WILHELM BÜCHNER HOCHSCHULE

MASTERTHESIS

Realisierung eines Source-to-Source Compilers zwischen Xamarin.Forms und Flutter zur automatisierten Transformation bestehender mobiler Anwendungen

---

Author:
Julian Pasqué

Betreuer:
Dr. Thomas Kalbe

Masterstudiengang Verteilte und mobile Anwendungen

Fachbereich Informatik

Matrikelnummer: 902953

28. Oktober 2020

Zusammenfassung

Abstract

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Durch die Entwicklung von verschiedenen mobilen Geräten mit unterschiedlichsten Hardwarekomponenten und Betriebssystemen hat sich ein stark fragmentierter Markt ergeben.[1] Diese Marktentwicklung hat einen direkten Einfluss auf die Softwareentwicklung für mobile Endgeräte und damit zu der Entwicklung von Cross-Plattform Frameworks wie Xamarin.Forms und Flutter gesorgt. Durch die Abstraktion von Hardware und Betriebssystem bieten diese Frameworks die Möglichkeit Anwendungen für die verschiedenen Plattformen mit einer gemeinsamen Quelltextbasis zu entwickeln und somit Kosten und Zeit zu sparen.[2] Der Möglichkeit Entwicklungsressourcen zu sparen steht das Risiko der Abhängigkeit entgegen, da sich alle Frameworks zur Cross-Plattform-Programmierung in den verwendeten Programmiersprachen sowie ihrer Arbeitsweise unterscheiden, ist ein Wechsel zwischen den einzelnen alternativen mit enormen Arbeitsaufwenden verbunden.[3]

1.1 Ziel der Arbeit

Im Rahmen dieser Arbeit soll ein Source-To-Source Compiler zwischen den Frameworks Xamarin.Forms und Flutter realisiert werden mit dessen Hilfe die folgende zentrale Forschungsfrage beantwortet werden soll: "Können Apps komplett automatisiert von Xamarin.Forms zu Flutter übersetzt werden, oder sind manuelle Arbeitsschritte erforderlich?"

1.2 Motivation

Im Mai 2020 hat Microsoft mit .NET MAUI einen Nachfolger für das Xamarin.Forms Framework angekündigt, der im Herbst 2021 zusammen mit der sechsten Version des

[1] Vgl. Joorabchi 2016, S. 3.
[2] Vgl. Vollmer 2017, S. 295.
[3] Vgl. Wissel, Liebel und Hans 2017, S. 64.

.NET Frameworks veröffentlicht werden soll. Zum aktuellen Zeitpunkt ist bereits angekündigt, dass .NET MAUI grundlegende Änderung mit sich bringt und Anwendungen die mit Hilfe von Xamarin.Forms entwickelt wurden angepasst werden müssen.[4]

Für Entwickler wird es also unausweichlich sein, grundlegende Änderungen an bereits realisierten Anwendungen vorzunehmen um in der Zukunft von Aktualisierungen zu profitieren. Aus diesem Grund eignet es sich auch alternative Frameworks zu analysieren und zu überprüfen ob ein Grundlegender Umstieg lohnenswert ist. Das Flutter Framework erfreut sich unter Entwicklern einer wachsenden Beliebtheit und ist nicht nur auf Grund seiner performanten Anwendungen mittlerweile weit verbreitet.

Ein automatisierter Umstieg auf das von Google zur Verfügung gestellte Framework würde also nicht nur die Anpassungen an .NET Maui verhindern, sondern auch leistungsfähigere Anwendungen generieren.

1.3 Gliederrung

[4] Vgl. Hunter 2020, Abgerufen am 28.10.2020.

2 Compiler

Programmiersprachen dienen als Verständigungsmittel zwischen Programmierern und Rechenanlagen. Je mehr sich diese Sprachen der Terminologie eines bestimmtes Anwendungsgebietes nähern, desto besser eignen sie sich zur Dokumentation von Algorithmen und Anwendungen. Jedoch entfernen sich diese Sprachen weiter von den Gegebenheiten des realen Rechners. Dieser Effekt muss vor der Ausführung eines Programmes wieder umgekehrt werden, daher das in einer problemorientierten Programmiersprache geschriebene Programm muss in eine maschinenorientierte Form überführt werden.[1] Bereits im Jahre 1951 stelle Rutishauser fest, dass Rechner in der Lage sind diesen Übersetzungsvorgang selbst durchzuführen.[2]
Der Bedarf an dieser automatischen Übersetzung hat sich historisch durch die Verwendung von Hochsprachen ergeben. Da diese statt Anwendungen aus maschinennahen Instruktionen aufzubauen menschenfreundliche Sprachelemente verwenden.[3] Durch diesen Bedarf wurden in 50er Jahren des 20 Jahrhunderts die ersten Compiler entwickelt, welche es erlaubten, dass Programmierer Sprachen verwenden konnten die neben der Problemlösung auch Dokumentationszwecken dienen konnten.[4]
Diese historische Einführung zeigt, dass Compiler schon eine lange Zeit existieren und sich in der Wissenschaft eine einheitliche Definition für diese ergeben hat. So beschreibt Ullman et al. im Jahre 2008 Compiler wie folgt:[5]

Defintion 1: Compiler

Ein Compiler ist ein Programm, welches ein anderes Programm aus einer Quellsprache in ein gleichwertiges Programm einer Zielsprache übersetzen kann.

Neben dem Compiler können noch andere Programme Aufgaben für die Sprachverarbeitung übernehmen, diese sollen an dieser Stelle ebenfalls eingeführt werden:[6]

[1] Vgl. