Bachelor Thesis

KBCV for Android

Christina Kohl
christina.kohl@student.uibk.ac.at

2 April 2017

Supervisor: Thomas Sternagel

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Datum	Unterschrift

Abstract

This thesis deals with the Knuth-Bendix completion procedure, which is a method to transform equational systems into equivalent complete rewrite systems, and it also covers Scala development on Android as well as the Android design guidelines. The goal of the underlying project was to bring the Knuth-Bendix Completion Visualizer (KBCV) to the Android platform and create an application that is visually appealing and easy to use. KBCV on Android is aimed at students who are studying term rewrite systems and gives them the opportunity to interactively perform Knuth-Bendix completion on their Android devices.

Contents

1 Introduction

Term rewrite systems are a fundamental concept of mathematics and computer science. They can be used to decide equivalence problems and to build automated theorem provers. They are also at the foundation of some functional programming languages [2]. When dealing with term rewriting we are mostly interested in systems that fulfill certain properties, namely termination and confluence, which can be ensured by completion. The original completion procedure was introduced by Knuth and Bendix, see [14], and transforms an equational system into a terminating and confluent term rewrite system. The procedure was later reformulated as an inference system by Bachmair [3] and that is what students learn today when it comes to completion. However applying the inference rules by hand can be tedious and sometimes leads to mistakes, which are hard to identify and correct. To help students with learning completion, the Knuth-Bendix Completion Visualizer (KBCV for short) was developed by Thomas Sternagel [25]. The purpose of this project was to bring KBCV to the Android platform.

This thesis not only covers the basics of term rewriting and the completion procedure, it also contains some Android related chapters. First we will elaborate on our motivation for this project and the goals we wanted to achieve. Also the existing work in form of the original KBCV tool is discussed in this first chapter. Chapter 2 then starts off with an introduction to equational systems and term rewrite systems for readers who are not yet familiar with these topics. In Chapter 3 we describe the completion procedure. This chapter also contains a detailed example on how to apply the completion procedure in practice. The following chapters are more focused on Android. First, in Chapter 4 we deal with the process of developing Android projects in Scala instead of Java. In Chapter 5 we talk about the official Android design guidelines, and how they were applied in our project. Chapter 6 is dedicated to a detailed description of the actual Android application that was being developed as part of this project. We sum up the thesis in Chapter 7 and conclude with some ideas for possible future work.

1.1 Motivation

Like the original KBCV tool, KBCV on Android is intended for students studying term rewriting and completion. We had especially the students that are attending the term rewriting course at the University of Innsbruck in mind. KBCV on Android allows them to interactively perform completion and thereby makes it easier for them to practice and understand the procedure. We decided to develop an Android application on top of the already existing tool because tablets and smartphones have become increasingly popular but the original KBCV tool can only be used as a desktop application or as a

Java web applet, both of which do not work on mobile devices. Since the market share of Android is a multiple of that of iOS or Windows [9], the decision to target Android seemed obvious.

1.2 Goals

We not only wanted a working interactive completion visualizer, we also wanted the application to be visually appealing and easy to use. The user interface of the app should be attuned to touch gestures and look good on devices of various screen sizes. Users should be immediately able to work with it, without needing a lengthy introduction first. The app should also provide all of the main features of the original KBCV tool, like interactive mode, automatic completion, adaptable LPO precedence, creation of equational systems, and import and export options.

1.3 Related Work

There are no other mobile applications for term rewriting, that we could find. There are however several desktop applications, first and foremost the Knuth-Bendix Completion Visualizer by Thomas Sternagel [25]. The handy library *termlib*, that deals with term rewriting and completion in general, is part of that project and KBCV on Android builds heavily on it. Some other tools are `mkbTT` which is also being developed at the University of Innsbruck, see [27], or `Slothrop` by Wehrmann, Stump and Westbrook described in [26].

2 Preliminaries

The goal of this chapter is to give people who are not yet familiar with term rewriting a basic understanding of the topic. We will start off with the concept of equational logic, which is the basis of term rewrite systems in Section 2.1. Then in Section 2.2 we will look at term rewrite systems and their properties. Please note that we will only cover the concepts needed for understanding completion. For more detailed information about term rewrite systems refer to [20, 2].

2.1 Equational Logic

Equational logic is a subset of first-order logic, with equality ($\approx$) as the only predicate symbol. Equational logic not only plays an important role in term rewriting, but also in the area of mathematical algebra in general and is for example used for automated theorem proving.

We will first introduce *terms* as the basic building blocks with which we can then define equations and equational systems.

2.1.1 Terms

Terms are built up from variables, constants and function symbols applied to terms.

Definition 2.1. A *signature* $\mathcal{F}$ is a set of *function symbols*. Associated with every $f \in \mathcal{F}$ is a natural number denoting its *arity*, i.e. the number of arguments that function is supposed to have. Function symbols of arity 0 are called *constants*.

Definition 2.2. Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a countably infinite set of variables disjoint from $\mathcal{F}$. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of *terms* built from $\mathcal{F}$ and $\mathcal{V}$ is the smallest set such that

- every variable is a term
- and if $f \in \mathcal{F}$ is a function symbol of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

Conventionally function symbols of arity greater than 0 are denoted by f, g, h, constants by a, b, c, variables by x, y, z, and terms by s, t, u. For constants we omit the empty parentheses, so we write a instead of $a()$.

We often need a means of talking about specific subterms of a given term. For that purpose we introduce the concept of *positions*.

Definition 2.3. A *position* is a finite sequence of positive integers. The *root position* is the empty sequence and denoted by ϵ. We denote the concatenation of positions p and q by the juxtaposition pq. We say that position p is *above* position q if there exists a (necessarily unique) position r such that $pr = q$ and we write $p \leq q$. If $p \leq q$ we also say that q is *below* p or p is a *prefix* of q. If $p \leq q$ and $p \neq q$ we write $p < q$ and say that p is a *proper prefix* of q. If neither $p \leq q$ nor $q \leq p$, then positions p and q are said to be *parallel.*

The following definition describes how to obtain the position for a specific subterm and gives a notation for replacing specific subterms with other terms.

Definition 2.4. The set $\mathcal{P}\mathsf{os}(t)$ of all *positions* of a term t is defined recursively:

$$\mathcal{P}\mathsf{os}(t) = \begin{cases} \{\epsilon\} & \text{if } t \text{ is a variable} \\ \{\epsilon\} \cup \{ip \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}\mathsf{os}(t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Let $p \in \mathcal{P}\mathsf{os}(t)$. The subterm of t at position p is denoted by $t|_p$, i.e.,

$$t|_p = \begin{cases} t & \text{if } p = \epsilon \\ t_i|_q & \text{if } t = f(t_1, \ldots, t_n) \text{ and } p = iq \end{cases}$$

$\mathcal{P}\mathsf{os}_{\mathcal{V}}(t) = \{p \in \mathcal{P}\mathsf{os}(t) \mid t|_p \in \mathcal{V}\}$ denotes the set of all variable positions in t.
$\mathcal{P}\mathsf{os}_{\mathcal{F}}(t) = \mathcal{P}\mathsf{os}(t) \setminus \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$ denotes the set of all non-variable positions in t.
If s is a term then $t[s]_p$ denotes the term that is obtained from t by replacing the subterm at position p by the term s. Formally:

$$t[s]_p = \begin{cases} s & \text{if } p = \epsilon \\ f(t_1, \ldots, t_i[s]_q, \ldots, t_n) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } p = iq \end{cases}$$

Example 2.5. Let $t = \mathsf{f}(\mathsf{c}, \mathsf{g}(\mathsf{f}(x, \mathsf{g}(x))))$. The positions of this term are depicted in Figure 2.1. The two positions of the subterm x are 211 and 2121. The subterm of t at position 212 is $t|_{212} = \mathsf{g}(x)$ and $t[\mathsf{g}(x)]_{211} = \mathsf{f}(\mathsf{c}, \mathsf{g}(\mathsf{f}(\mathsf{g}(x), \mathsf{g}(x))))$.

Next we introduce the notion of *contexts*, which can be viewed as terms containing a hole.

Definition 2.6. A *context* is a term containing exactly one occurence of the special constant symbol $\Box$, which is called *hole.* If C is a context then $C[t]$ denotes the result of replacing the hole by t. A binary relation R is closed under contexts if $C[s] \mathrel{R} C[t]$ for all contexts C and terms s, t with $s \mathrel{R} t$.

To be able to work with equations we need one more key concept, that is, how we can *instantiate* variables with terms.

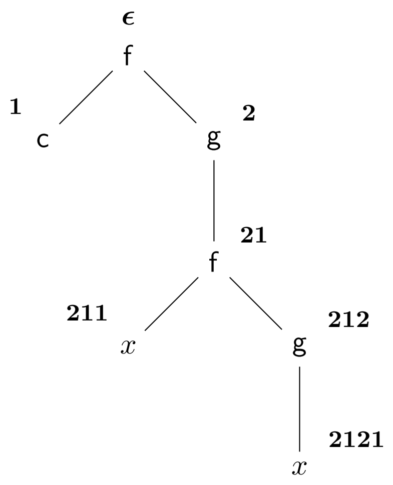


Figure 2.1: Positions in a term.

Definition 2.7. A *substitution* is a mapping σ from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F},\mathcal{V})$ with the property that its domain $\mathcal{D}om(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. We write $t\sigma$ for the result of applying the substitution σ to the term t:

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \text{ is a variable} \\ f(t_1\sigma, \dots, t_n\sigma) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

We call $t\sigma$ an *instance* of t. A *renaming* is a bijective substitution from $\mathcal{V}$ to $\mathcal{V}$. A term s is a *variant* of a term t if $s = t\sigma$ for some renaming σ.

Example 2.8. Consider the term $t = \mathsf{f}(x, \mathsf{f}(\mathsf{g}(x), y))$ and the substitution $\sigma = \{x \mapsto \mathsf{c}, y \mapsto \mathsf{g}(x)\}$. The result of applying σ to t is $t\sigma = \mathsf{f}(\mathsf{c}, \mathsf{f}(\mathsf{g}(\mathsf{c}), \mathsf{g}(x)))$. The term $s = \mathsf{f}(z, \mathsf{f}(\mathsf{g}(z), x))$ is a variant of t because $s = t\tau$ for the renaming $\tau = \{x \mapsto z, y \mapsto x\}$.

2.1.2 Equational Reasoning

Equational reasoning deals with the manipulation of terms by substituting equals for equals and thereby deriving new theorems from a set of equations.

Definition 2.9. An *equation* is a pair (s,t) of terms, written as $s \approx t$.

This means that the two terms s and t are supposed to be equal with respect to the inference rules introduced in Figure 2.2. We will later see how we can use these inference rules to derive new equations that are logical consequences from a given set of equations.

