Formalized Unification Algorithms

Kristoffer Hvidtfeldt

Kongens Lyngby 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

The goal of the thesis is to implement a unification algorithm from the book "Term rewriting and all that" in the theorem prover Isabelle/HOL, so that proofs of termination and correctness can be formalized. In doing so the unification algorithm will be analyzed and compared to more traditional algorithms.

It will also give an opportunity to look at the challenges which arises when implementing, proving and the formalizations of algorithms in Isabelle/HOL.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with Unification and Formalization of Algorithms.

Lyngby, 01-July-2017

Kristoffer Hvidtfeldt

Acknowledgements

I would like to thank my supervisor Jørgen Villadsen, and his two assigned Ph.D. students, Anders Schlichtkrull and John Bruntse Larsen, who all helped with the project.

I would also like to thank my friends for proof reading this thesis.

Contents

Chapter 1

Introduction

Humans have for a long time been interested in describing the world formally using logic, even before the birth of modern computer science. People such as Leibniz, Frege and Boole all had an interest in creating a system that would allow them to handle logical expressions like mathematicians handle arithmetic expressions. They wanted a system for making computations on logical expressions, where a singular truth value could be arrived at.
With the birth of computer science a need for such formalization arose, because computers can only work and apply logic when given as clearly defined values and with a clearly defined work flow. This has required the development of different systems and methods to process logic into something workable by machines.
One of these important parts are unification algorithms. The first one was presented by John Alan Robinson in 1965 as part of his resolution procedure, which is an effective tool for creating proofs in both propositional and first-order logic [BA12, chapter 4]. Unification algorithms are an important part of automated reasoning, the area of computer science dedicated to further understanding of reasoning. It is used in automated processes like artificial intelligence, as a better reasoning leads to better decisions.Unification algorithms have, as other important algorithms, been formalized to ensure and prove their correctness.
However since most formalizations of unification algorithms focus on correctness mainly, there are few formalizations of efficient unification algorithms in theorem provers. This has inspired this thesis to look at other unification algorithms than

those normally formalized. One such algorithm has been implemented in the theorem prover *ACL2* [JLRR06], but the algorithm has not been implemented in Isabelle/HOL.
This thesis is centered on the formalization of unification algorithms in the theorem prover Isabelle/HOL. It will focus on a worst case quadratic unification algorithm, which will be implemented in Isabelle/HOL. The project goals can be separated into three steps:

1. Implement a DAG-based unification algorithm in Isabelle/HOL.
2. Make the DAG-based unification algorithm have a quadratic worst-case run-time.
3. Create formal proofs for Termination and Correctness for the DAG-based unification algorithm using the tools present in Isabelle/HOL

The report will first go into the theory behind this report, such as the unification theory and the original algorithm that inspire the implementation.
It will then make an analysis of the problems and challenges of implementing this algorithm in Isabelle/HOL, as well as taken a look at other adaptations of the algorithm.
The implemented algorithm will then be presented and discussed. When the reader has become familiar with the implementation will there be taken a look at the formalization of a termination proof in Isabelle/HOL. Testing of the implementation will also be discussed.
Lastly before the thesis concludes, lessons learned and further work will be discussed in detail.

Chapter 2

Theory

This chapter will be used to introduce notation and theory necessary for understanding the reasoning and implementations in this report.

First a description of the formalization process is given and it is explained how it relates to the theorem prover Isabelle/HOL.
The next section contains a detailed explanation of the terminology used in this report together with the basics of unification theory. It will also present an example of a worst case for most standard unification algorithms to motivate the possible need for effective unification algorithms.
And lastly the DAG (Directed Acyclic Graph) based unification algorithm is presented. It is this algorithm which will be implemented in Isabelle/HOL.

2.1 Formalization and Theorem Proving

Formalization is the process of proving theorems using primitive axioms and inference rules. This is done using formal systems. A formal system is a way to calculate logic and contains:

1. A finite alphabet containing all symbols that may be used in the system.

2. A definition of how symbols of the alphabet should be used to create statement, such that they are well-formed. If a statement is not well-formed, it is not a part of the formal system. This is called the grammar of the formal system.

3. A set of universally valid statements called axioms.

4. A list of inference rules used to create theorems out of axioms and other proven theorems.

It is on the foundation of the axioms, which are known to always be true, that other theorems are built. Different formal systems have been created. Some uses a few axioms and a couple of inference rules while other uses a single inference rule with a lot of axioms.

Formalization allows for a step by step approach to working with logic with definable values. This enables computers to participate in the process of discovering and validating mathematical and logical proofs. This has been used to great effect in theorem provers, also called proof assistants. They are software made to help logician and computer scientist to create formal proofs with the assistance of a computer. They use a formal system and some highly efficient search algorithms to help the proving process.
Modern theorem provers have on the basis of simple axioms reached a high level of complexity in their formal systems. No theorem prover will accept a statement unless it can be proven using its own axioms and inference rules.
Theorem proving can be separated into two major categories: Automated and Interactive. Automated theorem provers use artificial intelligence to create formal proofs without the help of a human, while an Interactive theorem prover requires a human to guide the search. The Interactive theorem prover helps structure the process of creating the proof for the human, easing the burden of proving every single step and memorizing every remaining subgoal.

Common properties to show in the formalization of algorithms are Termination and Correctness:
Proof of Termination shows that the algorithm will always terminate. This might not be obvious in highly recursive or iterative algorithms.
Proof of Correctness show that the algorithms result at termination are always inside expected parameters. This could for an example be an function multiplying natural numbers, will always result in a natural number, since their is no way the number could become negative. Proving this formally would be a proof of correctness.

Proof assistants are very likely going to be an important technique in modern mathematics. It will allow us to expand our trust in theorems proven

long ago. Many famous theorems have been proven inside interactive theorem provers [Pro] such as Isabelle/HOL [Isa] and Coq [Coq]. It might help expand our knowledge as the tools become usable by leading mathematicians and logicians. Some speculate that it will become a part of a mathematician standard working environment [Thu94] and others that formalized proofs will be needed for publication in the future [RN14].

2.2 Unification Algorithms

This report will mainly focus on what is called first-order unification. For understanding unification one first needs to understand the concept of terms and substitution.

Terms

Terms are symbolic expressions used to model logical propositions. Terms are in this notation defined recursively as:

- Variables (`Var`) are, as the name implies, unbounded values. The variables are generally represented by the lower case letters $\{x, y, z, v\}$.
- Functions (`Fun`) consist of a function symbol to represent the function and a list of terms. The length of the list is determined by the arity of the function. Terms support a function to have varying arity. The function symbols are generally the lower case letters $\{f, g, h\}$.
- Constants are bounded values that cannot be changed. They are sometimes omitted from notations, since they functionally are identical to a function with no variables (0-ary). For easier representation, the lower case letters $\{a, b, c\}$ will be used to represent constants.

Terms can easily be represented as trees, where variables and constants are leaves and functions are branches. The term $f(x, y)$ can be seen modeled as a tree in Figure 2.1.

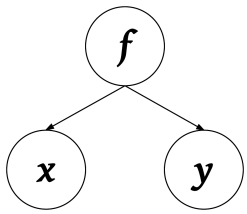

Figure 2.1: An example of the term $f(x, y)$ viewed as a tree.

Substitution

A substitution of terms is a set of variables paired with terms:

$$\{(x_1 \to t_1), ..., (x_n \to t_n)\}$$

Each pair represents a variable, x_i, that should be substituted with a term, t_i. Substitutions are represented by lower case Greek letters like $\{\lambda, \mu, \sigma, \theta\}$.

To further illustrate the notation the substitution $\sigma = \{(x \to y)\}$ can be applied on the term $t = f(x)$ such that it becomes $t\sigma = f(y)$. A substitution is also called a unifier when it used to unify two expression of terms. A larger example could be:

$$T = f(g(x), x, y, z)$$

$$\theta = \{x \to y, z \to h(y)\}$$

$$T\theta = f(g(y), y, y, h(y))$$

Substitutions can also be composed. For an example can $\sigma = \{(x \to y)\}$ be composed with the substitution $\mu = \{(y \to z)\}$ and become:

$$\sigma\mu = \{(x \to z), (y \to z)\}$$

This can also be used to update a substitution and thereby build substitution while the algorithms run.

Unification

Unification is the process of unifying equations, called terms, so that they become equivalent. This is generally speaking done by finding a substitution which, when applied on the variables of the terms will result in them becoming identical.

A unification algorithm commonly takes two terms and returns this substitution if it exist. The substitution that unifies the terms is called a unifier.

The most common method of unification is to parse two terms stored in some way as trees at the same time recursively. The two term trees are then parsed at the same time. At each point one of the terms is substituted for the other if they are different and it is possible. It is possible as long as both terms are not constants, do not have different function symbols and one term does not exist in the other. The substitution are then saved and returned as the unifier, if the two terms were unifiable.

