asi (rank_r r) r (cost_r r)"
      and cost_correct:
        "⟦valid_tree t; no_cross_products t; left_deep t⟧
          ⟹ cost_r (first_node t) (revorder t) = cost t"
```

However, since the ASI property may not be satisfied directly by the original cost function, we fix two additional functions *cost_r* and *rank_r*. These return cost and rank functions that satisfy the ASI property for sequences starting with some relation *r*. To keep the assumptions as general as possible, we only require equivalence of the *cost_r* and *cost* functions for valid, left-deep join trees without cross products.

In this context, we define an abbreviation for generating a precedence graph rooted at some vertex *r* and executing *ikkbz_sub* on it. Furthermore, we abbreviate the call of the *cost* function such that we can use it on lists.

```
abbreviation ikkbz_sub :: "'a ⇒ 'a list" where
  "ikkbz_sub r ≡
    precedence_graph.ikkbz_sub (dir_tree_r r) r (rank_r r) cmp"

abbreviation cost_l :: "'a list ⇒ real" where
  "cost_l xs ≡ cost (create_ldeep xs)"
```

Based on these abbreviations, we define the complete IKKBZ algorithm which picks the *ikkbz_sub* result that has the lowest cost:

```
definition ikkbz :: "'a list" where
  "ikkbz ≡ arg_min_on cost_l {ikkbz_sub r|r. r ∈ verts G}"
```

Finally, we use the proofs for *ikkbz_sub* to show that the left-deep tree created from the *ikkbz* result is a valid join tree for this query graph:

```
theorem ikkbz_valid_tree: "valid_tree (create_ldeep ikkbz)"
```

7.2 Optimality of IKKBZ

IKKBZ_Optimality.thy

In this section, we prove that the result of the IKKBZ algorithm produces an optimal solution within all valid, left-deep join trees without cross products. First, we add definitions to decide if a sequence conforms to a precedence graph. Then, we use these to prove that combining two nodes as per the normalization phase preserves the existence of an optimal solution. However, this requires some additional assumptions which we will then show to be invariants of the *ikkbz_sub* function. Finally, we combine these results to show that there exists an optimal solution for every intermediate set of *dverts*. From this, we conclude that the result of the *ikkbz* function is an optimal solution.

7.2.1 Conforming Sequences

A sequence $S = v_1, \ldots, v_k$ of nodes conforms to a precedence graph if the following conditions are satisfied [8, p. 50]:

1. for all i with $2 \leq i \leq k$ there exists a j with $1 \leq j < i$ such that $(v_j, v_i) \in E$
2. there is no edge $(v_i, v_j) \in E$ for $i > j$.

In other words, the first condition requires that every element except the first needs to have an incoming arc from a node before itself. Hence, we add this definition in the context of *directed_trees*. Note that since list indices in Isabelle/HOL start with zero, all numbers are shifted by one.

```
definition forward :: "'a list ⇒ bool" where
  "forward xs = (∀i ∈ {1..(length xs - 1)}. ∃j < i. xs!j →T xs!i)"
```

Furthermore, since list indices are sometimes a bit cumbersome, we include an equivalent recursive definition based on the reverse list.

```
fun forward_arcs :: "'a list ⇒ bool" where
  "forward_arcs [] = True"
| "forward_arcs [x] = True"
| "forward_arcs (x#xs) = ((∃y ∈ set xs. y →T x) ∧ forward_arcs xs)"

lemma forward_arcs_alt: "forward xs ⟷ forward_arcs (rev xs)"
```

Similarly, we define the second condition which states that there should be no arcs in the reverse direction. However, this time the recursive definition does not need to work on a reversed list.

```
definition no_back :: "'a list ⇒ bool" where
  "no_back xs = (∄i j. i < j ∧ j < length xs ∧ xs!j →T xs!i)"

fun no_back_arcs :: "'a list ⇒ bool" where
"no_back_arcs [] = True"
| "no_back_arcs (x#xs) = ((∄y. y ∈ set xs ∧ y →T x) ∧ no_back_arcs xs)"

lemma no_back_arcs_alt: "no_back xs ⟷ no_back_arcs xs"
```

We can now use these definitions to define conforming sequences:

```
definition seq_conform :: "'a list ⇒ bool" where
  "seq_conform xs ≡ forward_arcs (rev xs) ∧ no_back_arcs xs"
```

Some interesting observations include that this implies that all elements should be reachable from the first element in a sequence and that *forward* is sufficient for distinct sequences since it implies *no_back*:

```
lemma hd_reach_all_forward':
  "⟦length xs > 1; forward xs; x ∈ set xs⟧ ⟹ hd xs →*T x"

lemma no_back_if_distinct_forward:
  "⟦forward xs; distinct xs⟧ ⟹ no_back xs"
```

Moreover, two appended lists satisfy the *forward* property if both are already forward and there is an edge from the first sequence to the head of the second one.

```
lemma forward_app:
  "⟦forward s1; forward s2; ∃x ∈ set s1. x →T hd s2⟧ ⟹ forward (s1@s2)"
```

Furthermore, an arc from outside a *forward* list into it must lead to the head of said list. This is the case because every element in a *forward* sequence must have an incoming arc unless it is the head. However, since we are in the context of a directed tree, every vertex can have at most one incoming arc. Therefore, the arc must lead to the head of the list.

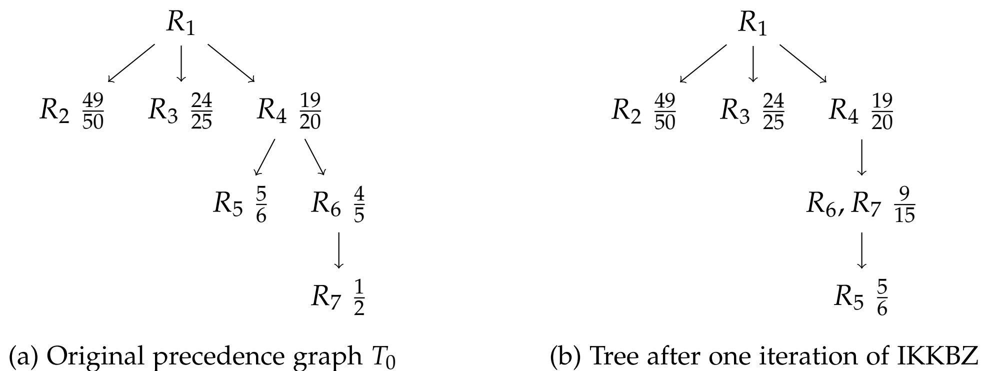
```
lemma forward_arc_to_head':
  assumes "forward ys" and "x ∉ set ys" and "y ∈ set ys" and "x →T y"
  shows "y = hd ys"

corollary forward_arc_to_head:
  "⟦forward ys; set xs ∩ set ys = {}; x ∈ set xs; y ∈ set ys; x →T y⟧
    ⟹ y = hd ys"
```

Based on these conforming sequences, we define the notion of *before* which is true if one subsequence must occur before another one in any conforming sequence that contains these two subsequences. The first three requirements are that both sequences already conform to the precedence graph and that they are disjoint. Furthermore, there must be an arc from an element of the first sequence to some node of the second sequence [8, p. 50][10, p. 41].

```
definition before :: "'a list ⇒ 'a list ⇒ bool" where
  "before s1 s2 ≡ seq_conform s1 ∧ seq_conform s2 ∧ set s1 ∩ set s2 = {}
    ∧ (∃x ∈ set s1. ∃y ∈ set s2. x →T y)"
```

Even though the definitions according to the "Query Optimization" lecture slides [10, p. 41] and the "Building Query Compilers" paper [8, p. 50] have another condition which states that all outgoing edges of the first sequence should lead to the second sequence, I omitted this condition because it is not necessary for my proofs and seems to cause some problems as explained on this example:



(a) Original precedence graph T_0 (b) Tree after one iteration of IKKBZ

Figure 11: Example of directed trees where nodes are annotated with their rank based on an example IKKBZ execution from the "Query Optimization" lecture [10, p. 56f]

In Figure 11, we can see a precedence graph and the result of executing one iteration of the IKKBZ algorithm on it. The definitions of a conforming sequence must refer to the original precedence graph T_0 shown in Figure 11a. Otherwise, $[R_6, R_7]$ would not be a conforming sequence since there no longer is an arc between R_6 and R_7 in Figure 11b. Of course, it would be a conforming sequence by considering the compound $[R_6, R_7]$ as a single node. However, this would make the whole definition superfluous since we would only consider sequences of singletons. Hence, the definitions make more sense when referring to T_0.

In the next iteration of IKKBZ, R_4 and $[R_6, R_7]$ would be combined since $rank(R_4) > rank(R_6, R_7)$. Therefore, to be able to use the definition of *before*, we would need that there is no arc from R_4 to anything except $[R_6, R_7]$. However, in T_0 there is an arc from R_4 to R_5 so this does not hold when referring to T_0. Hence, the definition of *before* is more useful without the fourth condition since it allows us to apply it directly to the original precedence graph.

A similar problem occurs with the definition of a module which is used in the optimality proofs from the “Query Optimization” lecture slides [10, p. 48f] and the “Building Query Compilers” paper [8, p. 53f]. While it would probably be possible to use these definitions and follow the proofs more closely, it seems to be much simpler to prove the optimality by referring directly to the original precedence graph. Therefore, I decided to slightly deviate from these proofs and show the optimality more directly.

But first, we show that our *before* definition works as intended by proving some properties. The first of these is that if one sequence is *before* another one, the combination of these two lists results in a conforming sequence. The proof is based on combining the *forward_app* lemma with the result that the arc connecting these two sequences must lead to the head of the second one as a consequence of *forward_arc_to_head*.