Definition 2.10. A *unifier* or *solution* of an equation $s \approx t$ is a substitution σ such that $s\sigma = t\sigma$, where $=$ denotes the syntactic equality. If there exists a solution for $s \approx t$ we also say that s and t are *unifiable*.

$$
\begin{array}{llll}
[\mathrm{r}] & \overline{t \approx t} & [\mathrm{t}] & \dfrac{s \approx t \quad t \approx u}{s \approx u} \\[2ex]
[\mathrm{s}] & \dfrac{s \approx t}{t \approx s} & [\mathrm{a}] & \dfrac{}{s\sigma \approx t\sigma}\; s \approx t \in \mathcal{E} \text{ and } \sigma \text{ a substitution} \\[2ex]
[\mathrm{c}] & \dfrac{s_1 \approx t_1 \quad \ldots \quad s_n \approx t_n}{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)} & &
\end{array}
$$

Figure 2.2: Inference rules of equational logic.

Most of the time we are not just interested in any unifier, we want a *most general unifier*.

Definition 2.11. A substitution σ is a *most general unifier* (mgu) of s and t if it unifies s and t and for any unifier τ of s and t there is a unifier ρ such that $\sigma\rho = \tau$.

We can think of the mgu σ as the least specific unifier. Any other unifier can be obtained from σ by substituting for some of its variables. Technically there can exist more than one mgu for an equation, but they are all the same up to variable renaming.

Definition 2.12. An *equational system* (ES for short) is a pair $(\mathcal{F}, \mathcal{E})$ consisting of a signature $\mathcal{F}$ and a set $\mathcal{E}$ of equations between terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

Often the signature is not explicitly provided, since it can be easily read from the given equations (assuming the signature consists only of the function symbols occurring in the equations).

Definition 2.13. The *equational theory* of an ES $\mathcal{E}$ is the set of all equations $s \approx t$ such that $s \approx t$ is derivable from $\mathcal{E}$. The *word problem* for a given ES $\mathcal{E}$ is the question whether an arbitrary equation $s \approx t$ belongs to the equational theory of $\mathcal{E}$.

We can use the set of inference rules in Figure 2.2 to prove that an equation belongs to the equational theory of an ES $\mathcal{E}$.

Example 2.14. Consider the following equations over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$,

$$\mathcal{E} = \{\mathsf{f}(\mathsf{a}, x) \approx \mathsf{g}(x), \mathsf{f}(x, \mathsf{b}) \approx \mathsf{h}(x)\}$$

Following is a proof that $\mathsf{g}(\mathsf{b}) \approx \mathsf{h}(\mathsf{a})$ follows from $\mathcal{E}$:

$$
[\mathrm{t}]\,\dfrac{[\mathrm{s}]\,\dfrac{[\mathrm{a}],\,\sigma_1\,\dfrac{\mathsf{f}(\mathsf{a}, x) \approx \mathsf{g}(x) \in \mathcal{E}}{\mathsf{f}(\mathsf{a}, \mathsf{b}) \approx \mathsf{g}(\mathsf{b})}}{\mathsf{g}(\mathsf{b}) \approx \mathsf{f}(\mathsf{a}, \mathsf{b})} \qquad \dfrac{\mathsf{f}(x, \mathsf{b}) \approx \mathsf{h}(x) \in \mathcal{E}}{\mathsf{f}(\mathsf{a}, \mathsf{b}) \approx \mathsf{h}(\mathsf{a})}\,[\mathrm{a}],\,\sigma_2}{\mathsf{g}(\mathsf{b}) \approx \mathsf{h}(\mathsf{a})}
$$

$$\sigma_1 = \{x \mapsto \mathsf{b}\}, \sigma_2 = \{x \mapsto \mathsf{a}\}$$

The rules [r], [s], and [t] stand for *reflexivity*, *symmetry*, and *transitivity*, respectively. The rule [a] refers to the *application* of some substitution to an equation of $\mathcal{E}$. Rule [c] denotes *congruence*, it states that an equation with the same root function on the left- and right-hand side can be proved equal if the respective arguments are equal to each other. Usually we try to build proof trees from the bottom up. We start with the equation we want to derive and split off branches as needed. Each branch has to be terminated by applying either [r] or [a] at the top. Note that although constructing proof trees may work for many instances of the word problem, the word problem is undecidable in general. Proving that an equation belongs to the equational theory of an ES $\mathcal{E}$ with this method is also quite inefficient and it is hard to say whether an equation is a logical consequence of $\mathcal{E}$ or not. With term rewrite systems we can solve many instances of the word problem much more efficiently, although the word problem itself remains undecidable of course.

2.2 Term Rewrite Systems

Term rewrite systems are a specialization of equational systems, where every equation must fulfill certain conditions, that make it a rewrite rule.

Definition 2.15. A *rewrite rule* is an equation $l \approx r$ that satisfies the following two restrictions:

- the left-hand side l is not a variable
- the variables which occur in the right-hand side r also occur in l

Rewrite rules will be written as $l \to r$. A *term rewrite system* (TRS for short) is an ES with the property that all its equations are rewrite rules.

We can *rewrite* a term s to term t if there exists a rewrite rule $l \to r \in \mathcal{R}$, a substitution σ, and a position $p \in \mathcal{P}\mathsf{os}(s)$, such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. In this case we call the term s *reducible*. The subterm $l\sigma$ of s at position p is called a *redex* and we say that s rewrites to t by *contracting* redex $l\sigma$. We write $s \to t$ for short. The relation induced by $\to$ is closed under contexts and substitutions and we call it a *rewrite relation*.

If the term s is not reducible, i.e., no rewrite rule is applicable, we say that s is in *normal form*.

When we are not interested in intermediate rewrite steps we can abbreviate a rewrite sequence like $s \to s_1 \to \cdots \to t$ by $s \to^* t$, where $\to^*$ denotes the reflexive and transitive closure of $\to$.

2.2.1 Termination

The concept of termination deals with the idea, that when we use a TRS to rewrite terms, we want to be sure that we will reach a normal form at some point. In this section we will first define what termination is formally and then describe a method that can be used to prove termination of rewrite systems.

Definition 2.16. A term t is called *terminating* if there are no infinite rewrite sequences starting at t. A TRS R is terminating if all terms over its signature $\mathcal{F}$ are terminating.

A simple example of non-termination is a TRS containing the rule $\mathsf{f}(x,y) \to \mathsf{f}(y,x)$. Here we have the infinite rewrite sequence $\mathsf{f}(x,y) \to \mathsf{f}(y,x) \to \mathsf{f}(x,y) \to \dots$

Although termination in general is an undecidable property, we can still prove termination in many cases. This can be done for example with reduction orders.

Definition 2.17. A *reduction order* $>$ is a proper order on terms that is well-founded and closed under contexts and substitutions, i.e.,

- $>$ is irreflexive and transitive
- $>$ admits no infinite descending sequences $s_1 > s_2 > s_3 > \dots$
- if $f \in \mathcal{F}$ and $s > t$ then $f(t_1, \dots, s, \dots, t_n) > f(t_1, \dots, t, \dots, t_n)$
- if σ is a substitution and $s > t$ then $s\sigma > t\sigma$.

Theorem 2.18. *A TRS R is terminating if and only if there exists a reduction order $>$ such that $l > r$ for every rewrite rule $l \to r \in R$.*

There exist several different methods to determine appropriate reduction orders for rewrite systems. KBCV uses a method called *lexicographic path order* (LPO). In LPO the ordering of two terms depends on the ordering of their function symbols. The ordering of the function symbols has to be provided in advance and it is called a *precedence.*

Definition 2.19. A *precedence* is a proper order on a signature.

Definition 2.20. Let $>$ be a precedence. The *lexicographic path order* $>_{lpo}$ on terms is defined inductively: $s = f(s_1, \dots, s_n) >_{lpo} t$ if one of the following alternatives holds:

- $t = f(t_1, \dots, t_n)$ and there exists an $i \in 1, \dots, n$ such that
  - $s_j = t_j$ for all $1 \leq j < i$,
  - $s_i >_{lpo} t_i$, and
  - $s >_{lpo} t_j$ for all $i < j \leq n$,
- $t = g(t_1, \dots, t_m)$, $f > g$, and $s >_{lpo} t_i$ for all $1 \leq i \leq m$, or
- $s_i >_{lpo} t$ or $s_i = t$ for some $1 \leq i \leq n$.

Theorem 2.21. *For any well-founded precedence $>$ the induced lexicographic path order $>_{lpo}$ is a reduction order.*

To prove termination of a given TRS, we therefore need to find a precedence for its signature such that for every rule $l \to r$ in the TRS one of the three cases above holds.

Example 2.22. Consider the TRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{g}(\mathsf{b}) \to \mathsf{c}$$
$$\mathsf{f}(\mathsf{a}, \mathsf{g}(x)) \to \mathsf{f}(x, x)$$

We will show its termination by means of LPO. As precedence we take $\mathsf{g} > \mathsf{f} > \mathsf{a} > \mathsf{b} > \mathsf{c}$. Termination of the first rule is easily shown because we have $\mathsf{g} > \mathsf{c}$ in our precedence and thus the second case in the definition holds. For the second rule we use the third case of the definition. We can prove that $\mathsf{g}(x) >_{lpo} \mathsf{f}(x, x)$ because $\mathsf{g} > \mathsf{f}$ according to our precedence and thus we can conclude that $\mathsf{f}(\mathsf{a}, \mathsf{g}(x)) >_{lpo} \mathsf{f}(x, x)$.

2.2.2 Confluence

When a TRS is terminating we know that for every term we will reach a normal form after finitely many rewrite steps. But we also want to make sure that we will always reach the same normal form, independent of how we apply the available rules. In other words, we want to make sure that there exists a unique normal form for every term. The following definitions will help us to formalize this concept.

Definition 2.23. Two terms s and t are *joinable*, denoted by $s \downarrow t$, if there exists a term u such that $s \to^* u \, {}^*\!\!\leftarrow t$.

Definition 2.24. A term t is *confluent* if for all terms t_1 and t_2 with $t_1 \, {}^*\!\!\leftarrow t \to^* t_2$ we have $t_1 \downarrow t_2$. A TRS $\mathcal{R}$ is confluent if all terms over its signature $\mathcal{F}$ are confluent.

Definition 2.25. A term is called *complete* if it is both terminating and confluent. A TRS $\mathcal{R}$ is complete if all terms over its signature $\mathcal{F}$ are complete.

Now if a TRS is confluent we know that we cannot reach two different normal forms for a term t. Because if t rewrites to t_1 and t_2, t_1 and t_2 must be joinable. But if they are normal forms they cannot be rewritten further, so they have to be equal.

Another important notion for completion is *local confluence*:

Definition 2.26. A term t is *locally confluent* if for all terms t_1 and t_2 with $t_1 \leftarrow t \to t_2$ we have $t_1 \downarrow t_2$. A TRS $\mathcal{R}$ is locally confluent if all terms over its signature are locally confluent.