To illustrate, here is an example of a unification of f(y,x) and f(x,z). When an

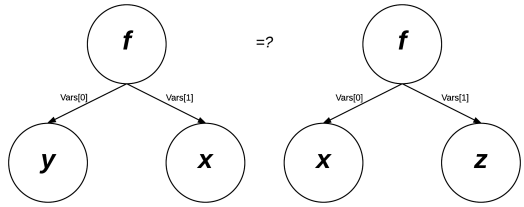

Figure 2.2: An example of f(y,x) =? f(x,z). "=?" is used to indicate unification between two terms.

algorithm starts, it will first compare the two function symbols to see if they are the same. If they are not then the two functions can not be unified, since functions of different names can not be unified. If they are identical the variables of the functions are compared in the order they appear as arguments for the function. In this example the algorithm will first try to unify x with y. Since

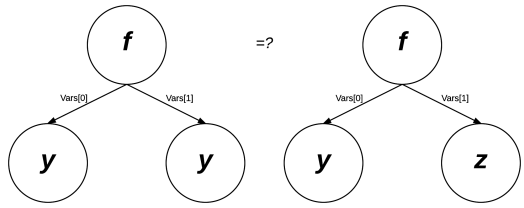

Figure 2.3: Figure 2.2 after the substitution $\{x \to y\}$ has been applied.

these terms can be unified without any problem, the substitution $[x \to y]$ will be applied on both terms, resulting in figure 2.3.
Then the algorithm will move to the second argument of both terms and try to unify y with z. Again there are no problems, so now the substitution $[x \to$

$y, y \to z]$ is applied to both terms. This results in figure 2.4. Since there are no

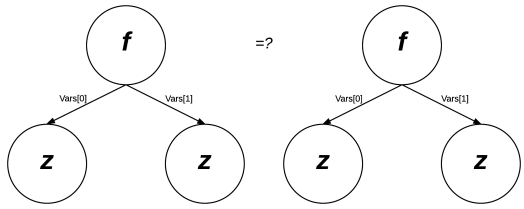

Figure 2.4: Figure 2.2 after the substitution $[x \to y, y \to z]$ has been applied. Note that the two terms are now identical. That means they have been successfully unified.

more arguments to the function and all other variables have been successfully unified, the algorithm can return the substitution as a unifier, indicating that $f(y, x)$ and $f(x, z)$ can indeed be unified by the unifier $[x \to y, y \to z]$.
Note that this is by no means the only possible unifier. $[y \to x, x \to z]$ and $[z \to y, y \to x]$ are also unifiers in this case. Even $[x \to y, y \to z, z \to v]$ is a unifier in this case, but this is not an ideal unifier.
The *most general unifier* (MGU) is a concept important for unification theory. It is the minimal set of substitution required for two terms to be unified, i.e. it is a unifier μ for a set of terms $U = t_1, ..., t_n$, such that any other unifier θ of U can be expressed as:

$$\theta = \mu\lambda \tag{2.1}$$

for some substitution λ [BA12, p. 189].
For example: $f(x, y)$ can be unified with $f(z, v)$ with the substitution $\theta = [x \to y, y \to z, z \to v]$ but it would not be an MGU. An example of an MGU would be $\mu = [x \to z, y \to v]$ since this unify the two terms and nothing more.

Unification is used in the general resolution procedure which is the foundation for logical programming, as seen in the programming language Prolog.

Occur Check

In unification theory occur check is an important concept. The occur check checks whether a variable appears inside the function, which it is being unified with. This is required to prohibit infinite terms like $X = f(X)$, which results in $f(f(f(f(...))))$. This can create cycles that could cause termination problems. In some application, like Prolog, the occur check is omitted since the run-time of the occur check can escalate as seen in the example in Section 2.2.1.

2.2.1 Example: Exponential Unification

Here is an example of a worst-case run-time example of a standard substitution unification algorithm:

The following two terms have to be unified:

$$f(x_1, x_2..., x_n)$$

$$f(g(x_0, x_0), g(x_1, x_1), ..., g(x_{n-1}, x_{n-1}))$$

Most unification algorithms will first reduce the problem by recognizing that the two function symbols f and f are identical. Then the problem becomes to transform the variable input of the two functions such that they become identical. Hence the unification algorithm will start unifying the variables of *f*. First x_1 will be substituted for $g(x_0, x_0)$. The rest of the list will then replace each instance of x_1 with $g(x_0, x_0)$. This means that x_2 will be substituted with $g(g(x_0, x_0), g(x_0, x_0))$. After only 3 terms have been unified, it has escalated in size:

$$\begin{aligned} x_1 &\rightarrow g(x_0, x_0), \\ x_2 &\rightarrow g(g(x_0, x_0), g(x_0, x_0)), \\ x_3 &\rightarrow g(g(g(x_0, x_0), g(x_0, x_0)), g(g(x_0, x_0), g(x_0, x_0))) \\ &\vdots \\ x_n &\rightarrow g(g(g(g(g(\cdots \end{aligned} \tag{2.2}$$

Especially the occur check at the last unification has a high run-time. Due to the exponential nature of the term $g(x_{n-1}, x_{n-1})$, an occur check for x_n on $g(x_{n-1}, x_{n-1})$ will have to check identical terms for x_n multiple times.

This results in both the worst-case run-time of substitution based algorithms and the data space required to store terms becoming $O(2^n)$.

2.2.2 DAG based Unification

What if there was a way to eliminate the exponential worst-case run-time from examples such as above? What if the algorithm could have a quadratic worst-case run-time? This is the main idea for the algorithm that will be inspected in this report.

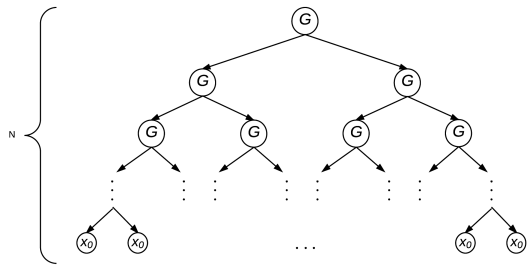

Figure 2.5: A figure showing how large x_n becomes at the end of unification

The unification algorithm, which forms the basis of this thesis, is taken from the book "Term rewriting and all that" by Franz Baader and Tobias Nipkow [BN99, chapter 4, p. 82]. It was originally implemented in Pascal.

Instead of storing terms as strings, trees or other typical classes, this algorithm will store them as directed acyclic graphs (DAGs). A DAG is graph structure where all edges are directed in such a way that no cycles exist in the graph.

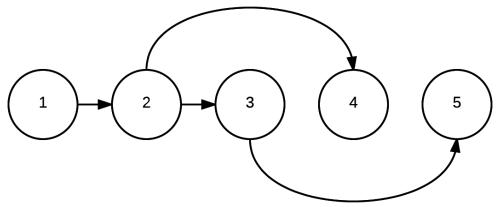

Figure 2.6: An example of a simple DAG

DAGs share some of the properties that trees also uses. They are both connected, directed, have a singular root node and contain no cycles.

The main difference between the trees and DAGS is that DAGs can have multiple parent nodes. This allows paths to converge, unlike a tree that keeps expanding. This allows every reference of a variable to be a single node in the DAG, as seen in Figure 2.8.

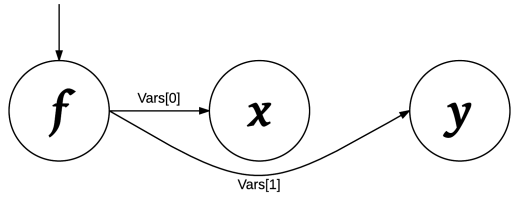

Figure 2.7: An example of the simple term f(x,y) represented as a DAG

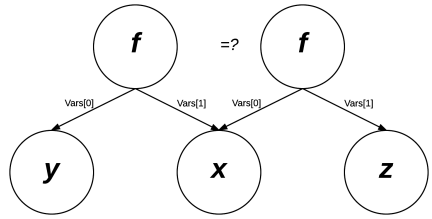

Figure 2.8: The unification between $f(y, x)$ and $f(x, z)$ represented as a DAG, before they have been unified. Note that the x from both functions is the same node.

This algorithm will update the graph, instead of substituting every single instance of the variable. The update adds extra edges that indicate which variables have been unified with which terms. Look at Figure 2.9 for an example. The graph can then be used as a look-up before comparing operations in the main unification algorithm. The main unification algorithm is other wise structure in a manner similar to other unification algorithms.

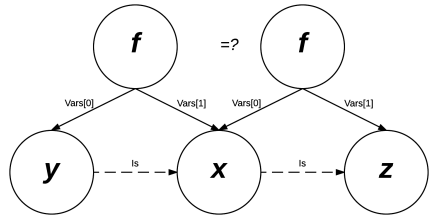

Figure 2.9: Figure 2.3 after unification. Notice the dotted lines, representing the substitution $[y \to x, x \to z]$

The graph will be able to reduce the data structure of the terms in regard to size for the worst cases like the example in the previous section. Furthermore the lookup function is also able to remove redundant substitutions.
It is worth highlighting that while this algorithm has a better worst-case run-time compared to other unification algorithms, it has higher running time in cases with a low degree of sharing [BN99]. This means that while worst-case run-time will improve, it is not necessarily the case for the average run-time.

Chapter 3

Analysis of the Problem

This chapter addresses the challenges and problems, which are expected in regard to the design and implementation of the Pascal algorithm in Isabelle/HOL.

It will take a look at the inherent difference in how the two languages grant access to pointers and how this affects the design.