```
lemma seq_conform_if_before: "before xs ys ⟹ seq_conform (xs@ys)"
```

The second property is that *before* implies that in any valid sequence which contains two subsequences U and V for which *before* U V holds, U must appear before V. With a valid sequence, we refer to a distinct list with *forward* arcs where the first element is the *root* of the directed tree. To capture the subsequence relationship, we use the predefined *sublist* definition. It returns true if the first argument appears within the second one.

```
definition sublist :: "'a list ⇒ 'a list ⇒ bool" where
  "sublist xs ys = (∃ps ss. ys = ps @ xs @ ss)"

lemma sublist_before_if_before:
  assumes "hd xs = root" and "forward xs" and "distinct xs"
      and "sublist U xs" and "sublist V xs" and "before U V"
    shows "∃as bs cs. as @ U @ bs @ V @ cs = xs"
```

The proof uses that U and V can not overlap due to their disjointness. Therefore, if U is not before V, it would have to be behind V. However, since V has an incoming arc, it can not be equal to the *root*. Moreover, since the list is distinct and every node has at

most one incoming arc, the only arc to the head of V starts from a node which is in U. Hence, the sequence can not be *forward* which is a contradiction.

7.2.2 Combining Preserves an Optimal Solution

This subsection explains some details of the proof that the normalization preserves the existence of an optimal solution if certain preconditions are satisfied. First, we add two definitions that define the solution space we are interested in. The first definition is *unique_set_r* which returns true if its input satisfies the usual correctness properties. This means that it should be a list that contains all elements of a set exactly once and the first element should be the root. While it would be possible to express the last property using the *hd* function, it turned out that using *take 1* is slightly more convenient. In *fwd_sub* we add the additional requirements that the list should satisfy the *forward* property and each element of the set of sequences Y should be contained as a sublist.

```
definition unique_set_r :: "'a ⇒ 'a list set ⇒ 'a list ⇒ bool" where
  "unique_set_r r Y ys ⟷
    set ys = ⋃(set ` Y) ∧ distinct ys ∧ take 1 ys = [r]"

definition fwd_sub :: "'a ⇒ 'a list set ⇒ 'a list ⇒ bool" where
  "fwd_sub r Y ys ⟷
    unique_set_r r Y ys ∧ forward ys ∧ (∀xs ∈ Y. sublist xs ys)"
```

The idea behind these definitions is that *unique_set_r* expresses the basic conditions required for a valid join tree. These are that a join tree must have distinct relations and contain exactly the relations contained in the query graph. Furthermore, we restrict it to those solutions starting with *r* since IKKBZ-Sub fixes a relation as a root. The *forward* restraint in *fwd_sub* represents exactly the left-deep trees without cross products as we will show near the end of Section 7.2.4. The last condition is necessary to show that combinations do not interfere with each other. Moreover, it can be lifted in the end since we start with lists of singletons. Hence, the condition is automatically satisfied because of the set equality.

Now we use these definitions to show the existence of a sequence that satisfies *fwd_sub root* Y and contains the combination $U@V$ of the contradictory sequences U and V. Furthermore, it needs to have a cost that is less than or equal to the cost of all other lists that satisfy *fwd_sub root* Y.

Therefore, the assumptions that we need for this lemma are that U has to be before V but does not have a lower rank. Furthermore, we assume that an element of Y must not have a lower rank than V if it has to be after U but can be before V. While it may not be obvious that this is the case, we will prove that this is an invariant of IKKBZ in the next section. Moreover, we assume that there exists some sequence which satisfies *fwd_sub root* Y and that the *rank* and *cost* function satisfy the ASI property.

We also need some additional conditions for Y to ensure that the goal is provable. The first two of these conditions are that all elements should be pairwise disjoint and

satisfy the *forward* property. This allows us to rearrange elements of Y without violating the conditions of *fwd_sub*. Furthermore, Y should be finite and must not contain the empty list. Of course, it is also necessary that U and V are contained in Y. Otherwise, we may not be able to prove the requirement that $U@V$ should be contained. This Y set will later be instantiated with the *dverts* of intermediate trees. The following is the corresponding lemma with all assumptions:

```
lemma no_gap_if_contr_seq_fwd:
  assumes "asi rank root cost"
      and "∀xs ∈ Y. ∀ys ∈ Y. xs = ys ∨ set xs ∩ set ys = {}"
      and "∀xs ∈ Y. forward xs" and "[] ∉ Y" and "finite Y"
      and "U ∈ Y" and "V ∈ Y"
      and "before U V" and "rank (rev V) ≤ rank (rev U)"
      and "⋀xs. ⟦xs ∈ Y; ∃y ∈ set xs. ¬(∃x' ∈ set V. x' →⁺T y)
                          ∧ (∃x ∈ set U. x →⁺T y); xs ≠ U⟧
         ⟹ rank (rev V) ≤ rank (rev xs)"
      and "∃x. fwd_sub root Y x"
    shows "∃zs. fwd_sub root Y zs ∧ sublist (U@V) zs
         ∧ (∀as. fwd_sub root Y as ⟶ cost (rev zs) ≤ cost (rev as))"
```

Its proof uses the *sublist_before_if_before* lemma combined with some of the assumptions to obtain an arbitrary sequence *as@U@bs@V@cs* which has minimal cost and satisfies *fwd_sub root Y*. Then, we show that the rank of *bs* can not be smaller than that of V. This allows us to swap *bs* and V without increasing the cost due to the ASI property. Moreover, this swapping preserves all properties of *fwd_sub root Y* because it is still *forward* since there is an arc from U to the head of V. All the other conditions are also satisfied since they were true before and we can not have any overlapping sequences due to the second assumption.

To prove that *bs* does not have a lower rank than V, we first prove that we can split *bs* into subsequences such that all of them have a rank greater than or equal to that of V. We start this by splitting *as*, *bs*, and *cs* into all elements of Y. We denote these split lists by appending a ′ to their name (e.g. *bs*′). This splitting is possible since all elements are sublists and pairwise disjoint.

Then we use a function *make_list_P* which generates the desired result by combining the sublists of *bs* until the condition is satisfied. It utilizes the *List.extract* function to find the first element that satisfies a predicate P and the lists before and after that element. This result triple is wrapped in an *option* since it is possible that no element satisfies P in which case it returns *None* instead. Then *make_list_P* uses an auxiliary function to combine an extracted element with its predecessor until it no longer satisfies P or does not have any predecessor. Once this terminates, *make_list_P* continues on the untouched rest of the list until it no longer finds a list that satisfies P.

Figure 12 shows an example of executing this function on a list that consists of seven lists initially. The lists [3], [5], and [1] satisfy that the sum of elements is less than 10 or exactly 17. However, this predicate does not have any meaning and we can just treat it as

some arbitrary P. The first element that is found is $[3]$ with its list of predecessors $[[10]]$ and its successors $[[2,9],[5],[1],[11],[7,4]]$. Calling the auxiliary function combines $[10]$ and $[3]$ to $[10,3]$ which does not satisfy P. Hence, *make_list_P* continues by finding the next element in the still untouched list $[[2,9],[5],[1],[11],[7,4]]$. This results in the next triple consisting of $[[2,9]]$, $[5]$, and $[[1],[11],[7,4]]$. Since $[2,9,5]$ does not satisfy P, the auxiliary function terminates after one iteration again. Therefore, *make_list_P* continues and finds $[]$, $[1]$, and $[[11],[7,4]]$ as the last triple. While $[1]$ does not have a predecessor in this triple, we still have the results of the previous iterations stored in an accumulator. Hence, it combines $[2,9,5]$ and $[1]$ to $[2,9,5,1]$. However, since this still satisfies P, it combines this list with its next predecessor which results in $[10,3,2,9,5,1]$. Finally, no list satisfies P and *make_list_P* terminates with the result $[[10,3,2,9,5,1],[11],[7,4]]$.

[10] [3] [2,9] [5] [1] [11] [7,4]
[10,3] [2,9] [5] [1] [11] [7,4]
[10,3] [2,9,5] [1] [11] [7,4]
[10,3] [2,9,5,1] [11] [7,4]
[10,3,2,9,5,1] [11] [7,4]

Figure 12: Example of combining lists with predecessors until their sums are no longer less than 10 or equal to 17. Lists that satisfy this property are highlighted in blue.

The following are the concrete definitions of *make_list_P* and its auxiliary function:

```
fun combine_lists_P
  :: "('a list ⇒ bool) ⇒ 'a list ⇒ 'a list list ⇒ 'a list list"
where
  "combine_lists_P _ y [] = [y]"
| "combine_lists_P P y (x#xs) =
    (if P (x@y) then combine_lists_P P (x@y) xs else (x@y)#xs)"

fun make_list_P
  :: "('a list ⇒ bool) ⇒ 'a list list ⇒ 'a list list ⇒ 'a list list"
where
  "make_list_P P acc xs = (case List.extract P xs of
      None ⇒ rev acc @ xs
    | Some (as,y,bs) ⇒
        make_list_P P (combine_lists_P P y (rev as @ acc)) bs)"
```

By instantiating it with $P = (\lambda x.\ rank(rev\ x) < rank(rev\ V))$ we get a list of sequences such that their concatenation is still equal to bs and none has a rank lower than V. We show that this is the case in the omitted lemma *make_list_notP*.

In its proof, we use that the y argument of the *combine_lists_P* function is not reachable from U. For the initial argument, this follows directly from the assumption that a sequences in Y can not have a rank lower than the rank of V if it can be before V and

must be behind U. Moreover, its predecessor x must have an arc to y as we illustrate in Figure 13. Otherwise, it would be possible to swap x and y while preserving *fwd_sub*. However, since y is the first element that has a rank lower than the rank of V, x's rank must be larger or equal to that. By transitivity, this would mean that y has a lower rank than x. Therefore, swapping these two subsequences would result in a sequence with a lower cost which contradicts the optimality of *(as@U@bs@V@cs)*. Hence, there must be an arc from x to y. By the transitivity of reachability and the *hd_reach_all_forward'* lemma, x and, therefore, the combined sequence *x@y* are not reachable from U either.

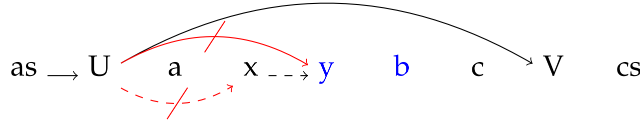


Figure 13: Example where *bs'* is [a,x,y,b,c]; *y* and *b* have lower ranks than *V*;
solid lines are known initially; dashed lines are infered by the above reasoning;
red, striked out lines mean that there is definitely no edge

By the same reasoning, there must always be a sequence between U and y because it would otherwise be possible to swap them for a lower cost while preserving *fwd_sub*. Since this would contradict the optimality of *(as@U@bs@V@cs)*, the *combine_lists_P* function's last combination must result in a sequence with a rank greater than or equal to V. Therefore, all invariants that we need for the accumulator *acc* are preserved and the lemma follows by induction. We call this function with an empty accumulator and the *bs'* list as its last argument which results in a list with the desired properties.

Hence, we can conclude that *bs* must have a rank greater than or equal to the rank of V since we can inductively swap V and the last element of the *make_list_notP* result without increasing the cost by using the ASI property.

While we can use this approach to prove the *no_gap_if_contr_seq_fwd* lemma, it is not quite sufficient to be used on our *normalize* function. This is because the lemma only works for a single combination while the *normalize* function may act on multiple subtrees at the same time. Therefore, we show another lemma that works on a set X of such combined nodes:

```
lemma combine_union_sets_optimal_cost:
  assumes "asi rank root cost"
      and "∀xs ∈ Y. ∀ys ∈ Y. xs = ys ∨ set xs ∩ set ys = {}"
      and "∀xs ∈ Y. forward xs" and "[] ∉ Y" and "finite Y"
      and "∃x. fwd_sub root Y x"
      and "∀ys ∈ X. ∃U ∈ Y. ∃V ∈ Y.
            U@V = ys ∧ before U V ∧ rank (rev V) ≤ rank (rev U)
            ∧ (∀xs ∈ Y. (∃y ∈ set xs. ¬(∃x' ∈ set V. x' →⁺T y)
                        ∧ (∃x ∈ set U. x →⁺T y) ∧ xs ≠ U)
                ⟶ rank (rev V) ≤ rank (rev xs))"
```

```
  and "∀xs ∈ X. ∀ys ∈ X. xs = ys ∨ set xs ∩ set ys = {}"
  and "∀xs ∈ X. ∀ys ∈ X. xs = ys
        ∨ ¬(∃x ∈ set xs. ∃y ∈ set ys. x →⁺T y)"
  and "finite X"
shows "∃zs. fwd_sub root (X ∪ {x. x ∈ Y ∧ set x ∩ ⋃(set`X) = {}}) zs
        ∧ (∀as. fwd_sub root Y as ⟶ cost (rev zs) ≤ cost (rev as))"
```

The goal states that an optimal solution within the union of X and all elements of Y that were not combined, has the same cost as an optimal solution within Y. The assumptions are the same as for the *no_gap_if_contr_seq_fwd* lemma just applied to all elements of X. The only addition is that none of the elements in X must be reachable from another element of X to ensure the independence of the combinations. The proof follows by induction on the set of combined nodes X combined with the previous lemma.

7.2.3 Additional Invariants of IKKBZ-Sub

Since we want to use the *combine_union_sets_optimal_cost* lemma on our *normalize* function, we need to prove all the assumptions of this lemma. Therefore, we extend the *ranked_dtree* locale by annotating it with the original precedence graph and a cost function. Furthermore, we add these additional invariants where the arcs refer to the newly annotated graph.

- *asi_rank*: The fixed rank and cost functions should satisfy the ASI property
- *dom_mdeg_gt1*: Subtrees with a maximum degree greater than one should have an arc from the root to its children's heads.
- *dom_sub_contr*: Subtrees that contain contradictory sequences should also have an arc from the root to its children's heads.
- *dom_contr*: If a root and its child are contradictory sequences and this subtree is a chain, all vertices in that subtree must be reachable from some other vertex along the path from the root to this vertex.
- *dom_wedge*: This concerns subtrees with more than one child. If it has a child whose subtree is a chain, all of its vertices must also be reachable by some node along the path from the root to this vertex.
- *arc_in_dlverts*: Arcs should be closed within the *dlverts* of its subtree. This means that if the root of a subtree has an outgoing arc, the head of this arc is contained in its *dlverts*.
- *verts_conform*: All vertices should be a conforming sequence of the annotated precedence graph.
- *verts_distinct*: All vertices should be distinct.

While most of these assumptions are quite straightforward, the assumptions that start with the prefix *dom* might be less clear. We need these to ensure that contradictory sequences have an arc from the parent to its child. This ensures that a normalized pair satisfies the *before* property which we require to use the *combine_union_sets_optimal_cost* lemma. Technically, we only need the *dom_contr* condition to prove that this is the case. However, the other properties are required to prove that this invariant is preserved. The following is the definition of this locale in Isabelle/HOL:

```
locale ranked_dtree_with_orig =
    ranked_dtree t rank cmp + directed_tree T root
  for t :: "('a list, 'b) dtree"
    and rank cost cmp and T :: "('a, 'b) pre_digraph" and root +
  assumes asi_rank: "asi rank root cost"
      and dom_mdeg_gt1:
    "⟦is_subtree (Node r xs) t; t1 ∈ fst ` fset xs;
       max_deg (Node r xs) > 1⟧
      ⟹ ∃v ∈ set r. v →T hd (Dtree.root t1)"
      and dom_sub_contr:
    "⟦is_subtree (Node r xs) t; t1 ∈ fst ` fset xs;
      ∃v t2 e2. is_subtree (Node v {|(t2,e2)|}) (Node r xs)
        ∧ rank (rev (Dtree.root t2)) < rank (rev v)⟧
      ⟹ ∃v ∈ set r. v →T hd (Dtree.root t1)"
      and dom_contr:
    "⟦is_subtree (Node r {|(t1,e1)|}) t;
       rank (rev (Dtree.root t1)) < rank (rev r);
       max_deg (Node r {|(t1,e1)|}) = 1⟧
      ⟹ dom_children (Node r {|(t1,e1)|}) T"
      and dom_wedge:
    "⟦is_subtree (Node r xs) t; fcard xs > 1⟧
      ⟹ dom_children (Node r (Abs_fset (children_deg1 xs))) T"
      and arc_in_dlverts:
    "⟦is_subtree (Node r xs) t; x ∈ set r; x →T y⟧
      ⟹ y ∈ dlverts (Node r xs)"
      and verts_conform: "v ∈ dverts t ⟹ seq_conform v"
      and verts_distinct: "v ∈ dverts t ⟹ distinct v"
```

Auxiliary Definitions for Arc Invariants

The definitions of these arc invariants use some additional auxiliary definitions: The first is *dom_children* which expresses that any vertex should be reachable from some other element along the path to this vertex. It assumes that all children have a maximum degree of one or less and uses the function *path_lverts*. This function accumulates all vertices between the root of a *dtree* and a certain element x.