Local confluence is a weaker property than confluence, but Newman's Lemma states that together with termination it is still enough to ensure confluence.

Lemma 2.27. *(Newman's Lemma). A terminating TRS is confluent if it is locally confluent.*

Since we are only interested in both terminating and confluent TRSs in completion, it suffices to construct a locally confluent TRS and prove that it is terminating. We will use this insight in the next chapter where we will learn about the completion procedure in detail.

3 Completion

Completion is a general method to transform equational systems into equivalent complete rewrite systems. In other words, a completion procedure takes an equational system $\mathcal{E}$ as input and tries to construct a confluent and terminating rewrite system $\mathcal{R}$ from it, which proves the same equations as the original system. We can then prove that an equation $l \approx r$ belongs to the equational theory of $\mathcal{E}$ by reducing both sides to normal form and checking if they are identical. Completion can not always be successful since there exist equational systems for which the word problem is unsolvable. Furthermore there exist equational systems with decidable word problem, but without a complete rewrite system [20].

3.1 Critical Pairs

In the previous section we saw that for proving confluence for a terminating system we only need to be concerned about local confluence. This means that it suffices to consider all possible peaks $t_1 \leftarrow d \rightarrow t_2$. But since many such peaks are trivially joinable it is in fact enough to consider only a kind of *critical peak*.

Consider for example the TRS containing the following two rewrite rules:

$$\mathsf{f}(\mathsf{a}, \mathsf{g}(x)) \rightarrow \mathsf{f}(x, x) \tag{1}$$
$$\mathsf{g}(\mathsf{b}) \rightarrow \mathsf{c} \tag{2}$$

Three possible peaks in this rewrite system are

$$\begin{tikzcd}[column sep=small] & \mathsf{f}(\mathsf{g}(\mathsf{b}), \mathsf{g}(\mathsf{b})) \arrow[dl, "(2)"'] \arrow[dr, "(2)"] & \\ \mathsf{f}(\mathsf{c}, \mathsf{g}(\mathsf{b})) & & \mathsf{f}(\mathsf{g}(\mathsf{b}), \mathsf{c}) \end{tikzcd} \quad \begin{tikzcd}[column sep=small] & \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{g}(\mathsf{b}))) \arrow[dl, "(2)"'] \arrow[dr, "(1)"] & \\ \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{c})) & & \mathsf{f}(\mathsf{g}(\mathsf{b}), \mathsf{g}(\mathsf{b})) \end{tikzcd} \quad \begin{tikzcd}[column sep=small] & \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{b})) \arrow[dl, "(2)"'] \arrow[dr, "(1)"] & \\ \mathsf{f}(\mathsf{a}, \mathsf{c}) & & \mathsf{f}(\mathsf{b}, \mathsf{b}) \end{tikzcd}$$

In the first peak we have two parallel redexes. Contracting one does not destroy the other. We can reduce both sides further and get the same term:

$$\mathsf{f}(\mathsf{c}, \mathsf{g}(\mathsf{b})) \rightarrow \mathsf{f}(\mathsf{c}, \mathsf{c}) \leftarrow \mathsf{f}(\mathsf{g}(\mathsf{b}), \mathsf{c})$$

In the second peak the smaller of the two redexes is contained in the larger one. But still, applying one rule first does not affect the potential to apply the other rule and the two reduced terms are joinable:

$$\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{c})) \rightarrow \mathsf{f}(\mathsf{c}, \mathsf{c}) \;{}^{*}\!\!\leftarrow \mathsf{f}(\mathsf{g}(\mathsf{b}), \mathsf{g}(\mathsf{b}))$$

Only the third peak causes difficulties. Here by contracting one redex we destroy the other one. And it is clear that with only the two rules defined above, the two terms $\mathsf{f}(\mathsf{a},\mathsf{c})$ and $\mathsf{f}(\mathsf{b},\mathsf{b})$ are not joinable. We say that the two redexes in the term $\mathsf{f}(\mathsf{a},\mathsf{g}(\mathsf{b}))$ are *overlapping.*

Definition 3.1. An *overlap* of a TRS $(\mathcal{F},\mathcal{R})$ is a triple $\langle l_1 \to r_1, p, l_2 \to r_2\rangle$ satisfying the following properties:

- $l_1 \to r_1$ and $l_2 \to r_2$ are variants of rewrite rules of $\mathcal{R}$ without common variables,
- $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$,
- l_1 and $l_2|_p$ are unifiable,
- if $p = \epsilon$ then $l_1 \to r_1$ and $l_2 \to r_2$ are not variants.

Every overlap gives rise to a critical pair.

Definition 3.2. Suppose $\langle l_1 \to r_1, p, l_2 \to r_2\rangle$ is an overlap of a TRS $\mathcal{R}$. Let σ be a most general unifier of l_1 and $l_2|_p$. The term $l_2\sigma[l_1\sigma]_p = l_2\sigma$ can be rewritten in two different ways:

$$\begin{tikzcd}
 & l_2\sigma[l_1\sigma]_p = l_2\sigma \arrow[dl, "l_1 \to r_1", "p"'] \arrow[dr, "\epsilon"', "l_2 \to r_2"] & \\
l_2\sigma[r_1\sigma]_p & & r_2\sigma
\end{tikzcd}$$

We call the quadruple $(l_2\sigma[l_1\sigma]_p, p, l_2\sigma, r_2\sigma)$ a *critical peak* and the equation $l_2\sigma[r_1\sigma]_p \approx r_2\sigma$ a *critical pair* of $\mathcal{R}$, obtained from the overlap $\langle l_1 \to r_1, p, l_2 \to r_2\rangle$. The set of all critical pairs of $\mathcal{R}$ is denoted by $\mathsf{CP}(\mathcal{R})$.
To make it easier to read, a critical peak (t, p, s, u) is usually denoted by $t \xleftarrow{p} s \xrightarrow{\epsilon} u$.

3.2 The Completion Procedure

Figure 3.1 depicts the completion procedure as a set of inference rules. Note that the special symbol $\dot{\approx}$, which is used for the definitions of *Orient* and *Simplify* signifies that these rules can be applied in both directions. We write $\mathcal{E},\mathcal{R} \vdash \mathcal{E}',\mathcal{R}'$ if the pair $\mathcal{E}',\mathcal{R}'$ can be obtained from $\mathcal{E},\mathcal{R}$ by applying one of the inference rules of Figure 3.1.

Deduce infers new equations from critical pairs. *Orient* creates rewrite rules from equations by orienting them according to the provided reduction order. *Delete* removes trivial equations from $\mathcal{E}$. When the right-hand side of a rule in $\mathcal{R}$ is further reducible with our current set of rules, we can use *Compose* to rewrite it to its simpler form. For reducible left-hand sides of rules we use *Collapse*. It removes the rule from $\mathcal{R}$, rewrites its left-hand side, and adds the resulting simpler equation to $\mathcal{E}$. When the right- or left-hand

$$\text{Deduce}\quad \frac{\mathcal{E},\mathcal{R}}{\mathcal{E}\cup\{s\approx t\},\mathcal{R}}\ \text{if } s\ {}_{\mathcal{R}}\!\leftarrow\cdot\rightarrow_{\mathcal{R}} t \qquad \text{Compose}\quad \frac{\mathcal{E},\mathcal{R}\uplus\{s\to t\}}{\mathcal{E},\mathcal{R}\cup\{s\to u\}}\ \text{if } t\to_{\mathcal{R}} u$$

$$\text{Orient}\quad \frac{\mathcal{E}\uplus\{s\mathrel{\dot\approx} t\},\mathcal{R}}{\mathcal{E},\mathcal{R}\cup\{s\to t\}}\ \text{if } s>t \qquad \text{Simplify}\quad \frac{\mathcal{E}\uplus\{s\mathrel{\dot\approx} t\},\mathcal{R}}{\mathcal{E}\cup\{u\approx t\},\mathcal{R}}\ \text{if } s\to_{\mathcal{R}} u$$

$$\text{Delete}\quad \frac{\mathcal{E}\uplus\{s\approx s\},\mathcal{R}}{\mathcal{E},\mathcal{R}} \qquad \text{Collapse}\quad \frac{\mathcal{E},\mathcal{R}\uplus\{t\to s\}}{\mathcal{E}\cup\{u\approx s\},\mathcal{R}}\ \text{if } t\to_{\mathcal{R}} u$$

Figure 3.1: Inference system for Knuth-Bendix completion.

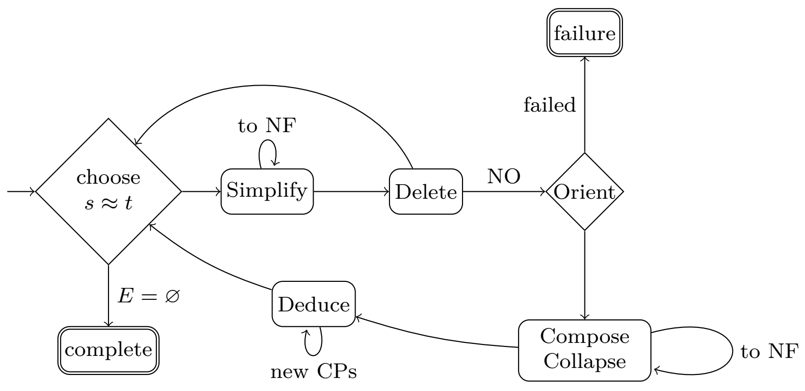


Figure 3.2: Flow chart of automatic completion as implemented in KBCV.

side of an equation is reducible, we can use *Simplify* to apply the reductions. Then the old equation is replaced by its simplified version.

Different strategies, specifying which rule to apply in each situation, can be used to obtain a complete rewrite system with these inference rules.

Definition 3.3. A *run* for a given ES $\mathcal{E}$ is a finite sequence $\mathcal{E}_0,\mathcal{R}_0 \vdash \mathcal{E}_1,\mathcal{R}_1 \cdots \vdash \mathcal{E}_n,\mathcal{R}_n$ such that $\mathcal{E}_0 = \mathcal{E}$ and $\mathcal{R}_0 = \varnothing$. The run *fails* if $\mathcal{E}_n \neq \varnothing$. The run is *fair* if $\mathsf{CP}(\mathcal{R}_n) \subseteq \bigcup_{i\geq 0} \mathsf{CP}(\mathcal{R}_i)$.

Fairness means that all possible critical peaks were considered during the run. A run may fail when some equations in $\mathcal{E}_i$ can't be oriented with respect to the given reduction order. The following theorem states that no matter what strategy is applied, we will always reach a complete rewrite system at the end of a fair and non-failing run.