3.1 From Pascal to Isabelle/HOL

As Pascal is an imperative programming language and Isabelle/HOL uses functional programming combined with logic, it is clear that an alteration of the algorithm is necessary. The biggest change is in relation to pointers. While Pascal allows for direct pointer manipulation, which the algorithm uses to create its data-structure, Isabelle/HOL does not allow for direct pointer manipulation. A method for representing pointers using functional programming concepts is required, if the algorithm is to be implemented successfully in Isabelle/HOL.

This project is not the first effort to prove a pointer-based program in Isabelle/HOL. One way to represent pointers was used in "Proving Pointer Programs in Higher-Order Logic" by Farhad Mehta and Tobias Nipkow [MN05].

They defined a pointer as a reference:

datatype 'a ref = Null | Ref 'a

This is a standard representation of a pointer. Either it points to some data of the type 'a, which in Isabelle/HOL means the type can be any type, or it is Null.
They also mention a different approach that requires creating the reference specific to type, so for example a term reference could look like:

datatype termref = Null | Var 'v | Fun 'f "'v list" 'a

This notation allows for terms to have the Null property of pointers, while having a shorter notation than the first notation. This approach however does not allow for the separation of terms and pointers which would require some changes to the algorithm. The article by Mehta and Nipkow also mentions that this approach has slightly less automatic proofs.

While pointers in imperative programming load data directly from memory using the address of the pointer, a data structure that can be passed between functions will be needed in Isabelle/HOL. In the article "Formal Correctness of a Quadratic Unification Algorithm" by José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos, José-Antonio Alonso and María-José Hidalgo [JLRR06], which also tries to adapt the quadratic algorithm to another theorem prover called *ACL2*, a structure for storing the DAG was shown. A DAG is stored as a list of terms with a hidden numerical pointer value. They use an efficient storing system, reducing each term down to the most essential information. This has inspired the following design.

It is worth highlighting that, no matter how pointers are modelled in Isabelle/HOL, the run-time of any implementation that uses pointers efficiently will not be as fast in languages without pointers. The direct access to memory can be used to great effect when working on improving run-times of algorithms in general. This is the reason why languages like *C* and *Pascal* are used when implementations need the best run-time. The run-time of the implementation in *ACL2* was tested against an implementation made in *C*, in "Formal Correctness of a Quadratic Unification Algorithm" [JLRR06]. The result was clear - the *C* implementation was twice as fast as the *ACL2*, but the *ACL2* implementation was still better than a standard unification algorithm in the worst-cases.

Chapter 4

Design & Implementation

This chapter presents the unification algorithm as it has been adapted to the theorem prover Isabelle/HOL.

The initial section will start by going over the methods used by the main algorithm. It presents essential parts of the code used and explains the motivation behind design decisions.
A short description of the method used for testing the implementation will be presented at the end.

4.1 Implementation

The different methods have been implemented three different keywords. Primrec, which means primitive recursive function, has been used for the simplest methods, like the heap handling methods. *fun* and *function* have been used for the more complex methods.
Function has been used when Isabelle/HOL was not able to detect a guaranteed termination, and it was necessary to give it a proof. *Fun* could be used as a shorthand when Isabelle/HOL could the proof without help.

4.1.1 Data Structure

One of the main challenges is to transform the naturally imperative programming structure of pointer of the algorithm to the functional nature of the Isabelle/HOL theorem prover. Farhad Mehta and Tobias Nipkow [MN05] have already made a implementation of a pointer structure in Isabelle/HOL which has been helpful in defining the data structure for the algorithm. The method revolved around having pointers defined as references:

$$datatype\ 'a\ ref = Ref\ 'a \mid Null$$

It should be noted that this definition of a pointer is functionally identical to the datatype option:

$$datatype\ 'a\ option = Some\ 'a \mid None$$

Which of these that are used is irrelevant, but ref is at least closer to the theory of the algorithm.
Note also that while it would be more appropriate to use natural numbers as the pointer values, since this is the type used in imperative languages, char list has been used instead since it provides more readability to the examples.

The only type of reference that will be used through out the algorithm is term references. Term references can take three forms:

$$Null \mid Ref\ (Var\ 'v) \mid Ref(Fun\ 'f\ ('v\ list))$$

The original algorithm defined terms as records which held the type and other values needed for variables and functions. Record is available in Isabelle/HOL, but the type interface was selected instead. This was mostly due to a lack of knowledge of how well record worked with the theorem proving aspect of Isabelle/HOL. It might be worth trying to implement using record, if one was interested in a more direct implementation of the algorithm.

The linked list property of terms is how list normally work in most functional languages so terms could easily be defined as a normal list of pointer values.

So in the the pointer was implemented as a simple undefined value ('a) and the heap was used as short hand for "$('v \times ('v,'f)\ trm\ ref)\ list$". So the heap contains a list of pointer values paired with references of terms.

datatype $('v,'f)$ *trm* =
  Var $'v$
| *Fun* $'f$ ‹$'v$ *list*›

datatype $'a$ *ref* = *Null* | *Ref* $'a$

type-synonym $('v,'b)$ *heap* = ‹($'v \times$ $('v,'b)$ *trm ref*) *list*›

Figure 4.1: The implemented Data Structures

So $f(x, y)$ would as a heap look like:

$$\begin{aligned} f(x,y) \rightarrow [&(x, Null),\\ &(y, Null),\\ &(a, Ref(Fun\ f\ [x,y]))] \end{aligned} \tag{4.1}$$

and $g(f(x, y))$ would look like:

$$\begin{aligned} g(f(x,y)) \rightarrow [&(x, Null),\\ &(y, Null),\\ &(a, Ref(Fun\ f\ [x,y])),\\ &(b, Ref(Fun\ g\ [a]))] \end{aligned} \tag{4.2}$$

It is worth noting that all methods handle a variable that is not in the heap, as if it was in the heap with the *Null* value. This is done since the meaning of these two occurrences are identical.

To access the heap a couple of helper functions has been designed:

- **Remove**: Takes an element from a heap and a heap and returns a heap. If the element is in the heap, the resulting heap will be the heap without the element.
- **Exist**: Checks whether a term is reference to in a heap.
- **Get**: Returns the reference that a given pointer value has in a heap.
- **Is_in**: Returns true if and only if a given element appears in a given heap.

```
primrec remove :: 'v × ('v,'b) trm ref => ('v,'b) heap => ('v,'b) heap

  where
  remove m [] = []
| remove m (x#xs) = (if m = x then xs else x#(remove m xs))

primrec exist :: ('v,'f) trm => ('v,'f) heap => bool
  where
  exist m [] = False
| exist m (x#xs) = (if Ref m = (snd x) then True else (exist m xs))

primrec get :: ⟨'v ⇒ ('v,'f) heap ⇒ ('v,'f) trm ref⟩
where
  ⟨get x [] = Null⟩
| ⟨get x (p#t) = (if x = fst p then snd p else get x t)⟩

primrec is-in :: ⟨'v × ('v,'f) trm ref ⇒ ('v,'f) heap ⇒ bool⟩
where
  ⟨is-in x [] = False⟩
| ⟨is-in x (p#t) = (if x =  p then True else is-in x t)⟩
```

Figure 4.2: The implemented heap operations

All of these methods have worst-case run-time $O(n)$ where n is the length of the heap. Note that these functions are not optimally implemented. With a better data-structure and some optimal helper methods, the rest of the algorithm will run faster.

4.1.2 Find

The `Find` method is used to access the value stored at a pointer value in the heap. It is important to treat the find call as one would a pointer in imperative programming. `Find` takes a pointer together with the heap as input and returns a term. That term is the last term in the path that the pointer points to in the heap.
`Find` runs recursively through the heap until it hits `Null` or `Ref(Fun ...)`. If it hits `Null`, it returns the last pointer value wrapped in the Var tag. If it hits `Ref(Fun ...)` it returns that Function. Note also that if pointer value x does not exist in the heap, will it be taken as if it was set to `Null` and return the value as a free variable, i.e Var x.