```
function path_lverts :: "('a list,'b) dtree ⇒ 'a ⇒ 'a set" where
  "path_lverts (Node r {|(t,e)|}) x =
    (if x ∈ set r then {} else set r ∪ path_lverts t x)"
| "∀x. xs ≠ {|x|} ⟹ path_lverts (Node r xs) x =
    (if x ∈ set r then {} else set r)"
```

```
definition dom_children
  :: "('a list,'b) dtree ⇒ ('a,'b) pre_digraph ⇒ bool"
where
  "dom_children t1 T = (∀t ∈ fst ` fset (sucs t1). ∀x ∈ dverts t.
      ∃r ∈ set (root t1) ∪ path_lverts t (hd x). r →T hd x)"
```

To prove that the arcs invariants are preserved during *merge1*, we need an equivalent definition which expresses *path_lverts* based on lists. We show their equivalence by using the *dtree_to_list* and *dtree_from_list* definitions.

```
definition path_lverts_list :: "('a list × 'b) list ⇒ 'a ⇒ 'a set"
where
  "path_lverts_list xs x =
    (⋃(t,e)∈ set (takeWhile (λ(t,e). x ∉ set t) xs). set t)"
```

The last new definition is called *children_deg1* and is just an abbreviation for the set of all children that have a maximum degree of at most one:

```
abbreviation children_deg1
  :: "(('a,'b) dtree × 'b) fset ⇒ (('a,'b) dtree × 'b) set"
where
  "children_deg1 xs ≡ {(t,e). (t,e) ∈ fset xs ∧ max_deg t ≤ 1}"
```

Preservation of Invariants

To prove that *ikkbz_sub* preserves all invariants, we start by showing that normalizing preserves the arc invariants. Since *normalize* is just the repeated application of *normalize1*, it is sufficient to show that *normalize1* is still a valid *ranked_dtree_with_orig*. From that, we can conclude that *normalize* satisfies all conditions as well. An important lemma for these proofs states that *normalize1* preserves the maximum degree or changes it from one to zero in the case that its child is a leaf. However, this only holds if at least one of the well-formedness conditions is satisfied. Otherwise, two previously different trees may be equal after applying *normalize1* which would decrease the cardinality of a successor set.

```
lemma normalize1_mdeg_eq:
  "wf_darcs t1
    ⟹ max_deg (normalize1 t1) = max_deg t1
      ∨ (max_deg (normalize1 t1) = 0 ∧ max_deg t1 = 1)"
```

From this lemma, we can conclude the preservation of the *dom_mdeg_gt1* invariant since the node's original degree must have been larger than one as well. Therefore, the element and its children are either unchanged or a parent was merged with its successor. In the first case, the lemma follows directly from *dom_mdeg_gt1*. In the other case, it must have had a single child with a maximum degree greater than one. Therefore, the property holds as well since the combination only adds elements to the set and does not change the head.

Furthermore, we prove that the second arc invariant *dom_sub_contr* is preserved as well. In the proof, we use that a new contradiction can only occur if there have already been contradictory sequences before the normalization. Hence, we can prove this in a lemma similar to the first one.

For the next properties, we need an additional lemma which states that the result of calling *path_lverts* with the head of some vertex can only increase. This follows from the definitions combined with the disjointness of the vertices. From this lemma, we conclude that *dom_children* is preserved if we normalize the single successor of a node.

```
lemma path_lverts_normalize1_sub:
  "⟦wf_dlverts t1; x ∈ dverts (normalize1 t1);
      max_deg (normalize1 t1) ≤ 1⟧
    ⟹ path_lverts t1 (hd x) ⊆ path_lverts (normalize1 t1) (hd x)"
```

```
lemma dom_children_normalize1:
  "⟦dom_children (Node r0 {|(t1,e1)|}) T; wf_dlverts t1; max_deg t1 ≤ 1⟧
    ⟹ dom_children (Node r0 {|(normalize1 t1,e1)|}) T"
```

While the preservation of the *dom_contr* invariant follows from these lemmas combined with the result that a contradiction must have been present before, the proof of *dom_wedge* splits the node into singleton successors first. This allows us to generalize the *dom_children_normalize1* lemma to multiple children and then reassemble the result.

Furthermore, we can show that the arcs are still closed within the *dlverts* since *normalize1* does not remove any relations. Moreover, it follows from the *seq_conform_if_before* lemma combined with the invariants, that all vertices in the new *dtree* still conform to the precedence graph.

Finally, we can use these results to prove that *normalize1* produces a valid *ranked_dtree_with_orig* since we have already shown the preservation of the distinctness invariant in Section 7.1.1. From this we conclude that *normalize* must preserve all of the invariants:

```
theorem ranked_dtree_orig_normalize:
  "ranked_dtree_with_orig (normalize t) rank cost cmp T root"
```

Similarly, we want to prove that merging preserves all invariants as well. However, since *merge1* depends on the *merge* function, we will first show some properties of this function. First, we prove that *path_lverts* applied to the child of a node is a subset of the *path_lverts* after executing the *merge* function.

```
lemma path_lverts_merge_sup:
  assumes "list_dtree (Node r xs)"
      and "t1 ∈ fst ` fset xs"
      and "a ∈ dlverts t1"
    shows "∃t2 e2. merge (Node r xs) = Node r {|(t2,e2)|}
            ∧ path_lverts t1 a ⊆ path_lverts t2 a"
```

To prove this lemma, we show that *Sorting_Algorithms.merge* produces a list where the *path_lverts_list* is a superset of both of its inputs' *path_lverts_list*s. From these lemmas, we can then conclude that it holds for *merge* as well by requiring the well-formedness of its input and applying an induction on the *ffold* function's recursion argument.

From these results, we can derive that *merge* preserves the *dom_children* property if all children have well-formed vertices:

```
lemma merge_dom_children:
  "⟦dom_children (Node r xs) T; ∀t1 ∈ fst ` fset xs. wf_dlverts t1⟧
    ⟹ dom_children (merge (Node r xs)) T"
```

Furthermore, we prove that a contradiction after a *merge* can occur only at the root if the input is a normalized *dtree*. This lemma is required to prove that *merge1* preserves the *dom_sub_contr* and *dom_contr* invariants.

```
lemma merge_root_if_contr:
  "⟦⋀r1 t2 e2. is_subtree (Node r1 {|(t2,e2)|}) t1
        ⟹ rank (rev r1) ≤ rank (rev (Dtree.root t2));
      is_subtree (Node v {|(t2,e2)|}) (merge t1);
      rank (rev (Dtree.root t2)) < rank (rev v)⟧
    ⟹ Node v {|(t2,e2)|} = merge t1"
```

The proof of this lemma requires an additional comparator since *normalize* does not guarantee that nodes with equal rank are properly sorted according to *cmp′*. Hence, we define *cmp″* to compare only the ranks without using the *cmp* comparator as a tie-breaker.

```
lift_definition cmp'' :: "('a list × 'b) comparator" is
  "(λx y. if rank (rev (fst x)) < rank (rev (fst y)) then Less
    else if rank (rev (fst x)) > rank (rev (fst y)) then Greater
    else Equiv)"
```

It is quite simple to prove that a normalized *dtree* that was converted into a list is sorted according to this comparator. Moreover, it follows from the definitions that merging two sorted lists according to the *cmp′* comparator produces a sorted list as well. Hence, we conclude that all strict subtrees of a merged *dtree* are sorted according to *cmp′*. Therefore, there is no contradiction in the strict subtrees which leads us to the *merge_root_if_contr* lemma.

Now that we have these lemmas, we can prove that *merge1* preserves all invariants. The proof of the *dom_mdeg_gt1* arc property is quite simple since it only concerns parts

of the *dtree* that were not changed by previous merges. Similarly, the nodes relevant to the *dom_sub_contr* invariant are mostly unchanged as well. The only possible change is that a new contradiction was introduced by converting a node with a degree greater than one, as we know from the *merge_root_if_contr* lemma. Therefore, it follows from *dom_mdeg_gt1* that this invariant is preserved as well.

Moreover, the *dom_contr* property also holds since we know from the *merge_root_if_contr* lemma that any contradiction must have been a wedge before the call of *merge1*. Hence, we can use the *dom_wedge* invariant combined with the *merge_dom_children* lemma to prove that it still holds.

Even though the proof of the *dom_wedge* invariant is a bit more complicated, it can essentially be reduced to showing that all children whose degree was reduced to one satisfy the *dom_children* property. Since a change implies that there must have been a wedge in that subtree, we can use the *dom_mdeg_gt1* and *dom_wedge* assumptions to show that all vertices are reachable from some node along the path. Children that already had a maximum degree of at most one are not affected by *merge1*. Hence, the *dom_children* property is preserved and holds for these as well.

Furthermore, we prove the last arc invariant by using that *Sorting_Algorithms.merge* does not change the order of elements within an input list. Therefore, the set of *dlverts* of the original *dtree*s is contained in the merged subtrees which leads us to the conclusion that the *arc_in_dlverts* property is preserved.

Moreover, we know that all other invariants are preserved since the *dverts* set is not changed during merging. Therefore, we can show that *merge1* produces a valid *ranked_dtree_with_orig* assuming that the input is normalized.