Theorem 3.4. *For every fair non-failing run $\mathcal{E}_0,\mathcal{R}_0 \vdash \mathcal{E}_1,\mathcal{R}_1 \cdots \vdash \mathcal{E}_n,\mathcal{R}_n$ the TRS $\mathcal{R}_n$ is a complete representation of $\mathcal{E}$.*

Proof. See [20]. □

In the Android version of KBCV the user is not restricted to a specific strategy. Every inference rule can be applied at any time and the app will only announce that the rewrite

$$
\begin{array}{lll}
\text{Deduce} & \dfrac{\mathcal{E}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}} & \text{if } s \;{}_{\mathcal{R}}\!\leftarrow \cdot \rightarrow_{\mathcal{R}} t \\[2ex]
\text{Compose} & \dfrac{\mathcal{E}, \mathcal{R} \uplus \{s \rightarrow t\}, \mathcal{C}}{\mathcal{E}, \mathcal{R} \cup \{s \rightarrow u\}, \mathcal{C}} & \text{if } t \rightarrow_{\mathcal{R}} u \\[2ex]
\text{Orient} & \dfrac{\mathcal{E} \uplus \{s \dot{\approx} t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E}, \mathcal{R} \cup \{s \rightarrow t\}, \mathcal{C} \cup \{s \rightarrow t\}} & \text{if } \mathcal{C} \cup \{s \rightarrow t\} \text{ terminates} \\[2ex]
\text{Simplify} & \dfrac{\mathcal{E} \uplus \{s \dot{\approx} t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{u \approx t\}, \mathcal{R}, \mathcal{C}} & \text{if } s \rightarrow_{\mathcal{R}} u \\[2ex]
\text{Delete} & \dfrac{\mathcal{E} \uplus \{s \approx s\}, \mathcal{R}, \mathcal{C}}{\mathcal{E}, \mathcal{R}, \mathcal{C}} & \\[2ex]
\text{Collapse} & \dfrac{\mathcal{E}, \mathcal{R} \uplus \{t \rightarrow s\}, \mathcal{C}}{\mathcal{E} \cup \{u \approx s\}, \mathcal{R}, \mathcal{C}} & \text{if } t \rightarrow_{\mathcal{R}} u
\end{array}
$$

Figure 3.3: Completion without a fixed reduction order.

system is complete when all possible critical peaks are joinable, and $\mathcal{E}$ is empty. However for automatic completion we use the strategy depicted in the flow chart in Figure 3.2.

Since it is often hard to say which reduction order will work in advance, KBCV uses an adapted completion algorithm which allows us to find and change a suitable reduction order during the run. For that an additional *constraint* TRS $\mathcal{C}$ is used. Before orienting a new rule $s \rightarrow t$ we use the extended constraint system $\mathcal{C} \cup \{s \rightarrow t\}$ to check for termination. This method was introduced in [26]. The adapted inference system is depicted in Figure 3.3.

We will now demonstrate how the completion procedure works in practice, by constructing a complete TRS for group theory.

Example 3.5. Group theory consists of the following three equations[1]:

$$\mathsf{e} \cdot x \approx x \qquad x^- \cdot x \approx \mathsf{e} \qquad (x \cdot y) \cdot z \approx x \cdot (y \cdot z)$$

We start off by orienting all three equations into rewrite rules, by applying the inference rule *Orient*.

$$\mathsf{e} \cdot x \rightarrow x \tag{1}$$

$$x^- \cdot x \rightarrow \mathsf{e} \tag{2}$$

$$(x \cdot y) \cdot z \rightarrow x \cdot (y \cdot z) \tag{3}$$

Rules (1) and (2) can only be oriented this way because of Definition 2.15. For rule (3) the other direction is also possible. To ensure termination with LPO we add $\cdot > \mathsf{e}$ to the precedence.

[1] For better readability we use infix notation for the function symbol $\cdot$ and postfix notation for the function symbol $^-$: $x \cdot y$ in infix corresponds to $\cdot(x, y)$ and x^- in postfix corresponds to $^-(x)$.

In the next step we check for critical peaks, whose critical pairs should be added as equations to $\mathcal{E}$ by *Deduce*. To make it easier for the reader to see how each critical peak is constructed, we mark the redex that is contracted with the left arrow by a line below that term, and the redex that is contracted on the right side by a line above that term. There are currently three critical peaks to be found:

$$\begin{tikzcd}
& \overline{(\underline{x^- \cdot x}) \cdot z} \arrow[dl, "(2)"'] \arrow[dr, "(3)"] & \\
\mathsf{e} \cdot z & & x^- \cdot (x \cdot z)
\end{tikzcd}
\quad
\begin{tikzcd}
& \overline{(\underline{(x \cdot y) \cdot x_1}) \cdot z} \arrow[dl, "(3)"'] \arrow[dr, "(3)"] & \\
(x \cdot (y \cdot x_1)) \cdot z & & (x \cdot y) \cdot (x_1 \cdot z)
\end{tikzcd}
\quad
\begin{tikzcd}
& \overline{(\underline{\mathsf{e} \cdot x}) \cdot z} \arrow[dl, "(1)"'] \arrow[dr, "(3)"] & \\
x \cdot z & & \mathsf{e} \cdot (x \cdot z)
\end{tikzcd}$$

In the second and third of our critical peaks, both sides can be rewritten to the same normal form, so we can safely ignore them. In terms of the inference system, we would first apply *Simplify* and then *Delete* to the equations added by *Deduce*. The left-hand side of the first peak can also be simplified $\mathsf{e} \cdot z \to z$ with rule (1). So we first deduce equation $\mathsf{e} \cdot z \approx x^- \cdot (x \cdot z)$ and then apply *Simplify* to it. Now $\mathcal{E}$ contains the single equation $z \approx x^- \cdot (x \cdot z)$ while $\mathcal{R}$ consists of the rules (1), (2) and (3). Then we orient the equation above from right to left:

$$x^- \cdot (x \cdot z) \to z \tag{4}$$

By adding this equation to $\mathcal{R}$ we introduce several new critical peaks. We will not discuss every single one of them, since most of them can be simplified and deleted in later steps. So we will focus on those critical peaks that we need to orient into rules to make the TRS ultimately confluent. At this stage there are two interesting critical peaks:

$$\begin{tikzcd}
& \overline{\mathsf{e}^- \cdot (\underline{\mathsf{e} \cdot x})} \arrow[dl, "(1)"'] \arrow[dr, "(4)"] & \\
\mathsf{e}^- \cdot x & & x
\end{tikzcd}
\qquad
\begin{tikzcd}
& \overline{x^{--} \cdot (\underline{x^- \cdot x})} \arrow[dl, "(2)"'] \arrow[dr, "(4)"] & \\
x^{--} \cdot \mathsf{e} & & x
\end{tikzcd}$$

We add the appropriate equations to $\mathcal{E}$ with *Deduce* and then use *Orient* to obtain the two rules:

$$\mathsf{e}^- \cdot x \to x \tag{5}$$

$$x^{--} \cdot \mathsf{e} \to x \tag{6}$$

Using *Deduce* again we look for critical peaks and find one between rules (3) and (6):

$$\begin{tikzcd}
& \overline{(\underline{x^{--} \cdot \mathsf{e}}) \cdot z} \arrow[dl, "(6)"'] \arrow[dr, "(3)"] & \\
x \cdot z & & x^{--} \cdot (\mathsf{e} \cdot z)
\end{tikzcd}$$

Simplifying the right-hand side $x^{--} \cdot (\mathsf{e} \cdot z) \to x^{--} \cdot z$ with rule (1) and then orienting it from right to left, results in the new rule:

$$x^{--} \cdot z \to x \cdot z \tag{7}$$

By adding this rule we got a new critical peak:

$$\begin{tikzcd} & \overline{\underline{x^{--} \cdot \mathsf{e}}} \arrow[dl, "(7)"'] \arrow[dr, "(6)"] & \\ x \cdot \mathsf{e} & & x \end{tikzcd}$$

Orienting from left to right yields the new rule:

$$x \cdot \mathsf{e} \to x \tag{8}$$

Now consider these new critical peaks:

$$\begin{tikzcd} & \overline{\underline{\mathsf{e}^{-} \cdot \mathsf{e}}} \arrow[dl, "(5)"'] \arrow[dr, "(8)"] & \\ \mathsf{e} & & \mathsf{e}^{-} \end{tikzcd} \qquad \begin{tikzcd} & x^{--} \cdot \mathsf{e} \arrow[dl, "(6)"'] \arrow[dr, "(8)"] & \\ x & & x^{--} \end{tikzcd}$$

We orient the first critical pair from right to left and the second one from left to right.

$$\mathsf{e}^{-} \to \mathsf{e} \tag{9}$$

$$x^{--} \to x \tag{10}$$

With these rules we can reduce some left-hand sides of existing rules. The inference rule for this process is called *Collapse*. In our example we can collapse rules (5), (6) and (7). We remove the rules from $\mathcal{R}$, rewrite their left-hand sides as far as possible and add the resulting equations to $\mathcal{E}$.

$$\mathsf{e}^{-} \cdot x \xrightarrow{(10)} \mathsf{e} \cdot x \xrightarrow{(1)} x$$

$$x^{--} \cdot \mathsf{e} \xrightarrow{(9)} x \cdot \mathsf{e} \xrightarrow{(8)} x$$

$$x^{--} \cdot z \xrightarrow{(9)} x \cdot z$$

The resulting equations $x \approx x$, $x \approx x$ and $x \cdot z \approx x \cdot z$ can immediately be deleted since they have equal left- and right-hand sides.