One of termination issues in the algorithm occurs in `Find`. A naive implemen-

```
function(sequential) find :: ⟨'v ⇒ ('v,'f) heap ⇒ ('v,'f) trm⟩
  where
  ⟨find x [] = Var x⟩
| ⟨find x h = (case get x h of
          Null ⇒ Var x
        | Ref (Fun f args) ⇒ Fun f args
        | Ref (Var v) ⇒ find v h)⟩
   by pat-completeness auto
termination
  sorry
```

Figure 4.3: A naive implementation of `Find`

tation of `Find` would run forever if the input heap where to contain a cycle, like $[x \rightarrow Ref(Vary), y \rightarrow Ref(Varx)]$. In this case would it call `Find x [x→ Ref(Var y),y→ Ref(Var x)]` and `Find y [x→ Ref(Var y),y→ Ref(Var x)]` recursively forever.
Some modification can be made to the algorithm, such that it is guaranteed to terminate no matter the input. One idea is to introduce a counter and a max number of recursions, so that the algorithm will terminate when the algorithm reaches a certain depth of recursion. The max value can be set based on the size of the terms given such that if the max value is it certain that a loop has occurred. While this can make sure that the algorithm terminate is it not very efficient.
Another approach is to remove pointers from the heap when used. Since `Find` operates on a copy of the heap, not the original, elements can be removed without harming the general run of the algorithm. This results in the heap being reduced in size for each iteration, thereby guaranteeing termination. In the case of `Find x [x→ Ref(Var y),y→ Ref(Var x)]` would `Find` start first get y and remove `x→ Ref(Var y)` from the heap. Then it would get x and remove `y→ Ref(Var x)` from the heap, leaving it empty. `Find` would then return x, which would be an acceptable result. This however means that while x and y are unified, when `Find` is called on them, they return different values. While this is technical wrong, it will not create any correctness issues for the main unification algorithm. Hence for this design the second option of removing pointers is chosen. Even though this modification technically introduce problems with correctness in the not well-formed cases, without them will arguing for termination in a formalized manner be difficult. The design can be seen in Figure 4.4.

```
function(sequential) find :: ⟨'v ⇒ ('v,'f) heap ⇒ ('v,'f) trm⟩
  where
  ⟨find x [] = Var x⟩
| ⟨find x h = (case get x h of
          Null ⇒ Var x
        | Ref (Fun f args) ⇒ Fun f args
        | Ref (Var v) ⇒ find v (remove (x, Ref (Var v)) h))⟩
   by pat-completeness auto
```

Figure 4.4: The implemented `Find`

A formal termination proof have been constructed for the implemented `Find`. This can be seen in Section 4.2.1.

4.1.3 Union

The `unin` function is a simple, but crucial function that unifies two terms in the graph. It takes the pointer value x that should be set and the term y that it should be set to. It searches through the heap until it finds the pointer values pair and then replaces the term. If the pointer value x is not in the heap, it is inserted with the term y.

```
primrec unin :: ⟨'v ⇒ ('v,'f) trm ⇒ ('v,'f) heap ⇒ ('v,'f) heap⟩
  where
  ⟨unin x y [] = [(x,Ref y)]⟩
| ⟨unin x y (t#ts)  = (if x=fst t then (x,Ref y)#ts else t#(unin x y ts))⟩
```

Figure 4.5: The implemented `Union`

Note the implementation of union is called `unin` because union is all ready defined as a set operation. Also, *Ref* is added as a prefix to the given term to convert it from a term to a term ref, since the heap is made of term ref.

4.1.4 Occur Check

`Occur` takes the heap, a variable pointer and a function pointer, and return a Boolean depending on the whether or not the variable appears inside the

function. A naive implementation of `Occur` would simply go through the entire function and check if the variable ever appear as an argument.

```
function(sequential) occ :: ⟨'v ⇒ 'v ⇒('v,'f) heap ⇒ bool⟩
and occs :: ⟨'v ⇒ 'v list ⇒('v,'f) heap ⇒ bool⟩
where
  ⟨occ u v h =  (case ((find u h),(find v h)) of
        (a,Var v) ⇒ False
      | (a,Fun f args) ⇒ (occs u args h))⟩

| ⟨occs u [] h = False⟩
| ⟨occs u (v#vs) h = ((u = v) ∨ (occ u v h) ∨ occs u vs h)⟩
   by pat-completeness auto
termination
  sorry
```

Figure 4.6: A naive implementation of `Occur`

This Occur check is not good enough if the algorithm is to be quadratic. Look back at the example given on Section 2.2.1. In the last case where x_n has to be unified with $g(x_{n-1}, x_{n-1})$. Since the variable x_n has to be unified with a function $g(x_{n-1}, x_{n-1})$, an occur check to see if x_n is in $g(x_{n-1}, x_{n-1})$ is required. But by this time the term $g(x_{n-1}, x_{n-1})$ has blown up to the size of 2^n. And the occur check has to visit each one of these functions to check if x_n is one of its argument. This should not be necessary since the high degree of identical functions in the term should let the algorithm know which terms have already been checked.

This can be solved by the DAG structure, since the identical terms are the exact same term. So by introducing a value to terms, like a boolean *visited* that can be set by `Occur` can the algorithm skip already checked terms and there possible arguments.

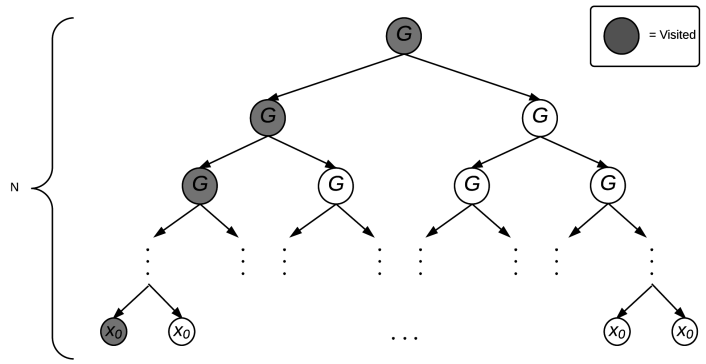

Figure 4.7: A tree showing which nodes of a tree that would need to be checked in the case if the example from Section 2.2.1

The quadratic algorithm from "Term rewriting and all that" solves this by introducing a time and stamp solution, where each term is marked with a time stamp which is updated each time it is visited by `Occur`. This is then compared and updated by a time value unique for that run of `Occur`. This, unlike the Boolean solution, does not need to be reset after each run of `Occur`.

However this design requires that Occur is only returning a Boolean. The problem of this comes from the fact that updating the heap is simple in Imperative programming languages, but here it requires that the method both takes the heap as input and as output. This would change the flow and control scheme of the main `Unify`. Due to time constraints could this version not be implemented.

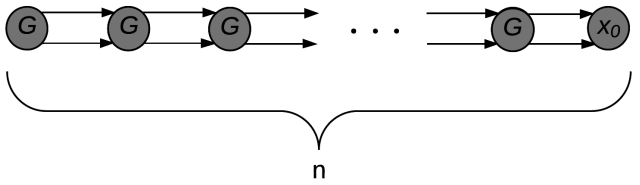

Figure 4.8: Figure 4.7, shown as the DAG it is stored as in the implementation.

So the solution used in the implementation was similar to what was used in `Find`. Since the heap worked on by `Occur` is a copy of the original, the algorithm can

freely manipulate the copy. So to mark a function as visited it is simply removed from the heap. This prevents the `Find` from finding the same function multiple times, and thus avoiding the problem discussed in Section 2.2.1.

```
function(sequential) occ :: ⟨'v ⇒ 'v ⇒('v,'f) heap ⇒(('v,'f) heap ×
bool)⟩
and occs :: ⟨'v ⇒ 'v list ⇒('v,'f) heap ⇒ (('v,'f) heap ×bool)⟩
where
  ⟨occ u v h =  (case ((find u h),(find v h)) of
        (a, Var v) ⇒ (h,False)
      | (a,Fun f args) ⇒ (occs u args (remove (v, Ref (Fun f args)) h)))⟩
| ⟨occs u [] h = (h,False)⟩
| ⟨occs u (v#vs) h = (let oc= occ u v h
                      in (if (u = v) ∨ snd(oc)
                          then (fst(oc), True)
                          else occs u vs (fst(oc))))⟩
   by pat-completeness auto
termination
  sorry

fun occurs :: ⟨'v ⇒ 'v list ⇒('v,'f) heap ⇒ bool⟩
  where
    ⟨occurs u v h = snd( occs u v h)⟩
```

Figure 4.9: The implemented `Occur`

4.1.5 Unify - The Core Algorithm

The main unification algorithm (`Unify`) takes two pointer values and a heap as input, and returns a boolean telling whether or not there exists a viable substitution and the heap modified by the unification algorithm. It is helped by another function, `Unifys`, which takes two pointer value lists and a heap as input. `Unifys` helps with the recursive part of the algorithm. They are defined together with *and* in Isabelle/HOL since they are mutually recursive.

It starts by simply checking whether the two pointer values are identical. No further actions are required if they are, and the algorithm can terminate since the two values obvious are unified. Otherwise it continue with a case expression.

The unification algorithm then applies `Find` on each of the two pointer values. The resulting terms are then used to decide between three cases: two Variables, two Functions or one of each.

```
function(sequential) unify :: ‹'v ⇒ 'v ⇒ ('v,'f) heap ⇒ (('v,'f) heap ×
bool)›
and unifys :: ‹'v list ⇒ 'v list ⇒ ('v,'f) heap ⇒ (('v,'f) heap × bool)›
where
  ‹unify v u h = (if v=u then (h,True) else (case ((find v h),(find u h))
of
     (Var v1,Var v2) ⇒
      (if v1 = v2 then (h,True) else ((unin v1 (Var v2) h),True))
   | (Var v2,Fun f args) ⇒
     (if (occurs v2 args h) then (h,False) else ((unin v2 (Var u) h),True))
   | (Fun f args,Var v2) ⇒
     (if (occurs v2 args h) then (h,False) else ((unin v2 (Var v) h),True))
   | (Fun f1 args1,Fun f2 args2) ⇒
     (if f1=f2 then unifys args1 args2 (unin v (Var u) h) else (h,False))))›
| ‹unifys [] [] h = (h,True)›
| ‹unifys [] (v2#vs2) h = (h,False)›
| ‹unifys (v1#vs1) [] h = (h,False)›
| ‹unifys (v1#vs1) (v2#vs2) h =
          (let u = unify v1 v2 h
           in (if (snd u) then (unifys vs1 vs2 (fst u)) else u))›
   by pat-completeness auto
termination
  sorry
```

Figure 4.10: The implemented `Unify`

If the `Find` function results in two variables *(Var v1, Var v2)*, then since these variables both have not been set to any term, $v1$ will, using `Union`, be set to $v2$ in the heap. If the two variables are identical i.e $v1 = v2$ then the `Union` function is unnecessary and the algorithm just returns the heap unchanged.