```
theorem merge1_ranked_dtree_orig:
  assumes "⋀r1 t2 e2. is_subtree (Node r1 {|(t2,e2)|}) t
              ⟹ rank (rev r1) ≤ rank (rev (Dtree.root t2))"
  shows "ranked_dtree_with_orig (merge1 t) rank cost cmp T root"
```

By combining this result with the *normalize_sorted_ranks* lemma and the *ranked_dtree_orig_normalize* theorem, we can show that combination of *merge1* and *normalize* preserves all invariants. Furthermore, since *ikkbz_sub* is the repeated application of this combination, we conclude that it results in a valid *ranked_dtree_with_orig* as well.

```
theorem merge1_normalize_ranked_dtree_orig:
  "ranked_dtree_with_orig (merge1 (normalize t)) rank cost cmp T root"
```

```
theorem ikkbz_sub_ranked_dtree_orig:
  "ranked_dtree_with_orig (ikkbz_sub t) rank cost cmp T root"
```

Invariants Hold Initially

Finally, we show that these invariants hold initially in the context of a *precedence_graph*. Since the initial *dtree* is created by converting the precedence graph, it follows directly

from the transformations that a child always has an arc from its parent. Therefore, we can use this lemma to prove the first four arc invariants:

```
lemma subtree_to_list_dtree_dom:
  assumes "is_subtree (Node r xs) to_list_dtree" and "t ∈ fst ` fset xs"
  shows "hd r →T hd (Dtree.root t)"
```

Furthermore, we can conclude from the transformations that an arc from some node implies that the head of that arc is a child of its tail. Moreover, the ASI property is an assumption of the *precedence_graph* locale and the last two conditions hold trivially for singleton lists. Hence, we can use these results to show that *to_list_dtree* is a valid *ranked_dtree_with_orig*.

```
theorem to_list_dtree_ranked_orig:
  "ranked_dtree_with_orig to_list_dtree rank cost cmp T root"
```

7.2.4 Optimality of IKKBZ-Sub and IKKBZ

At last, we show that these invariants are sufficient to apply the *combine_union_sets_optimal_cost* lemma from Section 7.2.2 on the *normalize1* function. First, we show that for any vertex v that is reachable from a root r but not reachable from r's child *t1*, the rank of v can not be smaller than the rank of the root of *t1*.

```
lemma subtree_rank_ge_if_mdeg_le1:
  "⟦is_subtree (Node r {|(t1,e1)|}) t;
      max_deg (Node r {|(t1,e1)|}) ≤ 1;
      v ≠ r; v ∈ dverts t;
      ∃y ∈ set v. ¬(∃x' ∈ set (Dtree.root t1). x' →+T y)
                  ∧ (∃x ∈ set r. x →+T y)⟧
    ⟹ rank (rev (Dtree.root t1)) ≤ rank (rev v)"
```

To prove this property, we make a case distinction on the maximum degree of the subtree we are interested in. If this subtree is a chain with a contradiction, we can use the *dom_sub_contr* and *dom_contr* invariants to show that every child of *t1* is reachable from the root of *t1*. Otherwise, if *t1* is a chain without a contradiction, all vertices in the subtree are ordered according to their rank and, therefore, v can not have a lower rank. On the other hand, if *t1* has a maximum degree of greater than one, we can show that any vertex in that subtree must be reachable from its root by using the *dom_mdeg_gt1* and *dom_wedge* invariants. However, we know that all vertices reachable from r must be in the subtree of one of its children because of the *arc_in_dlverts* property. Therefore, a v other than *Dtree.root t1* that satisfies both conditions does not exist.

Furthermore, we can show that any new vertex created by the *normalize1* function has to be the combination of two contradictory sequences. This follows from the definition of *normalize1* combined with the *dom_sub_contr* invariant. Afterward, we combine these results to prove that all new vertices satisfy the first requirement of the X set.

```
lemma normalize1_dverts_app_bfr_cntr_rnks:
  assumes "v ∈ dverts (normalize1 t)" and "v ∉ dverts t"
  shows "∃U ∈ dverts t. ∃V ∈ dverts t.
          U @ V = v ∧ before U V ∧ rank (rev V) < rank (rev U)
          ∧ (∀xs ∈ dverts t. (∃y ∈ set xs. ¬ (∃x' ∈ set V. x' →⁺T y)
                                  ∧ (∃x ∈ set U. x →⁺T y) ∧ xs ≠ U)
                ⟶ rank (rev V) ≤ rank (rev xs))"
```

Moreover, we prove that there is no arc between any pair of new vertices. In this proof, we use the *arc_in_dlverts* invariant combined with the fact that we only normalize multiple nodes at once if they are in different subtrees.

The only assumption that we keep for now is the existence of a sequence that satisfies all constraints of our optimality domain. All the other remaining assumptions follow directly from the invariants and basic properties of *dtree*s. Therefore, we can use these results to prove that there exists an optimal solution within the union of the new vertices created by *normalize1* and the old vertices that do not intersect with the new ones:

```
lemma normalize1_dverts_split_optimal:
  defines "X ≡ {v ∈ dverts (normalize1 t). v ∉ dverts t}"
  assumes "∃x. fwd_sub root (dverts t) x"
  shows "∃zs.
      fwd_sub root (X ∪ {x. x ∈ dverts t ∧ set x ∩ ⋃(set ` X) = {}}) zs
      ∧ (∀as. fwd_sub root (dverts t) as
          ⟶ cost (rev zs) ≤ cost (rev as))"
```

Since these two sets represent a partitioning of the *dverts* of the *normalize1* result, we can conclude that *normalize1* and, therefore, *normalize* preserve the existence of an optimal solution:

```
lemma normalize1_dverts_split2:
  fixes t1
  defines "X ≡ {v ∈ dverts (normalize1 t1). v ∉ dverts t1}"
  assumes "wf_dlverts t1"
  shows "X ∪ {x. x ∈ dverts t1 ∧ set x ∩ ⋃(set ` X) = {}}
        = dverts (normalize1 t1)"

lemma normalize_dverts_optimal:
  assumes "∃x. fwd_sub root (dverts t) x"
  shows "∃zs. fwd_sub root (dverts (normalize t)) zs
          ∧ (∀as. fwd_sub root (dverts t) as
              ⟶ cost (rev zs) ≤ cost (rev as))"
```

Finally, we prove that *ikkbz_sub* preserves the existence of an optimal solution as well. The proof simply combines the previous lemma with the result that *merge1* does not change the set of *dverts*.

```
theorem ikkbz_sub_dverts_optimal:
  assumes "∃x. fwd_sub root (dverts t) x"
  shows "∃zs. fwd_sub root (dverts (ikkbz_sub t)) zs
          ∧ (∀as. fwd_sub root (dverts t) as
              ⟶ cost (rev zs) ≤ cost (rev as))"
```

To discharge the assumptions, we show the existence of a sequence that satisfies all properties of *fwd_sub root (dverts t)*. Since we want to show that the IKKBZ-Sub result fulfills all of these properties anyway, we use it for this existence proof. In the previous sections, we have already shown that *ikkbz_sub* results in a distinct list with a set equal to *dlverts t*. Moreover, we know that neither *ikkbz_sub* nor *denormalize* change the first element. Therefore, we only need to show that the result contains all *dverts* as a sublist and satisfies the *forward* property.

The proof of the first property uses that all initial vertices are a sublist of some vertex after applying *normalize1*. From this, we can conclude that the same holds for *ikkbz_sub* by using the transitivity of *sublist* combined with the fact that *merge1* does not change the *dverts*.

```
lemma ikkbz_sub_verts_sublist:
  "v ∈ dverts t ⟹ ∃v2 ∈ dverts (ikkbz_sub t). sublist v v2"
```

For proofs concerning the *denormalize* function, we introduce an additional transformation function that does not change the result of the *denormalize* function. It reduces a chain to a single node by combining the root with its successor until it reaches the end. Since it acts similar to *normalize* except that it does not have a condition, we call it *normalize_full*. To prove its termination, we need an additional lemma which states that the size of a singleton child replaced by its own successor set has a smaller size.

```
lemma dtree_size_skip_decr:
  "size (Node r (sucs t1)) < size (Node v {|(t1,e1)|})"

function normalize_full :: "('a list,'b) dtree ⇒ ('a list,'b) dtree"
where
  "normalize_full (Node r {|(t1,e1)|}) =
    normalize_full (Node (r @ Dtree.root t1) (sucs t1))"
| "∀x. xs ≠ {|x|} ⟹ normalize_full (Node r xs) = Node r xs"
```

As mentioned before, an important property of this function is that it does not affect the *denormalize* result. Moreover, the same is true for the *normalize1* and *normalize* functions if the input satisfies at least one of the well-formedness conditions.

With this function, we can prove that the *denormalize* result contains all vertices by showing the same sublist preservation as before and combining this with the fact that only a single vertex remains if the input of *normalize_full* was a chain.