Now we continue our search for critical peaks and find one between rules (10) and (2).

$$\begin{tikzcd} & \overline{\underline{x^{--}} \cdot x^{-}} \arrow[dl, "(10)"'] \arrow[dr, "(2)"] & \\ x \cdot x^{-} & & \mathsf{e} \end{tikzcd}$$

We orient this critical pair from left to right and get the new rule:

$$x \cdot x^{-} \to \mathsf{e} \tag{11}$$

Doing this gave rise to two new critical peaks:

$$\begin{tikzcd} & \overline{(\underline{x \cdot x^-}) \cdot y} \arrow[dl, "(11)"'] \arrow[dr, "(3)"] & \\ \mathsf{e} \cdot y & & x \cdot (x^- \cdot y) \end{tikzcd} \qquad \begin{tikzcd} & \overline{\underline{(x \cdot y) \cdot (x \cdot y)^-}} \arrow[dl, "(11)"'] \arrow[dr, "(3)"] & \\ \mathsf{e} & & x \cdot (y \cdot (x \cdot y)^-) \end{tikzcd}$$

We can rewrite the left-hand side of the first critical pair further: $\mathsf{e} \cdot y \to_{(1)} y$. Then we orient it from right to left:

$$x \cdot (x^- \cdot y) \to y \tag{12}$$

We also orient the second critical pair from right to left:

$$x \cdot (y \cdot (x \cdot y)^-) \to \mathsf{e} \tag{13}$$

Now we are almost done. Only two more critical peaks to consider. The first one is between rules (4) and (13):

$$\begin{tikzcd} & \overline{x^- \cdot (\underline{x \cdot (y \cdot (x \cdot y)^-)})} \arrow[dl, "(13)"'] \arrow[dr, "(4)"] & \\ x^- \cdot \mathsf{e} & & y \cdot (x \cdot y)^- \end{tikzcd}$$

We first rewrite the left-hand side: $x^- \cdot \mathsf{e} \to_{(8)} x^-$, then we orient from right to left:

$$y \cdot (x \cdot y)^- \to x^- \tag{14}$$

Since we can now rewrite the right-hand side of rule (13) to match its left-hand side, we have made it superfluous:

$$x \cdot (y \cdot (x \cdot y)^-) \to x \cdot x^- \to \mathsf{e}$$

We have one last critical pair to consider:

$$\begin{tikzcd} & \overline{y^- \cdot (\underline{y \cdot (x \cdot y)^-})} \arrow[dl, "(14)"'] \arrow[dr, "(4)"] & \\ y^- \cdot x^- & & (x \cdot y)^- \end{tikzcd}$$

Orienting from right to left, we get the last rule of our rewrite system:

$$(x \cdot y)^- \to y^- \cdot x^- \tag{15}$$

Finally, the TRS $\mathcal{R}$ consisting of the following eleven rules is confluent:

$$\begin{aligned}
\mathsf{e} \cdot x &\to x \\
x^- \cdot x &\to \mathsf{e} \\
(x \cdot y) \cdot z &\to x \cdot (y \cdot z) \\
x^- \cdot (x \cdot z) &\to z \\
x \cdot \mathsf{e} &\to x \\
x^{--} &\to x \\
\mathsf{e}^- &\to \mathsf{e} \\
x \cdot x^- &\to \mathsf{e} \\
x \cdot (x^- \cdot y) &\to y \\
y \cdot (x \cdot y)^- &\to x^- \\
(x \cdot y)^- &\to y^- \cdot x^-
\end{aligned}$$

This can be verified by computing all critical pairs between these rules and checking that all of them are joinable.

So far we have discussed the theory of term rewriting and completion, together with some explanatory examples. In the following chapters we will move our focus to more Android related topics, starting with a description of Scala-on-Android development.

4 Scala for Android

In principle any language that uses the Java Virtual Machine (JVM) as a runtime environment can also run on the Android runtime environment. Some examples of languages that are actively being used for the development of Android applications (besides Java) are Scala, Clojure and Kotlin. But still the Java programming language is by far the most popular choice. In Section 4.1 we will describe the build process for Android applications to better understand the steps that are necessary to run Scala code on Android. Then, in Section 4.2, we want to highlight the reasons for choosing Scala as the main language for this project. Section 4.3 illustrates how to set up an Android project that contains Scala code with SBT as build tool. Finally, in Section 4.4 we present some of the challenges that may arise when using Scala-on-Android and give pointers on how to negotiate them.

4.1 The Android Build Process

Android applications are shipped as APK files (short for *Android Application Package*) to the user. The compiled source code and other resources of an application are packed into a single archive with the file extension `.apk`.

Figure 4.1 shows a diagram of the build process for Android applications. During the first step all XML resource files are compiled with *aapt* (short for *Android Asset Packaging Tool*). This generates the file `R.java` which contains a representation of all resource identifiers in such a way, that the resources (views in a layout file, colors, strings, . . .) can be referenced from code. The next step is to compile all source code to Java bytecode, resulting in several `.class` files. When developing in Java this step is performed by the *javac* compiler, for Scala the *scalac* compiler is used. Next, all the `.class` files are converted into the single file `classes.dex` by the *dex* tool. This file is a *Dalvik Executable* file and contains bytecode that can run on Android devices. Then the `classes.dex` together with all resources is packaged into an APK file by the *Apk Builder*. The last two steps are signing the app with a debug or release key and then using *zipalign* to perform some memory optimization [15].

4.2 Why Scala

In this section we will illustrate the reasons why we chose Scala as the main language for development, even though Android development is usually done with Java.

A big factor, and the reason why we considered Scala in the first place, was that all code for the original KBCV tool was written in Scala [25]. To use the existing code for the

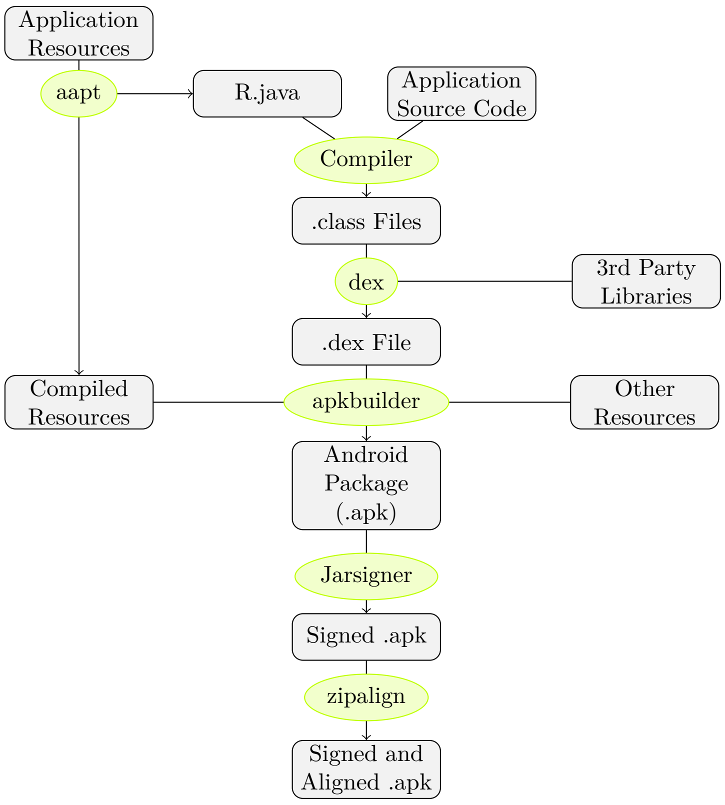


Figure 4.1: The Android Build Process.

Android project, we could either export it as a JAR library, or integrate the source code into the project as a separate module. The latter approach has the benefit of directly being able to adapt parts of the code to our specific needs. But when one module of your Android project contains Scala code, you already have to go through the whole Scala-on-Android build process. So it seemed obvious to consider writing the entire project code in Scala instead of Java.

Following are some additional benefits of Scala over Java [22].

- **Functional Programming**
  Scala combines object-oriented and functional programming concepts in a statically typed language. This means that we can use all the benefits of functional programming (concurrency, lazyness, pattern matching, . . .) without having to give up on an object-oriented design.

- **Syntactic Sugar**
  Semicolons are optional in Scala and there are a lot of instances where the dots for method calls or empty braces for methods without parameters can be omitted. Also class and variable declarations are much shorter in Scala, since repetitive type information can be left out. All of this results in less cluttered and more readable

```
KBCV/
├── arcLayout/
├── project/
│   ├── build.properties
│   └── plugins.sbt
├── src/
│   ├── java/
│   ├── res/
│   ├── scala/
│   └── AndroidManifest.xml
├── termlib/
├── android.sbt
└── build.sbt
```

Figure 4.2: The project structure of KBCV.

code, and the programmer has to type less boilerplate code.

- **Traits**
  Traits in Scala are similar to interfaces in Java. But unlike interfaces, they can also contain concrete method implementations. In addition, a class can mix in any number of different traits and thus gain several different properties.

- **Interoperability**
  Scala and Java classes can be seamlessly mixed together. Scala code can call Java methods, access Java fields, inherit Java classes, and implement Java interfaces. Scala code can also be invoked from Java, although sometimes some of Scala's more advanced features need to be encoded separately for this to work.

In this context it should also be noted that the performance of an Android application does not suffer when its code is written in Scala as compared to the Java version [6].

4.3 Android, Scala and SBT

When setting up an Android project in Scala we first have to decide which build tool to use. There are basically three options: SBT, Gradle or Maven [13].

SBT (short for Scala Build Tool) [16] is the standard build tool for Scala projects and uses Scala-based build definitions. SBT is the tool that is most often used for Scala-on-Android projects and there exist several plugins for that purpose. One of them is the *sbt-android plugin* which we will now discuss in detail.

The sbt-android plugin [21] provides an abstraction of the Android API for Scala. It is a well maintained plugin which is being developed by Perry Nguyen.

When using this plugin all project files have to be organized in a particular structure for it to recognize and compile the files correctly. Figure 4.2 shows this structure for the KBCV project. The topmost folder, `arcLayout`, is an Android library module that implements a radial layout, which is used in KBCV for creating variables and function symbols, as can be seen in Section 6.1. The module contains source code and XML resources to define all necessary functionality. KBCV contains another library module, `termlib`. It contains only regular Scala code, no Android specific definitions and no XML resources. Termlib has implementations of all term rewriting related functions and stems from the original KBCV project by Thomas Sternagel [25]. Below `arcLayout` we have the `project` folder. It contains the two files `build.properties` and `plugins.sbt`. The first is an optional file that specifies the SBT version for this build. The second manages the plugins for SBT. Here we have to add the sbt-android plugin and others that we want to use for the build. It is also possible to put a `Build.scala` file, which manages the whole build, into the `project` folder. In KBCV however we used a `build.sbt` file at the top level of the directory. In `build.sbt` the project with all its library and module dependencies is defined. The file `android.sbt` is used to determine all Android related properties, like minimal SDK version and ProGuard options. The actual source code of the project is located in the `src` folder. All Java code has to go into the `java` folder and Scala code into the `scala` folder. The `res` directory contains subfolders for images, layouts, and other Android resources. Also in the `src` directory we have the `AndroidManifest.xml` which contains configuration details of the application, like package name, activities that make up the application, and permissions that are requested.

The sbt-android plugin provides several Android-specific commands that allow to package APK files and install them to a connected device. Here are some of the available commands. A more complete list can be found on the plugin's github page [21].

- `compile`
  Compiles all Java and Scala sources of the project.

- `android:package`
  Builds an APK for the project.

- `android:install`
  Builds and installs an APK file of the project to a connected device.

- `android:uninstall`
  Removes the application from a connected device.

- `android:run`
  Builds the APK, installs it to a connected device and starts the application on the device.

- `android:debug`
  Builds the APK, installs it to a connected device and waits for the debugger to attach before starting the application.

4.4 Challenges

This section describes some of the obstacles one might encounter when using Scala-on-Android. When possible, solutions to the identified challenges are presented.

Finding documentation and examples Unfortunately there are very few articles and no books about Scala-on-Android. Also questions and answers on Stack Overflow [24] about Scala-on-Android specific problems are very rare. Besides Stack Overflow there is a Google Group, where users can ask questions [7]. A good tutorial to get started with programming in Scala for Android can be found at [12]. This website was developed by Niklas Klein as part of a bachelor thesis at the Freie Universität Berlin [13].