If the `Find` function results in one of each, the algorithm will try to substitute the variable with function. This is only possible if the variable does not occur inside the function. If this operation fails, i.e the occur check fails, then the unification will result in a failure. A failure is represented by returning Boolean value being false and the heap at the stage of the failure.

It is when the `Find` function results in two functions (`Fun` $f1$ $args1$, `Fun` $f2$ $args2$) that the recursive component of the algorithm comes in. It will start by comparing $f1$ and $f2$. If they are not identical then since functions can not be unified, the algorithm will terminate and return failure. If they are identical the algorithm will start unifying the terms in the functions variable list, i.e. $args1$ and $args2$. It will go through the list in linear order. It terminates if any of the unifications result in failure.

4.2 Formalization

A formal proof has been found for the termination of the `Find` function using the proof assisting tools of Isabelle/HOL. While fairly simple it serves as a demonstration of how to prove termination in Isabelle/HOL.

4.2.1 Formal Proof of Termination for Find

Initial attempts at proving termination for `Find` used a different version of `Find`. Instead of having it removing already visited parts of the copy of the heap, the initial version had a Boolean check at the start of the `Find` that checked the heap for cycles. It detected cycles by iteratively removing each node from the graph that did not have an edge going into it. If the graph at the end was empty then that meant there was no cycles. But if the number of nodes in the graph did not decrease after an iterative cycle, that would then mean that there was a cycle.

```
fun no-cycle1 :: ‹('v,'f) heap ⇒ ('v,'f) heap ⇒ ('v,'f) heap›
  where
  ‹no-cycle1 [] org = []›
| ‹no-cycle1 ((a,b)#t) org =
     (if exist (Var a) org
      then (a,b)#(no-cycle1 t org) else no-cycle1 t org)›

fun no-cycle :: ‹ ('v,'f) heap ⇒ bool›
  where
  ‹no-cycle [] = True›
| ‹no-cycle orgL =
     (let uptL=(no-cycle1 orgL orgL)
      in (if length uptL<length orgL then no-cycle uptL else False))›
```

Figure 4.11: An implementation of the cycle check

The problem was that the logical properties of `no_cycle` was not clear for the search algorithms of Isabelle/HOL. For this version of `Find`, a couple of lemmas would have been needed to help Isabelle/HOL see the properties of this check, such that the lemmas could be used for the termination proof. Some were implemented as seen in Appendix A, but a final termination proof for this version of `Find` was not found.

Further formalization uses the implementation shown in 4.1.2, which uses the `remove` function.

The termination proof of `Find` is not so simple that Isabelle/HOL can detect the proof without some help from a user. This is because the program can not see that with each new iteration of `Find` it nears termination. So a termination proof is needed to show that `Find` does indeed terminate.

Termination proofs in Isabelle/HOL uses a list measures that convert the input and output of functions into a sequence of natural numbers that can be sorted in a lexicographic ordering. [Kra07] A lexicographical ordering is to sort a set of sequences the following way: first sort the sequences comparing the first value of the sequences, then sort them by the next value and so forth. An example of a lexicographical ordering is an English dictionary containing all the English words. Isabelle/HOL knows a function terminates as long as each recursion of the function moves the lexicographical ordering toward 0.

To help Isabelle/HOL find the proof two lemmas have been constructed and proved:

lemma *is-in0* : *get x h* = *Ref x2* $\Longrightarrow$
is-in (*x*,*Ref x2*) *h*
apply (*induct h*,*simp*)
by (*metis get.simps*(*2*) *is-in.simps*(*2*) *prod.exhaust-sel*)

The first lemma states that if `get` when searching on x returns Ref y on the heap h, then the pair (x,Ref y) is in h.

lemma *remove1* : *is-in x h* $\Longrightarrow$(*length*(*remove x h*) < *length h*)
apply (*induct h*,*simp*)
by *simp*

The second lemma states that if `remove` is called on the pair x, which is in a heap h, then the length of the heap after the remove operation will be smaller than the length of the heap before the operation.

Both of these lemmas was easily proven using induction and the search algorithms of Isabelle/HOL, especially the tool "Sledgehammer"

Now the termination proof can be created using these two properties. When starting to make the proof in Isabelle/HOL, the program clearly defines what subgoals that need to be proven for the theorem to be true. In this case there are two subgoals:

First a measure is defined. A measure is a set of functions that can be applied to the input and output of a recursive function and result in natural numbers.

```
proof (state)
goal (2 subgoals):
 1. wf ?R
 2. ⋀x v va x2 x1.
       get x (v # va) = Ref x2 ⟹
       x2 = Var x1 ⟹ ((x1, remove (x, Ref (Var x1)) (v # va)), x, v # va) ∈ ?R
```

Figure 4.12: A screen shot of the output when selecting the beginning of the termination proof for `Find`

This will be used to create the lexicographic termination order. For this proof the only needed function is `Length` applied on the heap. The measure does not really need a proof. It is more the user guiding the program toward the proof.

For the second property, it is very important what measure was chosen. This is because the program now needs to be shown how this measure will be guarantee to decrease for every continuing iteration of the algorithm. In this case, Isabelle/HOL requires proof that if `Get x h` result in `Ref(Var x1)`, then the `Find` method will still be one step closer to termination. It is here we use the combination of the two proven lemmas, to show that if `Get x h` result in `Ref(Var x1)` then if `remove` is called on `(x,Ref(Var x1)`, the resulting heap will be smaller.

```
termination find
  proof
    let ?R = measures [λ(M, N).length N]
  show wf ?R
    by simp
  fix x x1:: 'a
  fix x2 :: ('a,'b) trm
  fix v :: ('a × ('a,'b) trm ref)
  fix va :: ('a,'b) heap
  show⟨get x (v # va) = Ref x2 ⟹
      x2 = Var x1 ⟹
      ((x1, remove (x, Ref (Var x1)) (v # va)), x, v # va)
                    ∈ measures [λ(M, y). length y]⟩
    apply auto
    by (metis is-in0 prod.collapse remove1)
qed
```

Figure 4.13: Termination Proof for Find

This shows that even if `Find` does not terminate in the current iteration, the length of the heap will converge towards 0, where termination is guaranteed.

Thus termination has been proven for the function `Find`.

4.3 Testing

To make sure the algorithm worked as expected a number of tests were defined. These tests use the Value feature in Isabelle/HOL, which returns the value of the functions. Each test has the expected output written as a comment next to it. It should be noted that due to the complexity of some of the functions, the *char list* was used to instead of the undefined *'a*. Some early testing deviated from expected results, which helped point out implementation mistakes. All tests have been separated into their own .thy file called test.thy and can be seen in Appendix B.

For each of the core functions a list of tests has been written for the testing. These cases range from trivial to edge cases. The test cases of non-trivial functions are:

- Find
  - with empty heap.
  - with a reference.
  - with multiple references.
  - without a reference.
  - with a loop. Note that termination was most important for this case.
- Occur
  - with an empty heap.
  - with an occur.
  - with an occur, requiring recursion.
  - without an occur.
- Unify
  - with an empty heap.
  - with identical terms.
  - with a variable and a function.
  - with two variables.
  - with two functions.

– with a variable and a function, containing that variable.

As a small note about *char list* in Isabelle/HOL; *char list* is indicated by two ’ apostrophe symbols on either side, not by " quotation marks as in most other languages, since quotation marks are reserved to indicate functions in Isabelle/HOL.
Since Isabelle/HOL does not build incomplete functions, like functions lacking a termination proof such as `Unify` and `Occur`, it was necessary to start a proof of these methods and then close them using the Isabelle/HOL symbol "sorry". This allowed the function to be build and tested.

One of the more interesting tests is the case of `"unify "x" "a" [("x",Ref(Var "a")),("a",Ref(Fun "f" ["x"]))]` which translate to unify x and $f(x)$, when they already are unified in the heap. In this case the algorithm would say this is a current unification. While x and $f(x)$ should never be unified doing a well inputted run, this is a more grey area since they are already unified. This presents a dilemma of proving for not well-formed inputs which will be discussed further in Section 5.2.

Chapter 5

Discussion

In this chapter, we will discuss the findings, observations and challenges discovered during the project.

First arguments for termination and correctness of the implementation will be presented.
The next section elaborates on proving algorithms in Isabelle/HOL and the challenges of proving an algorithm, which can take not well-formed input.
We will then take a look at the result of the algorithm and discuss the lack of an easily readable MGU in both the original algorithm and in the implementation.
Lastly a list of missing features will be presented as inspiration for further work.

5.1 Argument for Termination and Correctness

We will now present some intuitive arguments for the Termination and Correctness of the implementation, since no formal proof has been constructed for either Termination or Correctness in Isabelle/HOL. This was due to time constraints.