```
lemma denormalize_full_sublist_preserv:
  assumes "sublist xs v" and "v ∈ dverts t1" and "max_deg t1 ≤ 1"
  shows "sublist xs (denormalize (normalize_full t1))"
```

```
corollary denormalize_sublist_preserv:
  "⟦sublist xs v; v ∈ dverts (t1::('a list,'b) dtree); max_deg t1 ≤ 1⟧
    ⟹ sublist xs (denormalize t1)"
```

Similarly, we show that the *denormalize* result satisfies the *forward* property if the input chain fulfills *dom_children* and all vertices are well-formed and *forward* lists: We first show that *normalize_full* results in a *dtree* where all vertices are *forward*. Then we use that a chain reduces to a single node to conclude that the *denormalize* result is *forward*.

```
lemma denormalize_forward:
  "⟦max_deg t1 ≤ 1; dom_children t1 T;
      ∀v ∈ dverts t1. forward v; wf_dlverts t1⟧
    ⟹ forward (denormalize t1)"
```

Since we know that *ikkbz_sub* only changes a *dtree* if the maximum degree of the input is greater than one, we know that *merge1* must have done the last change. Moreover, since *merge1* only alters the input if there is a wedge, we can use the *dom_mdeg_gt1* and *dom_wedge* invariants to show that the result of *merge1* satisfies the *dom_children* property if it is a chain. Therefore, we conclude that *ikkbz_sub* produces a *dtree* that satisifies *dom_children* unless its input already is a chain.

```
lemma dom_mdeg_le1_merge1:
  "⟦max_deg (merge1 t) ≤ 1; merge1 t ≠ t⟧ ⟹ dom_children (merge1 t) T"
```

```
lemma dom_mdeg_le1_ikkbz_sub:
  "ikkbz_sub t ≠ t ⟹ dom_children (ikkbz_sub t) T"
```

Finally, we combine this result with the *verts_conform* invariant and the *denormalize_forward* lemma to show that the IKKBZ-Sub result must be *forward* if we assume that an input chain already satisfies the *dom_children* property:

```
theorem ikkbz_sub_forward:
  "⟦max_deg t ≤ 1 ⟹ dom_children t T⟧
    ⟹ forward (denormalize (ikkbz_sub t))"
```

At last, we can use these results to replace the existence assumption with the appropriate preconditions. Since these follow directly from already proved properties, the resulting lemma is more useful. While it would be possible to include these conditions as invariants, they hinder the use of induction since they require that the root does not change. Hence, we exclude them from the locale and have them as separate assumptions:

```
lemma ikkbz_sub_dverts_optimal':
  assumes "hd (Dtree.root t) = root"
      and "max_deg t ≤ 1 ⟹ dom_children t T"
    shows "∃zs. fwd_sub root (dverts (ikkbz_sub t)) zs
              ∧ (∀as. fwd_sub root (dverts t) as
                  ⟶ cost (rev zs) ≤ cost (rev as))"
```

To prove that *denormalize* results in a sequence with minimal cost, we use a lemma that states that combining two nodes U and V preserves the existence of an optimal solution if U must be before V and no other element has a lower rank than V. The proof of this lemma is quite similar to the proof of *combine_union_sets_optimal_cost*.

We use this lemma to prove that an optimal solution within the *dverts* of *normalize_full* exists if its input is a sorted chain that satisfies the *dom_children* property and has the *root* as its first element.

By applying the *normalize* function beforehand, we can get rid of the sortedness assumption. Since neither of these functions changes the *denormalize* result and we only have a single vertex left, we conclude that *denormalize* produces an optimal solution.

```
lemma denormalize_optimal_if_mdeg_le1:
  assumes "max_deg t ≤ 1" and "hd (Dtree.root t) = root"
      and "dom_children t T"
    shows "∀as. fwd_sub root (dverts t) as
          ⟶ cost (rev (denormalize t)) ≤ cost (rev as)"
```

Finally, we combine all the results to prove that the denormalized *ikkbz_sub* result is an optimal solution, given the two remaining assumptions:

```
theorem denormalize_ikkbz_sub_optimal:
  assumes "hd (Dtree.root t) = root"
      and "max_deg t ≤ 1 ⟹ dom_children t T"
    shows "(∀as. fwd_sub root (dverts t) as
          ⟶ cost (rev (denormalize (ikkbz_sub t))) ≤ cost (rev as))"
```

As a conclusion of IKKBZ-Sub's optimality, we carry the results over to the *precedence_graph* context which allows us to get rid of the assumptions:

```
lemma forward_ikkbz_sub: "forward ikkbz_sub"

lemma ikkbz_sub_optimal:
  "⟦set xs = verts T; distinct xs; forward xs; hd xs = root⟧
   ⟹ cost (rev ikkbz_sub) ≤ cost (rev xs)"
```

We complete the optimality proof in the *ikkbz_query_graph* context. First, we show that for sequences that start with the root r of their precedence graph, the *forward* property is equivalent to requiring that the left-deep tree does not have a cross product. While one direction follows from the definition, the other direction uses the *no_cross_awalk* lemma. Combining this with the *apath_in_dir_if_apath_G* lemma leads to the result that a valid sequence without a cross product must be *forward*.

```
lemma no_cross_ldeep_iff_forward:
  "⟦xs ≠ []; r ∈ verts G; hd xs = r; distinct xs⟧
   ⟹ no_cross_products (create_ldeep xs)
     ⟷ directed_tree.forward (dir_tree_r r) xs"
```

Hence, we can replace *forward* with *no_cross_products* in our results for IKKBZ-Sub.

```
lemma ikkbz_sub_no_cross:
  "r ∈ verts G ⟹ no_cross_products (create_ldeep (ikkbz_sub r))"

lemma ikkbz_sub_optimal_cost_r:
  "⟦set xs = verts G; distinct xs;
      no_cross_products (create_ldeep xs); hd xs = r; r ∈ verts G⟧
    ⟹ cost_r r (rev (ikkbz_sub r)) ≤ cost_r r (rev xs)"
```

Finally, we combine the basic properties of the *Max* function with the *cost_correct* assumption to apply these results to *ikkbz*. Since any list that contains exactly the vertices of *G* must have one of the vertices as its head, this results in the optimality conditions we wanted to achieve:

```
lemma ikkbz_no_cross: "no_cross_products (create_ldeep ikkbz)"

theorem ikkbz_optimal_tree:
  "⟦valid_tree t; no_cross_products t; left_deep t⟧
    ⟹ cost (create_ldeep ikkbz) ≤ cost t"
```

7.3 Application of IKKBZ

IKKBZ_Examples.thy, CostFunctions.thy, JoinTree.thy

To conclude this chapter, we show how to apply the algorithm on the C_{out}, C_{nlj}, and C_{hj} cost functions. We start with a general cost function and then show how to instantiate it such that it leads to these concrete cost functions.

7.3.1 A General Cost Function

The general cost function expresses different cost functions by using functions h_j. These functions can be different for every relation R_j and map cardinalities to a cost value. The join cost it calculates is given by this equation [8, p. 50]:

$$cost(R_i \bowtie R_j) = |R_i| \cdot h_j(|R_j|)$$

Since every relation can have a different function, we express h by an additional parameter of type *'a ⇒ real ⇒ real*. Moreover, the cases where a tree is not left-deep are left undefined since they are not considered by IKKBZ. Hence, we get the following definition:

```
fun c_IKKBZ ::
  "('a ⇒ real ⇒ real) ⇒ 'a card ⇒ 'a selectivity
    ⇒ 'a joinTree ⇒ real"
where
  "c_IKKBZ _ _ _ (Relation _) = 0"
| "c_IKKBZ h cf f (Join l (Relation rel)) =
    card cf f l * (h rel (cf rel)) + c_IKKBZ h cf f l"
| "c_IKKBZ _ _ _ (Join l r) = undefined"
```

Since the ASI property is defined for cost functions that operate on sequences of relations instead of join trees, we derive an equivalent definition based on lists. This definition uses an additional parameter to decide which relation is first as that one should have a cost of zero. Furthermore, it works on reversed lists since the first relation is present in all joins while the last relation is only relevant during the last join. Moreover, we simplify the h function by omitting the cardinality parameter. We can do this since we always call the h function of a relation with its own cardinality [5, p. 495][7, p. 132].

```
fun c_list ::
  "('a ⇒ real) ⇒ 'a card ⇒ ('a ⇒ real) ⇒ 'a ⇒ 'a list ⇒ real"
where
  "c_list _ _ _ _ [] = 0"
| "c_list _ _ h r [x] = (if x=r then 0 else h x)"
| "c_list sf cf h r (x#xs) =
    c_list sf cf h r xs + ldeep_T sf cf xs * c_list sf cf h r [x]"
```

It utilizes the definition *ldeep_T* which computes the cardinality of left-deep trees by calculating the product of cardinalities and contributing selectivities s_k of all relations that occur in a sequence S [5, p. 495][7, p. 132]:

$$T(\epsilon) = 1$$

$$T(S) = \prod_{R_k \in S} (s_k \cdot |R_k|)$$

```
definition ldeep_T :: "('a ⇒ real) ⇒ 'a card ⇒ 'a list ⇒ real" where
  "ldeep_T sf cf xs = foldl (λa b. a * cf b * sf b) 1 xs"
```

The contributing selectivity of a relation depends on the order of the relations in a sequence. It is calculated by computing the product of the selectivities of all the relations that appear before a relation. Hence, we can use the *list_sel_aux'* function which does exactly this calculation. However, since that requires an input list, we define *ldeep_s* which maps all relations that appear in a sequence to their contributing selectivity. Since the empty product is equal to one, we use this as a default value.