Using an IDE For conventional Android projects, the official IDE *Android Studio* is a good choice [1]. But for Scala-on-Android *IntelliJ IDEA* [10], on which Android Studio is based, is easier to set up, because it has more configuration options. Most importantly one has to add Scala language support via the *Scala* plugin [11] and an interactive SBT console that comes with the *SBT* plugin [17]. Also note that IntelliJ only recognizes Android specific code and references to resources, if the Project SDK is set to an Android SDK.

Multi-module projects In KBCV we rely on two additional modules, *arcLayout* and *termlib*. They both have to be declared as dependencies in the build configuration. It was especially challenging to find the right way to include arcLayout as an Android library. Android libraries differ from conventional libraries in that they contain Android specific resource files. These are for example XML layout files or the `AndroidManifest.xml` for that library. When including arcLayout like any other module in `build.sbt` these will be ignored by the compiler and can not be accessed from within the main module. Key here was to use the command `androidBuildWith`, which makes sure that the library and all its resources can be properly used in the project.

The 64K reference limit DEX files have a limited number of methods that can be referenced within a single file. This is referred to as the *64K reference limit* [5] and it means that a DEX file can only contain 64×1024 references, including Android framework methods, library methods, and methods of the own source code. With Scala-on-Android there are already so many dependencies coming from the Scala library that it is impossible to stay below that specified limit, [13]. To solve this problem we can use *ProGuard* [8], a tool that shrinks, optimizes and obfuscates Java bytecode. The sbt-android plugin invokes ProGuard by default with a set of generic options. Custom settings can be specified in the build file.

In this chapter we covered the technical details of Android development in Scala. Next, we are going to look at some design principles for Android applications.

5 Android User Interface Design

The main goal of this bachelor project was to create a user interface (UI) that is as easy and intuitive to use as possible. To achieve this we kept as close to the official Android design guidelines as possible and used widely recognized design patterns. In this chapter we introduce the current version of Android design guidelines, called *Material Design*. Also several of the patterns that were incorporated into the KBCV on Android application are described in more detail.

Material Design is a set of design principles, that was first introduced with the launch of Android 5.0 in 2014. The goal of Material Design is to provide guidelines that are applied across devices and software versions such that user can easily recognize patterns and start using applications without needing an explanation of the interface [4].

The Material Design guidelines are being published at [19]. The following descriptions of some of the concepts of Material Design that also appear in KBCV are based on the content of this website.

5.1 App Bar

The App Bar, formerly known as Action Bar, was first introduced in Android 3.0. It sits at the top of the screen right below the status bar (the one which displays notifications and system icons). Its purpose is to provide a well-organized overview of the user's current location in the app and the most important available actions there.

A typical App Bar is displayed in Figure 5.1. The leftmost icon is the so called *hamburger icon*. It indicates that a navigation drawer (see Section 5.3) is part of the current screen. When the users moves deeper down the screen hierarchy the hamburger icon might change into a left arrow, representing an *up* or *back* action. To the right of the hamburger icon we have the title which reflects the current activity. This way a user can always see at one glance what his current location is. Further to the right follow several action icons and/or text, promoting relevant actions. These may change depending on the current activity. Less frequently used actions are hidden within the overflow menu which can be opened by tapping the three vertical dots at the far right end of the App Bar.

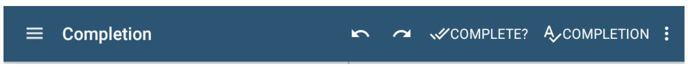


Figure 5.1: The App Bar of the Completion activity in KBCV.

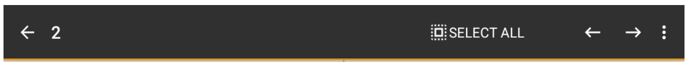


Figure 5.2: The CAB after selecting two equations in KBCV.

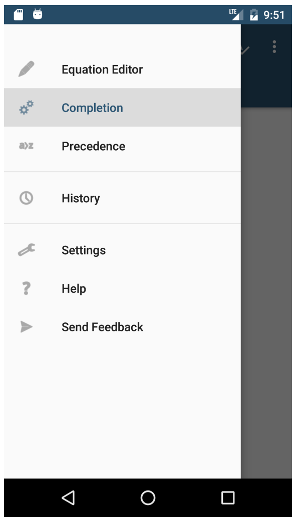


Figure 5.3: The Navigation Drawer in KBCV.

5.2 Contextual Action Bar

The Contextual Action Bar (CAB) is a pattern, that is often used when implementing selection of items. The CAB is a temporary toolbar that overlaps the App Bar while a selection is active. It provides actions specific to the selected content. By tapping the arrow on the left the selection can be canceled. The number right next to it represents the number of items that are currently selected. Like with the App Bar, when space is narrow less frequently used actions move into the overflow menu. KBCV on Android uses CABs in several of its activities to handle the selection of equations and rules. Figure 5.2 shows the CAB that appears in the Completion screen when selecting two equations.

5.3 Navigation Drawer

The Navigation Drawer is a panel that slides in from the left edge of the screen. It displays the app's main navigation options and allows easy switching between them. A Navigation Drawer is particularly useful when the app contains deep navigation branches or has a lot of top-level views.

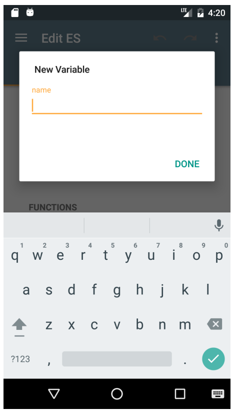
The Navigation Drawer can be revealed by either pressing the hamburger icon or sliding it in from the outer left edge of the screen. To dismiss the Navigation Drawer a user can either swipe it back to the left edge, touch the area outside the Navigation

Drawer, or press the back button. As can be seen in Figure 5.3 the Navigation Drawer spans the whole height of the screen, overlapping any other content, including the App Bar. It is not quite as wide as the screen and the content that is still visible behind the drawer is darkened.

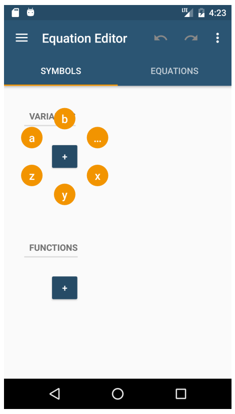
Figure 5.3 displays the six top-level views of KBCV: Equation Editor, Completion, Precedence, History, Settings, and Help. The seventh entry, Send Feedback, takes the user outside of KBCV to an email app. It is recommended to group similar views together and use dividers to separate them. In KBCV there are three groups. The main activities Equation Editor, Completion, and Precedence form one group. Settings, together with the support features Help and Send Feedback forms another. The History is located in a separate area because its functionality differs from that of the other two groups. The Navigation Drawer also indicates the view where the user is currently located. In Figure 5.3 this is the Completion section.

5.4 Radial Menu

In KBCV we looked for a method to efficiently and intuitively enter new function and variable names. The original KBCV tool uses a dialog window in which the variable symbols and equations are to be entered. In KBCV on Android we first implemented a similar design, where the user needed to enter symbols into a dialog by using the on-screen keyboard. This approach however was not satisfying for touchscreens. Inspired by the clock-like circular menu that is used in Android for setting times (e.g. for an alarm, or in the calendar app), we decided to do something similar in KBCV. Unfortunately there is no official library that makes this pattern available for your own menus. We used a library called *ArcLayout*, which we found on The Android Arsenal [23], as the basis for the radial menus implemented in KBCV.



(a) Adding new variables with a dialog and the on-screen keyboard.



(b) Adding new variables with the radial menu.

Figure 5.4: Two different ways of adding new variables.

Figure 5.4 shows a comparison of the old dialog to enter symbol names versus the new method with radial menus. Assuming that most users will be satisfied with the offered selection of symbols, this input method greatly increases the speed and ease with which variables and functions can be added to the system. A thorough description of the functionality of our radial menus can be found in the next chapter in Section 6.1.1.

5.5 Responsive Design

The goal of responsive design is to adapt a layout to different screen sizes. Since more space is available on larger devices, more information and more details can be shown than on smaller devices. Android offers a simple way to provide different layouts for different screen sizes with its resource system. There the developer can put different layouts into the appropriate resource folders and the operating system will inflate the right ones at runtime. Resource folders can depend on width, height, density or other qualifiers. Most of the time width is the factor on which layout changes are based, since more width means more content can be shown side by side. Height is less important because many layouts incorporate some kind of scrollview and those inherently adapt very well to height changes. The Material Design guidelines provide specific width breakpoints at which layout changes should be triggered. These points are 480, 600, 840, 960, 1280, 1440, and 1600dp. The Material Design guidelines also suggest several patterns that can be used to make an app responsive.

Divide When additional space becomes available on larger screens it may be used to show multiple views of the UI at the same time. In KBCV this pattern was implemented for the two activities Equation Editor and Completion. They appear as tabbed layouts on small screens, but show the content of the tabs side by side on larger screens.

Reflow UI elements may be arranged differently on different screen sizes. An example of that pattern in KBCV is the *FlowLayout* which is used to display variable and function symbols. Symbols are basically arranged horizontally, but when the screen is not wide enough, they can flow over into the next line.

Expand UI elements may be expanded and their margins increased when more space becomes available. In KBCV this is what happens to the scrollviews that contain the lists of current equations and rules. They always stretch out to use all the available space.

Position The positions of UI elements may change depending on the screen size. For example actions in the App Bar can move into the overflow menu on narrow screens, which is also something we implemented in KBCV.

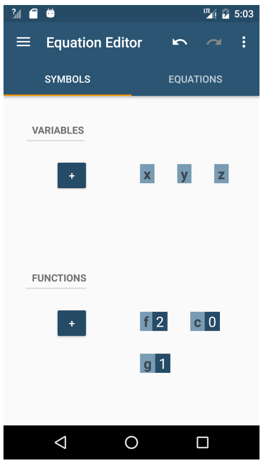
This chapter examined the basic principles of design that we followed during the development of KBCV on Android. What comes now is a detailed description of the app's actual user interface and functionality.