Termination

Termination proofs of other unification algorithms rely on the non-cyclic property of the tree-like structure of terms, which ensures that when the terms are processed top-down, an end is guaranteed.
This property is shared with the DAG structure used for terms in this algorithm. Any DAG can be ordered and enumerated such that any node only has directed edges going to nodes of ascending number. This guarantees that at some point an end point of the graph is reached.

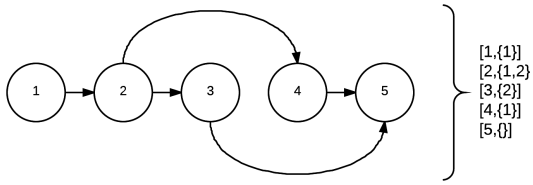

Figure 5.1: An example of an enumerated DAG. Note that the list to the right shows the nodes and their edges represented as increments from the current node. So for example [1,1] means that node 1 has an edge to the enumerated node 1 after 1, i.e 2.

A concern one might have is that union would be able to extend the DAG infinitely by linking to previously visited parts of the graph. This would create a cycle which would cause non-termination.
This is however prevented by the occur check. For the cycle to be created a variable would have to be unified with a previously used function. But since the explicit purpose of the occur check is to prevent variables being unified with functions in which the variables exist, any attempt to unify such a case will end in the algorithm reporting failure and terminating.
However correctness can be affected, if `Unify` is given a heap already containing a variable substituted for a function containing that variable. This will be discussed further in Section 5.2.

Another concern might be that a loop could occur in the list of variable references which could create a loop. Such a loop will not occur naturally due to the use of `Find` before any `Union` call. As an example assume that someone is actively trying to create a loop. This opponent wants to create a small loop between x and y. So first x is unified with y. Now to create the loop, the algorithm will need to run `Union(y,x,h)` where h is the heap that contains the substitution $\{x \to y\}$. But before each call to `Union`, `Find` is called to look-up

the final value in the heap for any given value. This means that after x has been unified with another term, it can never appear in `Union` after that point. So such a loop can not occur in a natural run of the algorithm.
If such a loop should be presented as not well-formed input, the issue has already been addressed and handled in the `Find` method, so that it guarantees termination.

Correctness

Unlike termination where progress has been made in the implementation, a couple large theorems and concepts need to be defined before one would be able to prove the correctness of the implemented algorithm. Thus this section serves to help further formalization of this algorithm.
Correctness for the unification can be broken down to:

1. It returns a boolean value which is True if and only if the two terms can be unified and False if not.
2. It returns a Heap which contains the minimal amount of substitutions for the two terms to be unified, i.e. an MGU.

The first property can also be written as:

$$Unify\ t_1\ t_2\ h = TRUE \Leftrightarrow \exists \mu\ t_1\mu = t_2\mu \tag{5.1}$$

The argumentation for this property will be split into two parts:

$$Unify\ t_1\ t_2\ h \Rightarrow \exists \mu\ t_1\mu = t_2\mu \tag{5.2}$$

First I would argue that if the `Unify` function finds a substitution for t_1 and t_2 then a substitution that makes t_1 and t_2 identical exists. This look trivial, but it means we have to make sure that the substitution found by the function successfully unifies the two terms.

$$\exists \mu\ t_1\mu = t_2\mu \Rightarrow Unify\ t_1\ t_2\ h \tag{5.3}$$

Secondly I would argue that if there exist a substitution that makes t_1 and t_2 identical then the `Unify` function will find a substitution for t_1 and t_2.

What a unifier, and more specifically what an MGU, is needs to be defined clearly before the second property can be proven correct. A unifier would be

defined as a heap h that when applied on two termst_1 t_2, would make them identical:

$$\texttt{Unifier}\ t_1\ t_2\ h\ \Leftrightarrow \texttt{Substitution}\ t_1\ h\ = \texttt{Substitution}\ t_2\ h \tag{5.4}$$

An MGU will then be defined as:

$$\begin{aligned}&\texttt{MGU}\ t_1\ t_2\ h\ \Leftrightarrow\\ &\quad(\texttt{Unifier}\ t_1\ t_2\ h\ \wedge(\forall h_1.\ \texttt{Unifier}\ t_2\ h_1 \Rightarrow (\exists h_2.\ h_1 = \texttt{Union}\ h\ h_2)))\end{aligned} \tag{5.5}$$

With these definitions can the following theorem be proposed:

$$\texttt{Unify}\ t_1\ t_2\ h\ = s \Rightarrow \texttt{MGU}\ s\ t_1\ t_2 \tag{5.6}$$

If Theorem 5.1 and Theorem 5.6 could both be proven would a correctness proof have been constructed for the quadratic algorithm.

Note that the current implementation does not result in an MGU. This is discussed further in Section 5.3.

5.2 Proving for Not Well-formed input

A detail not considered at the start of the project was the concept of not well-formed input that could be passed to the functions. E.g. the `unify` method takes a heap and two pointer values as input. This heap as it is implemented could contain cycles and thus be not well-formed. However cycles should not occur when terms are expressed naturally, because their structure resembles a tree. But if a user where to give not well-formed input such as seen in the test showed in Figure it could easily result in an infinite loop that compromised termination or correctness, depending on implementation.

An algorithm might be allowed to handle some not well-formed input in normal software as long as it is guaranteedthat through natural use of the software the algorithm will never receive these not well-formed inputs. In most imperative programming languages the concept of private and public methods can be used to achieve this effect. This concept does not exist in Isabelle/HOL.
The workarounds discovered were to either:

1. change the algorithm so that it can handle or discard this not well-formed input. This has been applied to the find method in the design used in the implementation. This raises the question of whether termination has been proven for the original algorithm or whether it shows a weakness in the original design?

2. to limit proofs to well-formed input. At the start of a proof an assumption can be made that the input, which the method receives, is well-formed and simply prove the theorem using this assumption. This should still be all one theoretically needs to feel confident in one's algorithm since the well-formed input can be guaranteed through other means.

An alternative approach could also be to model the data structure in such a way that there can not be not well-formed inputs. This could be done by using the DAG structure of the heap with natural numbers as the pointer values. The next step would be to sort the heap and let the value in the terms be how many nodes forward the next value is. Since this approach resembles a DAG structure and would eliminate the possibility of loops, it is a viable option. An example of this DAG structure can be seen in Figure 5.1 on page 32.
However, a number of wrapper methods would be needed to sort and maintain the DAG as the algorithm ran. The run-time of these methods have not been derived and this data-structure has not been implemented.

5.3 Finding the Most General Unifier

A problem surrounding the implemented algorithm is that the requirements of a unification algorithm is two fold:

1. It has to be able to answer whether it is possible to unify the two terms or not.
2. It has to be able to return a most general unifier.

While the first requirement is solved, there are still some problems with the second requirement. The current implementation abstracts the problem to a point where reading the most general unifier is not a simple task. This is not unique to this implementation. The original design from the book share this design, returning only a Boolean value.

Some wrapper functions could make the algorithm easier to use. It would help with an initial function that could take functions written as strings such as "$f(x)$" and convert it into a heap the could be used as input for the algorithm like "`[(a,Ref(Fun f [x]))]`". And it could also use a concluding function that could take the result heap and clean it up so that could be an MGU. With these wrapper functions could this algorithm be considered a complete unification algorithm.

An alternative implementation has been implemented, which works more like a normal substitution unification algorithm. But instead of substituting each time, it will instead use the `Find` on the substitution heap. The heap resulting from this unification should be an MGU if the algorithm does indeed create a most general unifer. Unfortunately, due to time constraints, this has not been proven in this project.

5.4 Future Work

Here is a list of tasks this project did not get to, which could be worked on in the future:

1. The formal proofs for correctness and termination still needs to be implemented. While termination should be fairly easy to prove with a few lemmas, correctness would need some work. One would have to define an MGU, and prove the result always would be MGU.
2. In the book, where the initial algorithm was found, there are also details for reducing the worst case run-time to it being almost linear. It should be possible to modify the current implementation such that this would also be the case in the implementation. Like the quadratic modifications, these changes do not strictly increase run-time, but only guarantee better worst case scenarios.
3. If the algorithm is to see any practical use, it will need a parser that can take two normally written terms, preferably as strings, and parse them to the heap format needed for the algorithm. Some experiments with this has been made and can be seen in Appendix A. These experiments tried to clean up the output heap.
4. If an interest in increasing the efficiency of the algorithm is taken, one could take closer look at the data-structure used for the heap. Currently a standard linked list is used, which does not have the best possible run-times for operations like `Union`, `Is_in` and `Remove`. This hurts the general run-time of `Unify`.

Chapter 6

Conclusion

A DAG-based unification algorithm from the book "Term rewriting and all that" has been implemented in Isabelle/HOL. Due to the imperative nature of the original algorithm, the data-structure and functions used has been refitted for the not imperative language of Isabelle/HOL. A design for implementing the DAG to store the terms has been inspired by the article "Formal Correctness of a Quadratic Unification Algorithm". The goal of implementing a worst case quadratic run-time algorithm was not met due to the use of data-structures access functions with inefficient run-times.