```
fun ldeep_s :: "'a selectivity ⇒ 'a list ⇒ 'a ⇒ real" where
  "ldeep_s f [] = (λ_. 1)"
| "ldeep_s f (x#xs) =
    (λa. if a=x then list_sel_aux' f xs a else ldeep_s f xs a
```

We can show that the *c_IKKBZ* and *c_list* functions are indeed equivalent. However, we need the usual correctness assumptions to do so. These are that the relations are distinct and the cardinalities have reasonable values. Of course, we also require that the join tree is left-deep and that the h functions must agree for all the relations that appear in the tree. Furthermore, the list argument must be the reverse traversal of the join tree as we mentioned before.

```
theorem c_IKKBZ_eq_c_list:
fixes t
defines "xs ≡ reorder t"
assumes "distinct_relations t"
    and "reasonable_cards cf f t"
    and "left_deep t"
    and "∀x ∈ relations t. h1 x (cf x) = h2 x"
shows "c_IKKBZ h1 cf f t = c_list (ldeep_s f xs) cf h2 (first_node t) xs"
```

Furthermore, we want to prove that *c_list* satisfies the ASI property. We use the fact that since we have the root as a parameter, we can split sequences at any point to gain an equivalent definition which depends only on *c_list* and *ldeep_T* of its subsequences [5, p. 495].

```
lemma c_list_app:
  "c_list f cf h r (ys@xs)
  = c_list f cf h r xs + ldeep_T f cf xs * c_list f cf h r ys"
```

We define the rank function for *c_list* dependent on the cardinality (*ldeep_T*) and the cost function itself [5, p. 495]:

$$rank(S) = \frac{T(S) - 1}{c_list(S)}$$

Now we can use the above lemma to rewrite the difference of the cost of two sequences with swapped subsequences. That difference is equal to the difference of the rank functions of the swapped subsequences multiplied by the cardinality of the unchanged part before the change and the cost of the exchanged elements [5, p. 495]:

$$c_list(AUVB) - c_list(AVUB) = T(A) \cdot c_list(U) \cdot c_list(V) \cdot [rank(U) - rank(V)]$$

From this equation we can conclude that $c_list(AUVB) \leq c_list(AVUB) \longleftrightarrow rank(U) \leq rank(V)$ holds for positive $T(A) \cdot c_list(U) \cdot c_list(V)$. Since all three of these factors are non-negative for valid inputs, we require them to be positive in the proof of this relationship between the cost and rank functions.

Since we know that *ldeep_T* is positive for reasonable inputs, we only need to require that the *h* function is positive as well. With this requirement we can conclude that the *c_list* function returns positive values for the relevant nonempty sequences.

Finally, by combining these results, we can show that the *c_list* function satisfies the ASI property if the *h* function has this property and the other two basic validity conditions are satisfied as well:

```
theorem c_list_asi:
  fixes sf cf h r xs
  defines "f ≡ ldeep_s sf xs"
  defines "rank ≡ (λl. (ldeep_T f cf l - 1) / c_list f cf h r l)"
  assumes "sel_reasonable sf" and "∀x. cf x > 0" and "∀x. h x > 0"
  shows "asi rank r (c_list f cf h r)"
```

7.3.2 Replacing List Based Input for Tree Queries

A problem with these definitions is that *ldeep_s* requires a list as input. More concretely, the *c_IKKBZ_eq_c_list* theorem even requires that this list is the same as the last argument. However, this is not a reasonable assumption, since we can not properly instantiate it for use in IKKBZ. However, we know that in a sequence that conforms to a precedence graph, every relation has a unique incoming arc. Hence, every relation can have at most one predecessor with which it has a selectivity that is not one [8, p. 51f].

Therefore, we add two definitions in the context of a directed tree: The first one is *contr_sel* which uses the *THE* operator to map all relations to the selectivity of their parent if they have one. The second definition is *tree_sel* and decides if a selectivity corresponds to the arcs. That means that non-neighboring nodes must have a selectivity of one.

```
definition contr_sel :: "'a selectivity ⇒ 'a ⇒ real" where
  "contr_sel sel y = (if ∃x. x →T y then sel (THE x. x →T y) y else 1)"

definition tree_sel :: "'a selectivity ⇒ bool" where
  "tree_sel sel = (∀x y. ¬(x →T y ∨ y →T x) ⟶ sel x y = 1)"
```

To prove that *contr_sel* is correct, we first prove that *list_sel_aux'* is one if *tree_sel* holds and the list parameter (*xs*) does not contain an arc to or from *y*. We require that *forward_arcs (y#xs)* does not hold to express that there is no arc from *xs* to *y*. While this may not be true by itself, we additionally require that there is some sequence *ys* for which *forward_arcs (y#ys@xs)* holds. On the one hand, this ensures that *xs* by itself is correct which means that the problem must be the missing arc to *y*. On the other hand, this means that it is possible to insert some elements in between such that it becomes correct. This implies that there can not be an arc from *y* to *xs* either. Therefore, it follows by induction and the definition of *tree_sel* that *list_sel_aux'* must indeed be one.

With this auxiliary lemma we can now prove that *contr_sel* is equal to *list_sel_aux'* if the concatenation of *y* and xs is distinct and satisfies *forward_arcs*.

```
lemma contr_sel_eq_list_sel_aux'_if_tree_sel:
  "⟦tree_sel sel; distinct (y#xs); forward_arcs (y#xs); xs ≠ []⟧
    ⟹ contr_sel sel y = list_sel_aux' sel xs y"
```

Finally, we can conclude that *contr_sel* is equal to *ldeep_s* called with a *forward* list that contains all vertices exactly once:

```
corollary contr_sel_eq_ldeep_s_if_tree_dst_fwd_verts':
  "⟦tree_sel sel; distinct xs; forward xs; set xs = verts T⟧
    ⟹ contr_sel sel = ldeep_s sel (rev xs)"
```

To transfer these results to the context of a *tree_query_graph*, we abbreviate the call of *contr_sel* such that we only need the root node as a parameter. As the selectivity argument, we use *match_sel* since that is the selectivity that corresponds to our query graph. Therefore, it is the selectivity that we want to reason about.

```
abbreviation sel_r :: "'a ⇒ 'a ⇒ real" where
  "sel_r r ≡ directed_tree.contr_sel (dir_tree_r r) match_sel"
```

Since we know that the query graph is an undirected tree, we know that the only edges that are not contained in a directed tree must be the reverse arcs. Therefore, *match_sel* must always satisfy the *tree_sel* property. Thus, we can use *contr_sel* instead of *ldeep_s* for all directed trees of such a query graph.

```
lemma dir_tree_sel:
  "r ∈ verts G ⟹ directed_tree.tree_sel (dir_tree_r r) match_sel"
```

7.3.3 Application of IKKBZ on Simple Cost Functions

To apply IKKBZ, we need three additional definitions. The first one wraps the cardinality function *cf* into a wrapper that maps everything that is not a vertex of our query graph to one. This is necessary since *c_list_asi* requires that all cardinalities are positive while the cardinality of a query graph is only required to be positive for vertices. However, since we are only interested in join trees that contain the vertices of our query graph, *cf′* produces the same cardinalities and costs as *cf*.

```
definition cf' :: "'a ⇒ real" where
  "cf' x = (if x ∈ verts G then cf x else 1)"
```

Furthermore, we use the definitions *c_list_r* and *rank_r* to abbreviate the calls of *c_list* and the corresponding rank function. Apart from the list, we only need an *h* function and a root node *r* as parameters since everything else is instantiated with fixed arguments.

```
definition c_list_r  :: "('a ⇒ real) ⇒ 'a ⇒ 'a list ⇒ real" where
  "c_list_r h r = c_list (sel_r r) cf' h r"
```

```
definition rank_r  :: "('a ⇒ real) ⇒ 'a ⇒ 'a list ⇒ real" where
  "rank_r h r xs = (ldeep_T (sel_r r) cf' xs - 1) / c_list_r h r xs"
```

With these definitions, our requirements for the ASI property reduce to requiring that the *r* parameter is a vertex of *G* and that the *h* function is positive. The proof of this theorem follows from *c_list_asi* by replacing *ldeep_s* and unfolding the definitions.