6 The App

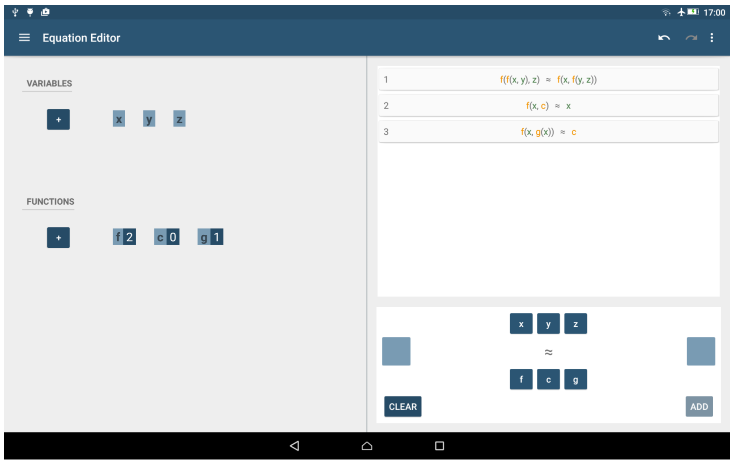
An APK of the KBCV on Android app is available for download on the KBCV website at `http://cl-informatik.uibk.ac.at/software/kbcv`. The app is divided into several areas, which are accessible through the Navigation Drawer. The two main areas are the Equation Editor and the Completion screen. In addition there is a screen for viewing and editing the LPO precedence, a settings screen, a history screen and a help section. In this chapter we will look at each of those areas in turn (except for the help section which is self-explanatory), starting with the Equation Editor in Section 6.1 and the Completion screen in Section 6.2. In Section 6.3 we will describe the functionality of the Precedence screen. Then, in Section 6.4, the undo/redo options that are available in every of the main screens are explained and also how to use the history that is available in the navigation drawer. Finally, in Section 6.5 we will specify which settings are available to the user and what effect they have on the behavior of the App.

6.1 Equation Editor

Depending on the screen width the Equation Editor appears as a tabbed view (< 480dp) or two panels side by side (≥ 480dp). A comparison of the two layouts can be seen in Figure 6.1. In the first tab or panel the user can define the variable and function symbols he wants to use for his equational system. In the second tab or panel he can create the equations.



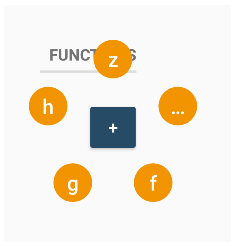
(a) width $<$ 480dp



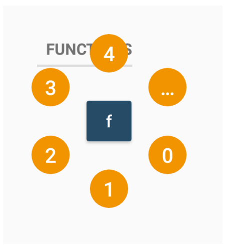
1 f(f(x, y), z) ≈ f(x, f(y, z))
2 f(x, c) ≈ x
3 f(x, g(x)) ≈ c

(b) width $\geq$ 480dp

Figure 6.1: The Equation Editor on different screen sizes.



(a) The default radial menu for function symbols.



(b) The radial menu for arities as it appears after choosing a function symbol (here f).

Figure 6.2: The two radial menus that are used to add a new function symbol.

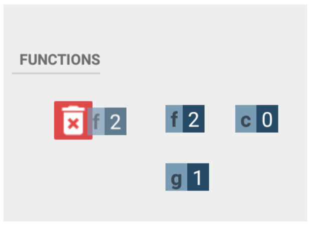


Figure 6.3: Deleting existing symbols.

Alternatively an equational system can be imported from a file. Possible file formats are `.xml` [28] and `.trs` [18]. Also after creating his own equations, a user can save them as a TRS file. All the import and export options are available in the overflow menu of the action bar (the three vertical dots at the far right end).

6.1.1 Defining Symbols

The symbols view, which is located either in the first tab or left panel, is divided into two areas, one for variables and one for function symbols. They both contain a '+' button on the far left side and a flowlayout for the already defined symbols to the right.

Pressing the '+' buttons in the respective views opens a radial menu with a selection of common variable or function symbols. A symbol can be added by simply tapping it. For function symbols it is also required to select an arity. For this purpose, after tapping a function symbol, another radial menu opens which allows the user to select the corresponding arity. Figure 6.2 shows how the radial menus for functions and arities look like. If the user wants to define a symbol that is not contained in the radial menu he can tap the '...' button. This opens a dialog where any character string can be entered to use as a new symbol. Also, if the user needs an arity larger than four he can also open a dialog via the '...' button and enter any positive integer as arity.

Existing symbols can be deleted by longpressing them until a trash icon appears in place of the '+' button and then dragging and dropping the symbol onto the trash. Figure 6.3 depicts this process.

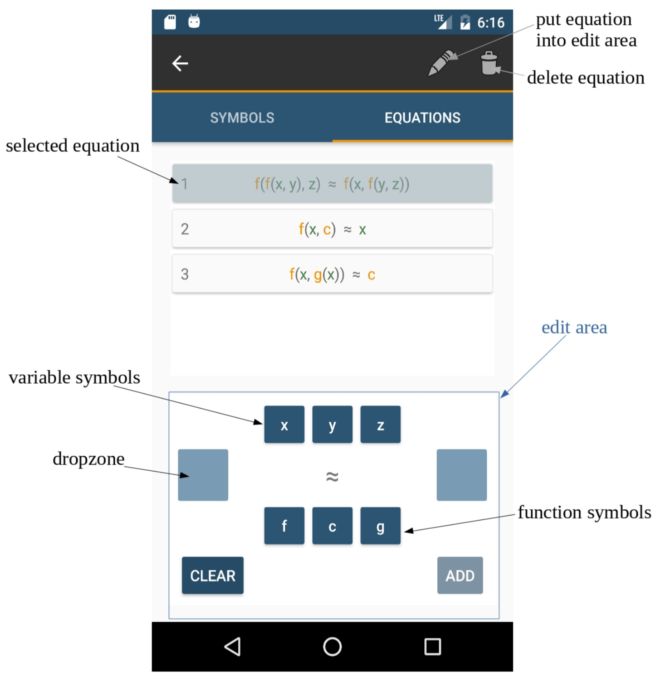


Figure 6.4: Editing equations in KBCV.

Which symbols should appear in the radial menu can be customized in the settings (refer to Section 6.5).

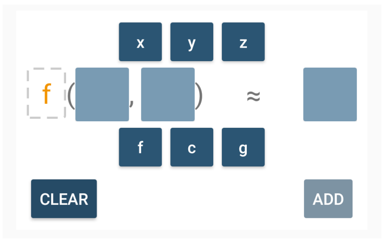
6.1.2 Creating Equations

The equations view is also divided into two parts as shown in Figure 6.4. At the bottom of the equations view the edit area is located. Above it, a list of all current equations is displayed. The edit area holds a list of all previously defined variables and function symbols as draggable buttons. By dragging and dropping a symbol into the designated dropzone that symbol will be added to the equation. For functions with an arity greater than zero, the appropriate number of dropzones will be created. Figure 6.5(a) shows the edit area after dropping function symbol f with arity two into the left dropzone.

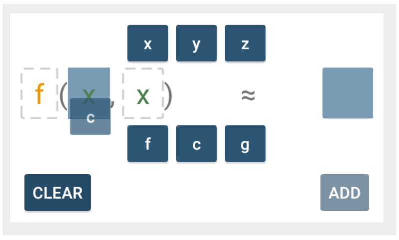
Symbols can also be dragged onto other symbols and thereby overwrite that part of the equation. In Figure 6.5(b) dropping symbol c onto the variable x results in the new left-hand side $\mathsf{f}(\mathsf{c}, x)$.

When an equation is finished, i.e., containing no more dropzones, the user can either press the ‘ADD‘ button or drop the equation into the list of existing equations by dragging it up at the ‘$\approx$‘ sign.

Existing equations can be edited or deleted by selecting them in the list and then



(a) Adding a function symbol.



(b) Overwriting symbols with other symbols.

Figure 6.5: Dropping symbols into the edit area.

choosing the desired action in the action bar. When choosing *edit* the equation will appear in the edit area at the bottom of the screen and the user can start dragging different symbols onto it.

6.2 Completion

The Completion screen is structured similarly to the Equation Editor, in that it appears as a tabbed view on smaller screens and as two panels on larger screens. The first tab or panel shows a list of all equations of the current equational system, the second one shows all rules of the current rewrite system. Figure 6.6 shows the Completion Screen on a tablet.

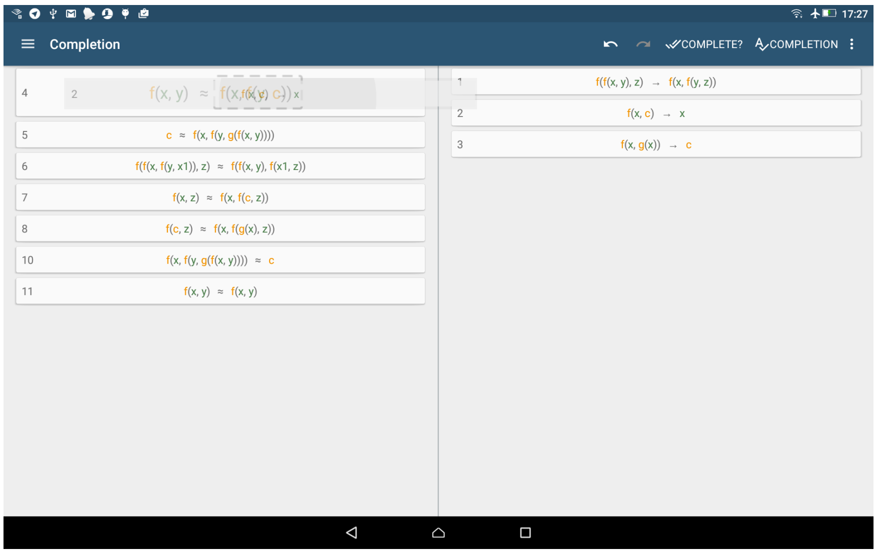
In both views any number of items (equations or rules) can be selected and one of the inference rules of Knuth-Bendix completion can be applied to them. If the user selected equations, the contextual action bar shows the actions *Orient*, *Simplify* and *Delete*. When selecting rules the available actions are *Compose*, *Collapse* and *Deduce*.

The current term rewrite system can always be exported as a TRS file to local storage. This feature is accessible through the overflow menu of the action bar.

6.2.1 Gestures

There are several built-in gestures in KBCV for Android that perform certain parts of Knuth-Bendix completion.

- **Orient:** Equations can be oriented into rules by swiping them to the side. A leftswipe means orient from right to left and a rightswipe means orient from left to right, accordingly.
- **Simplify and Delete:** To simplify an equation as far as possible, the equation has to be longpressed until it is draggable, dragged some distance down, and then released. This will also delete the equation if it proves to be trivial after performing the simplification.
- **Deduce:** Longpressing a rule and then dragging it onto the arrow of another rule makes the app look for critical peaks between those two rules. Resulting critical



4 2 f(x, y) ≈ f(x, f(y, c)) f(x, c) → x

5 c ≈ f(x, f(y, g(f(x, y))))

6 f(f(x, f(y, x1)), z) ≈ f(f(x, y), f(x1, z))

7 f(x, z) ≈ f(x, f(c, z))

8 f(c, z) ≈ f(x, f(g(x), z))

10 f(x, f(y, g(f(x, y)))) ≈ c

11 f(x, y) ≈ f(x, y)

1 f(f(x, y), z) → f(x, f(y, z))

2 f(x, c) → x

3 f(x, g(x)) → c

Figure 6.6: Dragging a rule onto an equation in the Completion screen.

pairs will be added to the existing equations.