While a formal proof was not made for neither the unification algorithm termination nor correctness, a formalization for termination of an auxiliary function `Find` has been described. The formalization process leads to a discussion on formalizing algorithms in Isabelle/HOL. For future formalizations of algorithms it is recommended that before the algorithm is implemented that the programmer reflects over how to handle these not well-formed input. One such reflection is whether the programmer wants to use assumptions in the proof construction or he wants to implement the algorithm such that there are no wrong inputs. This is a choice he has to make.

The formalization process also highlighted the not ideal unifier produced by the original algorithm. It is understandable, since it originally only was intended to return a Boolean. While this is all that might be needed in some cases, it

does make the algorithm less useful in other implementations where the unifier is important. Further development could be to work on an efficient parsing of the resulting heap, such that all unnecessary elements were removed. This could allow the unification algorithm to result in a proper *MGU*.

Appendix A

Code

1 Datatypes

datatype $('v,'f)\ trm =$
  $Var\ 'v$
| $Fun\ 'f\ \langle'v\ list\rangle$

datatype $'a\ ref = Null \mid Ref\ 'a$

type-synonym $('v,'b)\ heap = \langle(\ 'v \times\ ('v,'b)\ trm\ ref)\ list\rangle$

primrec $remove :: 'v \times ('v,'b)\ trm\ ref \Rightarrow\ ('v,'b)\ heap\ \Rightarrow ('v,'b)\ heap$
  where
  $remove\ m\ [] = []$
| $remove\ m\ (x\#xs) = (if\ m = x\ then\ xs\ else\ x\#(remove\ m\ xs))$

primrec $exist :: ('v,'f)\ trm \Rightarrow\ ('v,'f)\ heap\ \Rightarrow bool$
  where
  $exist\ m\ [] = False$
| $exist\ m\ (x\#xs) = (if\ Ref\ m = (snd\ x)\ then\ True\ else\ (exist\ m\ xs))$

primrec $get :: \langle'v \Rightarrow ('v,'f)\ heap \Rightarrow ('v,'f)\ trm\ ref\rangle$
where
  $\langle get\ x\ [] = Null\rangle$
| $\langle get\ x\ (p\#t) = (if\ x = fst\ p\ then\ snd\ p\ else\ get\ x\ t)\rangle$

primrec $is\text{-}in :: \langle'v \times ('v,'f)\ trm\ ref \Rightarrow ('v,'f)\ heap \Rightarrow bool\rangle$
where
  $\langle is\text{-}in\ x\ [] = False\rangle$
| $\langle is\text{-}in\ x\ (p\#t) = (if\ x =\ p\ then\ True\ else\ is\text{-}in\ x\ t)\rangle$

2 Find

function(*sequential*) $find :: \langle'v\ \Rightarrow ('v,'f)\ \ heap \Rightarrow ('v,'f)\ trm\rangle$
  where
  $\langle find\ x\ [] = Var\ x\rangle$
| $\langle find\ x\ h = (case\ get\ x\ h\ of$
        $Null \Rightarrow Var\ x$
      $\mid Ref\ (Fun\ f\ args) \Rightarrow Fun\ f\ args$
      $\mid Ref\ (Var\ v) \Rightarrow find\ v\ (remove\ (x,\ Ref\ (Var\ v))\ h))\rangle$
   by *pat-completeness auto*

3 Union

primrec $unin :: \langle'v \Rightarrow ('v,'f)\ trm \Rightarrow ('v,'f)\ heap \Rightarrow ('v,'f)\ heap\rangle$
  where
  $\langle unin\ x\ y\ [] = [(x,Ref\ y)]\rangle$
| $\langle unin\ x\ y\ (t\#ts)\ \ = (if\ x{=}fst\ t\ then\ (x,Ref\ y)\#ts\ else\ t\#(unin\ x\ y\ ts))\rangle$

4 Occur check

```
function(sequential) occ :: ⟨'v ⇒ 'v ⇒('v,'f) heap ⇒(('v,'f) heap × bool)⟩
and occs :: ⟨'v ⇒ 'v list ⇒('v,'f) heap ⇒ (('v,'f) heap ×bool)⟩
where
  ⟨occ u v h =  (case ((find u h),(find v h)) of
        (a,Var v) ⇒ (h,False)
      | (a,Fun f args) ⇒ (occs u args (remove (v, Ref (Fun f args)) h)))⟩
| ⟨occs u [] h = (h,False)⟩
| ⟨occs u (v#vs) h = (let oc= occ u v h
                      in (if (u = v) ∨ snd(oc)
                          then (fst(oc),True)
                          else occs u vs (fst(oc))))⟩
   by pat-completeness auto
termination
  sorry

fun occurs :: ⟨'v ⇒ 'v list ⇒('v,'f) heap ⇒ bool⟩
  where
    ⟨occurs u v h = snd( occs u v h)⟩
```

5 Unification Algorithm

```
function(sequential) unify :: ⟨'v ⇒ 'v  ⇒ ('v,'f) heap ⇒ (('v,'f) heap ×
bool)⟩
and unifys :: ⟨'v list ⇒ 'v list ⇒ ('v,'f) heap ⇒ (('v,'f) heap × bool)⟩
where
  ⟨unify v u h = (if v=u then (h,True) else (case ((find v h),(find u h)) of
     (Var v1,Var v2) ⇒
       (if v1 = v2 then (h,True) else ((unin v1 (Var v2) h),True))
   | (Var v2,Fun f args) ⇒
       (if (occurs v2 args h) then (h,False) else ((unin v2 (Var u) h),True))
   | (Fun f args,Var v2) ⇒
       (if (occurs v2 args h) then (h,False) else ((unin v2 (Var v) h),True))
   | (Fun f1 args1,Fun f2 args2) ⇒
       (if f1=f2 then unifys args1 args2 (unin v (Var u) h) else (h,False))))⟩
| ⟨unifys [] [] h = (h,True)⟩
| ⟨unifys [] (v2#vs2) h = (h,False)⟩
| ⟨unifys (v1#vs1) [] h = (h,False)⟩
| ⟨unifys (v1#vs1) (v2#vs2) h =
          (let u = unify v1 v2 h
           in (if (snd u) then (unifys vs1 vs2 (fst u)) else u))⟩
   by pat-completeness auto
termination
  sorry
```

6 Parser

fun *parse-heap-mgu1* :: ‹ ('v,'f) *heap* ⇒ ('v,'f) *heap* ⇒ ('v × ('v,'f) *trm*) *list*›
where
‹*parse-heap-mgu1* [] *h* = []›
| ‹*parse-heap-mgu1* (*p*#*t*) *h* = (*case p of*
(*a*,*Null*) ⇒ (*parse-heap-mgu1 t h*)
| (*a*,*Ref* (*Var v*)) ⇒ (*case* (*find v h*) *of*
(*Var v2*) ⇒(*a*,*Var v2*)#(*parse-heap-mgu1 t h*)
| (*Fun f args*) ⇒ (*a*,*Fun f args*)#(*parse-heap-mgu1 t h*))
| (*a*,*Ref* (*Fun f args*)) ⇒ (*parse-heap-mgu1 t h*))›

fun *parse-heap-mgu* :: ‹ ('v,'f) *heap* ⇒ ('v × ('v,'f) *trm*) *list*›
where ‹*parse-heap-mgu h* = *parse-heap-mgu1 h h* ›

7 Cycle Detection

fun *no-cycle1* :: ‹('v,'f) *heap* ⇒ ('v,'f) *heap* ⇒ ('v,'f) *heap*›
where
‹*no-cycle1* [] *org* = []›
| ‹*no-cycle1* ((*a*,*b*)#*t*) *org* = (*if exist* (*Var a*) *org then* (*a*,*b*)#(*no-cycle1 t org*) *else no-cycle1 t org*)›

fun *no-cycle* :: ‹ ('v,'f) *heap* ⇒ *bool*›
where
‹*no-cycle* [] = *True*›
| ‹*no-cycle orgL* = (*let uptL*=(*no-cycle1 orgL orgL*)
in (*if length uptL*<*length orgL then no-cycle uptL else False*)
)›

lemma *no-cycleProof*: *length*(*no-cycle1 x org*) ≤ *length x*
apply (*induct x*)
apply *simp*
using *nat-le-linear* **by** *auto*

lemma *no-cycleProof2*: *set*(*no-cycle1* ([(*x*,*y*)]) *org*) ⊆ *set* ([(*x*,*y*)])
by *simp*

lemma *no-cycleProof3*: *set*(*no-cycle1* ([*a*]) *org*) ⊆ *set* ([*a*])
by (*metis list.sel*(*1*) *list.simps*(*3*) *no-cycle1.elims no-cycleProof2*)

lemma *no-cycleProof1*: *set*(*no-cycle1 x org*) ⊆ *set x*
apply (*induct x*)
apply *simp*
by (*smt contra-subsetD dual-order.trans list.discI list.inject list.set-intros*(*1*) *no-cycle1.elims set-ConsD set-subset-Cons subrelI*)