```
theorem c_list_asi:
  "⟦r ∈ verts G; ∀x. h x > 0⟧ ⟹ asi (rank_r h r) r (c_list_r h r)"
```

Moreover, we can show that valid, left-deep join trees without cross products satisfy the correctness requirement of the *ikkbz_query_graph* locale. This follows from the *c_IKKBZ_eq_c_list* lemma combined with the new definitions and their equivalences.

```
lemma c_IKKBZ_list_correct_if_simple_h:
  assumes "valid_tree t" and "no_cross_products t" and "left_deep t"
  shows "c_list_r (λx. h x (cf' x)) (first_node t) (revorder t)
        = c_IKKBZ h cf match_sel t"
```

Therefore, we can instantiate the *cost_r*, *rank_r* and *cost* parameters with these definitions if h is positive for all positive input cardinalities. However, we still need a comparator whose existence we assume by moving to the *cmp_tree_query_graph* context. Inside it, we define an anonymous context for simple cost functions with a valid h function. We refer to cost functions as simple if the h function does not change when the root node changes.

```
context
  fixes h :: "'a ⇒ real ⇒ real"
  assumes h_pos: "∀x. h x (cf' x) > 0"
```

Inside this context, it follows from the assumptions and previous theorems that the instantiations satisfy all requirements of an *ikkbz_query_graph*. We also add an abbreviation that allows us to call IKKBZ from outside this context. It may seem like it is longer, but we can only call *ikkbz* directly since we used an *interpretation* in this context.

```
theorem ikkbz_query_graph_if_simple_h:
  defines "cost ≡ c_IKKBZ h cf match_sel"
  defines "h' ≡ (λx. h x (cf' x))"
  shows
    "ikkbz_query_graph to_psp sel cf G cmp cost (c_list_r h') (rank_r h')"

abbreviation ikkbz_simple_h :: "'a list" where
  "ikkbz_simple_h ≡ ikkbz"
```

Finally, we can apply these results to the C_{nlj} and C_{hj} cost functions. The h function that expresses nested loop joins maps every relation to the identity function. For hash joins we have an even simpler h function which ignores all input and returns 1.2 [8, p. 50].

```
theorem c_nlj_IKKBZ:
  "left_deep t ⟹ c_nlj cf f t = c_IKKBZ (λ_. id) cf f t"

theorem c_hj_IKKBZ:
  "left_deep t ⟹ c_hj cf f t = c_IKKBZ (λ_ _. 1.2) cf f t"
```

Since both of these h functions are obviously positive, we can apply IKKBZ and use all of its properties like the optimality of its result:

```
corollary ikkbz_optimal_nlj:
  "⟦valid_tree t; no_cross_products t; left_deep t⟧
    ⟹ c_nlj cf match_sel (create_ldeep (ikkbz_simple_h (λ_. id)))
      ≤ c_nlj cf match_sel t"

corollary ikkbz_optimal_hj:
  "⟦valid_tree t; no_cross_products t; left_deep t⟧
    ⟹ c_hj cf match_sel (create_ldeep (ikkbz_simple_h (λ_ _. 1.2)))
      ≤ c_hj cf match_sel t"
```

7.3.4 Application of IKKBZ on the C_{out} Cost Function

Expressing the C_{out} function is slightly more complex since it requires the cardinality of the result of computing a join. Since the h function has one cardinality as its input and its output is then multiplied with the other cardinality, we can use the selectivity to calculate the resulting cardinality. Hence, we use *ldeep_s* inside h. This leads us to the following general equivalence of *c_IKKBZ* and C_{out}:

```
theorem c_out_IKKBZ:
  "⟦distinct_relations t; reasonable_cards cf f t; left_deep t⟧
    ⟹ c_IKKBZ (λa b. ldeep_s f (revorder t) a * b) cf f t
      = c_out cf f t"
```

While this h definition again has the problem that it requires the *revorder* of the input join tree, we already know that we can replace it with *contr_sel*. Hence, we add these definitions of the cost and rank functions to the context of a *tree_query_graph*.

```
definition c_out_list_r  :: "'a ⇒ 'a list ⇒ real" where
  "c_out_list_r r = c_list_r (λa. sel_r r a * cf' a) r"

definition c_out_rank_r  :: "'a ⇒ 'a list ⇒ real" where
  "c_out_rank_r r = rank_r (λa. sel_r r a * cf' a) r"
```

Since we defined *sel_r* such that it is always positive, we know that these two definitions must satisfy the ASI property. Furthermore, we can use our equivalences to show that *c_out_list_r* is equal to C_{out} as well. This leads us directly to the proof of the correctness requirement of the *ikkbz_query_graph* locale:

```
lemma c_out_list_correct:
  "⟦valid_tree t; no_cross_products t; left_deep t⟧
    ⟹ c_out_list_r (first_node t) (revorder t) = c_out cf match_sel t"
```

Hence, we know that all the requirements are met and we can interpret the instantiation with the corresponding arguments as an *ikkbz_query_graph*. Therefore, we can apply the IKKBZ algorithm to the C_{out} function as well and use all of its results. For example, we can conclude the optimality of the solution:

```
interpretation QG_out:
  ikkbz_query_graph to_psp sel cf G cmp
    "c_out cf match_sel" c_out_list_r c_out_rank_r

corollary ikkbz_optimal_cout:
  "⟦valid_tree t; no_cross_products t; left_deep t⟧
    ⟹ c_out cf match_sel (create_ldeep QG_out.ikkbz)
      ≤ c_out cf match_sel t"
```

8 Conclusion

We have defined a type for selectivities and some related operations and properties in Isabelle/HOL. Furthermore, we have added definitions of query graphs and join trees in Chapter 4. Together with the example cost functions and their general properties from Chapter 5, these build a basic framework for query optimization [8, 9, 10].

In the last two chapters, we extended directed trees and implemented the join ordering algorithm IKKBZ. Moreover, we have shown that the implementation behaves correctly and produces an optimal left-deep tree without cross products if the query graph is tree-shaped. Finally, we have shown how to instantiate the parameters to apply IKKBZ on the examples of the C_{nlj}, C_{hj}, and C_{out} cost functions. The proof contains a general parameterized cost function that should simplify the extension to other cost functions [5, 7, 8].

For future work, it might be interesting to implement other query optimization algorithms in Isabelle/HOL. Since the current definitions are focused on their simplicity, replacing them with more efficient implementations could be another possible extension. As mentioned before, it would also be possible to analyze different cost functions.

Bibliography

[1] C. Ballarin. *Tutorial to Locales and Locale Interpretation*. URL: `https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/locales.pdf` (visited on 02/07/2022).

[2] S. Chaudhuri. "An Overview of Query Optimization in Relational Systems." In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*. Ed. by A. O. Mendelzon and J. Paredaens. ACM Press, 1998, pp. 34–43. DOI: `10.1145/275487.275492`.

[3] L. Fegaras. "A New Heuristic for Optimizing Large Queries." In: *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*. Ed. by G. Quirchmayr, E. Schweighofer, and T. J. M. Bench-Capon. Vol. 1460. Lecture Notes in Computer Science. Springer, 1998, pp. 726–735. DOI: `10.1007/BFb0054528`.

[4] *HyPer WebInterface - Uni Schema*. URL: `https://hyper-db.de/interface.html` (visited on 02/06/2022).

[5] T. Ibaraki and T. Kameda. "On the Optimal Nesting Order for Computing N-Relational Joins." In: *ACM Trans. Database Syst.* 9.3 (1984), pp. 482–502. DOI: `10.1145/1270.1498`.

[6] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: `https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/functions.pdf` (visited on 02/08/2022).

[7] R. Krishnamurthy, H. Boral, and C. Zaniolo. "Optimization of Nonrecursive Queries." In: *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*. Ed. by W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi. Morgan Kaufmann, 1986, pp. 128–137. ISBN: 0-934613-18-4.

[8] G. Moerkotte. *Building Query Compilers*. 2020. URL: `https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf` (visited on 12/20/2021).

[9] T. Neumann and B. Radke. *Query Optimization Lecture*. URL: `https://db.in.tum.de/teaching/ws2021/queryopt/` (visited on 12/22/2021).

[10] T. Neumann and B. Radke. *Query Optimization Lecture - Chapter 3*. URL: `https://db.in.tum.de/teaching/ws2021/queryopt/slides/chapter3.pdf` (visited on 12/22/2021).

[11] T. Nipkow. *Programming and Proving in Isabelle/HOL*. 2021. URL: `https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/prog-prove.pdf` (visited on 02/07/2022).

[12] L. Noschinski. "Graph Theory." In: *Archive of Formal Proofs* (Apr. 2013). https://isa-afp.org/entries/Graph_Theory.html, Formal proof development. ISSN: 2150-914x.

[13] L. C. Paulson. *Isabelle - A Generic Theorem Prover*. Vol. 828. Lecture Notes in Computer Science. Springer, 1994. ISBN: 3-540-58244-4. DOI: 10.1007/BFb0030541.

[14] L. Stevens and M. Abdulaziz. *Fast Diameter Estimation*. URL: https://gitlab.lrz.de/ls21/lukas-stevens/fast-diameter-estimation (visited on 02/04/2022).