- **Simplify, Compose, Collapse:** When a rule is longpressed it can be dragged on any left- or right-hand side of another equation or rule. The side on which the rule was dragged will then be reduced with that rule if possible. Depending on whether it was an equation or rule, and in the latter case also depending on the side, this results in a *Simplify*, *Compose*, or *Collapse* action. Figure 6.6 shows the process of dragging a rule onto the right side of an equation.

6.2.2 Automatic Completion

The Completion screen also offers the ability to automatically perform completion as described in Section 3.2. This feature is available in the action bar of the Completion screen. Automatic completion will run until it either succeeds and finds a complete rewrite system, fails because it can not orient a specific equation, or until it reaches the maximum number of rounds. The maximum number of rounds is initially set to 100 and can be modified in the settings. If automatic completion was not successful, it will nevertheless display the last intermediate equational system and term rewrite system it has found.

6.2.3 Completeness Check

By default the app automatically checks the rewrite system for completeness after the application of an *Orient* or *Delete* inference step. This behavior can also be modified in the Settings. To perform a completeness check manually one can also use the appropriate button in the action bar.

6.3 Precedence

The KBCV app uses the lexicographic path order (LPO) to determine whether an equation can be oriented into a rule, such that the resulting term rewrite system is still terminating. The user can either provide the precedence for LPO himself or leave it to the app, to figure out the necessary restrictions. The Precedence screen shows the ordering that is used at the moment to ensure the termination of the current term rewrite system. The user can add, edit and delete rules at will as long as termination of the current rewrite system can be proved with this precedence.

Adding an entry to the precedence works similarly to adding equations. At the bottom of the Precedence screen an edit area is located. Inside is a list of draggable buttons of all available function symbols and a stub of a new rule with a dropzone at each side. Function symbols can be dragged onto the dropzones (and onto other function symbols) to create a new rule. The rule can then be added to the precedence by either pressing the add button or dragging the rule up on the '>' sign. To edit or delete an existing rule, the user needs to select it by tapping it and then choosing the appropriate action in the contextual action bar. When selecting *edit*, the rule appears in the edit area and the user can drag different function symbols onto the existing ones to change it. If the user tries to add a restriction that contradicts LPO termination, or if he tries to remove a rule that is needed for termination, the app will show an error message.

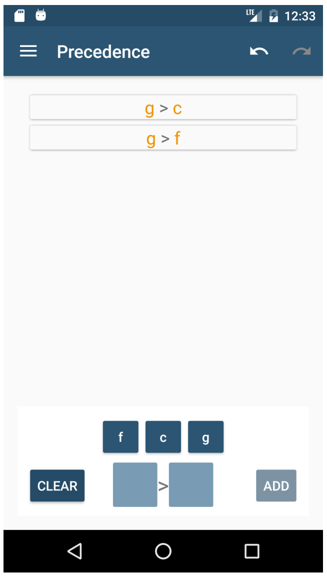
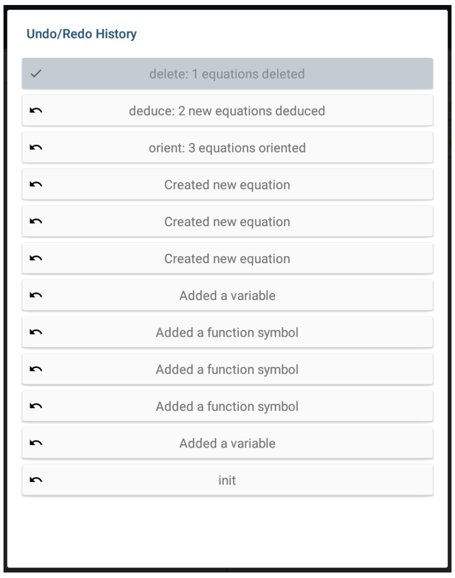


Figure 6.7: The Precedence screen.



Undo/Redo History

- delete: 1 equations deleted
- deduce: 2 new equations deduced
- orient: 3 equations oriented
- Created new equation
- Created new equation
- Created new equation
- Added a variable
- Added a function symbol
- Added a function symbol
- Added a function symbol
- Added a variable
- init

Figure 6.8: The History dialog.

6.4 Undo/Redo and History

The app keeps track of, and records, every intermediate step the user performs in it. The three activities Equation Editor, Completion, and Precedence, all feature *undo* and *redo* buttons in their respective action bars. In addition the entire history is available via the navigation drawer. Figure 6.8 shows an example of how the history can look like. The user can tap any of the history entries to immediately restore that state of the app.

6.5 Settings

There are five individual set-up options accessible via the Settings screen.

- **Automatically Check for Completeness**
  This option enables the user to turn off the automatic completeness check that is by default performed after every action. This may be useful for users who are studying completion and want to perform as much of the completion process manually as possible.
- **Cache Overlaps**
  By default the app will only consider each overlap once. It remembers which overlaps have already been considered and won't add those equations repeatedly. Again it might be useful to turn off this feature for users who are studying completion and want to test their ability of finding critical pairs.
- **Rounds**
  This setting allows the user to specify the maximum number of rounds for automatic completion. An interesting option here is to set the number of rounds to 1. Then one can perform automatic completion round by round and view all intermediate equational systems and term rewrite systems.
- **Variable Symbols**
  Tapping this setting opens a dialog window where the user can specify which variable symbols should be shown in the radial menu of the Equation Editor. Individual symbols have to be entered with a comma as delimiter, spaces being ignored. The maximum number of symbols that can be displayed in the radial menu is six. Everything the user enters after the sixth symbol will be ignored.
- **Function Symbols**
  Here the user can specify which function symbols should appear in the radial menu of the Equation Editor. As with the variable symbols, characters can be entered into the edit dialog with a comma as delimiter and anything that exceeds the limit of six different symbols will be ignored.

In this chapter we gave a thorough description of all of the features implemented in KBCV on Android. The next chapter will conclude this thesis with some final remarks on the project, and also some ideas for possible future work.

7 Conclusion

During the course of this project we developed an Android application that allows its users to perform interactive completion on their tablets and smartphones. We designed the app in such a way that it not only provides the necessary functionality but also has a user interface that is especially adapted to touchscreens and very intuitive and easy to use.

In this thesis we covered the basics of equational systems and term rewriting in Chapter 2 and studied the theory behind the completion procedure in Chapter 3. Then we moved our focus to Android design and development. In Chapter 4 we learned how to make Scala work on the Android platform and what to do about common problems. In Chapter 5 we studied some of Android's design best-practices. And finally, in Chapter 6 we got a good look at the Android application that was being developed as part of this project.

7.1 Future Work

Given that the Android platform is under constant development, there will always be work to do to keep the application up to date. New versions of Android may require adapting the KBCV app to the new environment. Also the design guidelines may be changed significantly in the future. For example when new input methods (speech recognition, eye-gesture recognition, ...) become more popular.

Apart from that, there are several features that could be added to the app, but were out of the scope of this work. The original KBCV tool has an option to import and export command logs. If the KBCV app had the ability to import and export its history in the same format, the two versions would become more compatible and mobile users could exchange and compare their logs with desktop users.

The support for different languages could also be a great upgrade for the app.

Bibliography

[1] Android Studio. `https://developer.android.com/studio/index.html`. Accessed: 2017-03-01.

[2] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge university press, 1998.

[3] L. Bachmair. *Canonical equational proofs.* Birkhauser, 1991.

[4] I. G. Clifton. *Android User Interface Design: Implementing Material Design for Developers.* Addison-Wesley Professional, 2015. ISBN 0-13-419140-4.

[5] Configure apps with over 64K methods. `https://developer.android.com/studio/build/multidex.html`. Accessed: 2017-02-20.

[6] M. Denti and J. K. Nurminen. Performance and energy-efficiency of Scala on mobile devices. In *Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 50–55. IEEE, 2013.

[7] Scala-on-Android google group. `https://groups.google.com/forum/#!forum/scala-on-android`. Accessed: 2017-01-12.

[8] GuardSquare. ProGuard. The open source optimizer for Java bytecode. `https://www.guardsquare.com/en/proguard`. Accessed: 2017-03-01.

[9] IDC. Smartphone OS market share, 2016 Q3. `http://www.idc.com/promo/smartphone-market-share/os`. Accessed: 2017-02-28.

[10] JetBrains. IntelliJ IDEA. `https://www.jetbrains.com/idea/`. Accessed: 2017-03-01.

[11] JetBrains. Scala for IntelliJ IDEA. `https://plugins.jetbrains.com/idea/plugin/1347-scala`. Accessed: 2017-03-01.

[12] N. Klein. Scala on Android the comprehensive documentation. `http://scala-on-android.taig.io/`. Accessed: 2017-03-01.

[13] N. Klein. Scala on Android: problems and solutions. Bachelor thesis, Freie Universität Berlin, 2015.

[14] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[15] S. R. Kotipalli and M. A. Imran. *Hacking Android.* Packt Publishing Ltd, 2016. ISBN 9781785883149.

[16] Lightbend. SBT. The interactive build tool. `http://www.scala-sbt.org/`. Accessed: 2017-03-01.

[17] E. Luontola. SBT for IntelliJ IDEA. `https://plugins.jetbrains.com/idea/plugin/5007-sbt`. Accessed: 2017-01-12.

[18] C. Marché, A. Rubio, and H. Zantema. Termination problem data base: format of input files. `https://www.lri.fr/~marche/tpdb/format.html`. Accessed: 2017-03-01.

[19] Material Design guidelines. `https://material.io/guidelines/`. Accessed: 2017-03-01.

[20] A. Middeldorp. Term rewriting. Lecture Notes, University of Innsbruck, 2016.

[21] P. Nguyen. sbt-android-plugin. `https://github.com/scala-android/sbt-android`. Accessed: 2017-03-01.

[22] M. Odersky, L. Spoon, and B. Venners. Programming in Scala. 2016.

[23] ogaclejapan. ArcLayout. `https://android-arsenal.com/details/1/1689`. Accessed: 2016-10-23.

[24] Scala on android at stack overflow. `http://stackoverflow.com/questions/tagged/scala+android`. Accessed: 2017-02-28.

[25] T. Sternagel and H. Zankl. KBCV–Knuth-Bendix completion visualizer. In *International Joint Conference on Automated Reasoning*, volume 7364 of *LNCS*, pages 530–536. Springer, 2012.

[26] I. Wehrman, A. Stump, and E. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *International Conference on Rewriting Techniques and Applications*, volume 4098 of *LNCS*, pages 287–296. Springer, 2006.

[27] S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara. Optimizing mkbTT. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 373–384, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[28] XTC format specification. `http://www.termination-portal.org/wiki/XTC_Format_Specification`. Accessed: 2017-03-01.