8 Proofs

lemma *remove-proof* : *length* (*remove m h*) ≤ *length h*
by (*induct h*, *simp*, *simp*)

lemma *is-in0* : *get x h* = *Ref x2* ⟹
is-in (*x*,*Ref x2*) *h*
apply (*induct h*,*simp*)
by (*metis get.simps*(*2*) *is-in.simps*(*2*) *prod.exhaust-sel*)

lemma *is-in1* : *get x h* = *Ref x2* ⟹
x2 = *Var x1* ⟹
is-in (*x*,*Ref*(*Var x1*)) *h*
apply (*induct h*,*simp*)
by (*meson is-in0*)

lemma *remove1* : *is-in x h* ⟹(*length*(*remove x h*) < *length h*)
apply (*induct h*,*simp*)
by *simp*

termination *find*
proof
let *?R* = *measures* [λ(*M*, *N*).*length N*]
show *wf ?R*
by *simp*
fix *x x1*:: *'a*
fix *x2* :: (*'a*,*'b*) *trm*
fix *v* :: (*'a* × (*'a*,*'b*) *trm ref*)
fix *va* :: (*'a*,*'b*) *heap*
show⟨*get x* (*v* # *va*) = *Ref x2* ⟹
x2 = *Var x1* ⟹
((*x1*, *remove* (*x*, *Ref* (*Var x1*)) (*v* # *va*)), *x*, *v* # *va*)
∈ *measures* [λ(*M*, *y*). *length y*]⟩
apply *auto*
by (*metis is-in0 prod.collapse remove1*)
qed

Appendix B

Test

1 Tests

1.1 Get

value *get* (''x'') ([])
value *get* (''x'') ([(''x'',*Ref* (*Var* ''y''))])
value *get* (''y'') ([(''x'',*Ref*(*Var* ''y''))])

1.2 Remove

value *remove* (''x'',*Ref* (*Var* ''y'')) ([])
value *remove* (''x'',*Ref* (*Var* ''y'')) ([(''x'',*Ref* (*Var* ''y''))])
value *remove* (''y'',*Ref* (*Var* ''y'')) ([(''x'',*Ref* (*Var* ''y''))])
value *remove* (''y'',*Ref*(*Fun* ''f'' [''x'])) ([(''y'',*Ref*(*Fun* ''f'' [''x']))])

1.3 Find

value *find* ''x'' ([])
value *find* ''x'' ([(''x'',*Ref* (*Var* ''y''))])
value *find* ''x'' ([(''x'',*Ref* (*Fun* ''f'' []))])
value *find* ''x'' ([(''x'',*Ref* (*Var* ''y'')),(''y'',*Ref* (*Var* ''z''))])
value *find* ''z'' ([(''x'',*Ref* (*Var* ''y'')),(''y'',*Ref* (*Var* ''z''))])
value *find* ''a'' ([(''x'',*Ref* (*Var* ''y'')),(''y'',*Ref* (*Var* ''z''))])
value *find* ''a'' ([(''a'',*Ref* (*Var* ''y'')),(''y'',*Ref* (*Var* ''a''))])

1.4 Union

value *unin* ''x'' (*Var* ''y'') ([(''x'',*Ref* (*Var* ''y'')),(''y'',*Ref* (*Var* ''z''))])
value *unin* ''z'' (*Var* ''x'') ([(''x'',*Ref* (*Var* ''y'')),(''y'',*Ref* (*Var* ''z''))])

1.5 Occur

value *occur* ''y'' ''x'' []
value *occur* ''y'' ''x'' [(''x'',*Null*),(''y'',*Null*)]
value *occur* ''y'' ''x'' [(''x'',*Null*),(''y'',*Ref*(*Fun* ''a'' []))]
value *occur* ''x'' ''y'' [(''x'',*Null*),(''y'',*Ref*(*Fun* ''f'' []))]
value *occur* ''x'' ''y'' [(''x'',*Null*),(''y'',*Ref*(*Fun* ''f'' [''x']))]
value *occur* ''x'' ''y'' [(''x'',*Null*),
(''y'',*Ref*(*Fun* ''f'' [''z'])),
(''z'',*Ref*(*Fun* ''f'' [''x']))]
value *occur* ''x'' ''y'' [(''x'',*Null*),
(''y'',*Ref*(*Var* ''z'')),
(''z'',*Ref*(*Fun* ''f'' [''x']))]

1.6 Unification

value *unify* ''y'' ''x''
[]

```
value unify ''y'' ''x''
        [(''x'',Null),
         (''y'',Null)]
value unify ''x'' ''x''
        [(''x'',Null)]
value unify ''x'' ''a''
        [(''x'',Null),
         (''a'',Ref(Fun ''f'' []))]
value unify ''x'' ''a''
        [(''x'',Null),
         (''a'',Ref(Fun ''f'' [''x'']))]
value unify ''x'' ''a''
        [(''x'',Ref(Var ''a'')),
         (''a'',Ref(Fun ''f'' [''x'']))]
value unify ''a'' ''b''
        [(''a'',Ref(Fun ''f'' [])),
         (''b'',Ref(Fun ''f'' []))]
value unify ''x'' ''a''
        [(''x'',Null),
         (''a'',Ref(Fun ''f'' [''x'']))]

value unify ''a'' ''b''
        [(''a'',Ref(Fun ''f'' [''x'])),
         (''b'',Ref(Fun ''f'' [''x'']))]

value unify ''a'' ''b''
        [(''a'',Ref(Fun ''f'' [''x'',''y'])),
         (''b'',Ref(Fun ''f'' [''z'',''v']))]

value unify ''a'' ''b''
        [(''x'',Null),(''y'',Null),(''z'',Null),(''v'',Null),
         (''a'',Ref(Fun ''f'' [''x'',''y'])),  (*Should return: False,*)
         (''b'',Ref(Fun ''g'' [''z'',''v']))]

value unify ''a'' ''b''
        [(''a'',Ref(Fun ''f'' [''x'',''x'])),
         (''b'',Ref(Fun ''f'' [''z'',''v']))]

value unify ''a'' ''b''
        [(''x'',Null),(''y'',Null),(''z'',Null),(''v'',Null),
         (''a'',Ref(Fun ''f'' [''x'',''y'',''z'])),
         (''b'',Ref(Fun ''f'' [''z'',''v'',''x']))]

value unify ''a'' ''b''
        [(''x'',Null),(''y'',Null),(''z'',Null),
         (''a'',Ref( Fun ''f'' [''x'',''y'',''z'])),
         (''b'',Ref( Fun ''f'' [''c'',''z'',''x'])),
         (''c'',Ref( Fun ''g'' []))]

value unify ''a'' ''b''
```

```
          [(''x'',Null),(''y'',Null),(''z'',Null),
           (''a'',Ref( Fun ''f'' [''x'',''y'',''z'])),
           (''b'',Ref( Fun ''f'' [''c'',''z'',''x'])),
           (''c'',Ref( Fun ''g'' [''z'']))]

value unify ''x'' ''y''
          [(''y'',Ref(Var ''x'')),
           (''x'',Ref(Var ''y''))]


value unify ''a'' ''b''
          [(''y'',Null),(''x'',Null),(''z'',Null),
           (''a'',Ref( Fun ''f'' [''c'',''x'])),
           (''b'',Ref( Fun ''f'' [''d'',''d'])),
           (''c'',Ref( Fun ''g'' [''z'])),
           (''d'',Ref( Fun ''g'' [''y'']))]
```

1.7 Cycle

```
value no-cycle []
value no-cycle [(''a'',Ref (Var ''b''))]
value no-cycle [(''a'',Ref (Var ''a''))]
value no-cycle [(''a'',Ref (Var ''b'')),(''b'',Ref (Var ''a''))]
value no-cycle [(''a'',Ref (Var ''b'')),(''b'',Ref (Var ''c'')),
                (''c'',Ref (Var ''a''))]
```

1.8 Parser

```
definition test-mgu :: (char list,char list) heap
  where test-mgu=fst (unify ''a'' ''b''
          [(''y'',Null),(''x'',Null),(''z'',Null),
           (''a'',Ref( Fun ''f'' [''c'',''x'])),
           (''b'',Ref( Fun ''f'' [''d'',''d'])),
           (''c'',Ref( Fun ''g'' [''z'])),
           (''d'',Ref( Fun ''g'' [''y'']))])

value test-mgu
value parse-heap-mgu test-mgu
```

Bibliography

[BA12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer, third edition edition, 2012.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Springer, 1999.

[Coq] Description of Coq. `https://coq.inria.fr/about-coq`. [accessed 26-06-2017].

[Isa] Description of Isabelle/HOL. `http://isabelle.in.tum.de/overview.html`. [accessed 26-12-2017].

[JLRR06] José-Antonio Alonso María-José Hidalgo José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos. Formal correctness of a quadratic unification algorithm. *Journal of Automated Reasoning*, 37(1-2):67–92, August 2006.

[Kra07] Alexander Krauss. Dening recursive functions in Isabelle/HOL. *Department of Informatics, Technische Universität München*, 2007.

[MN05] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1):200 – 227, 2005.

[Pro] Formalizing 100 Theorem. `http://www.cs.ru.nl/~freek/100/`. [accessed 26-06-2017].

[RN14] Herman Geuvers Rob Nederpelt. Type theory and formal proof. *Cambridge University Press*, page 385, 2014.

[Thu94] William P. Thurston. On proof and progress in mathematics. *Bulletin (New Series) of the American Mathematical Society*, 30(2):161–177, April 1994.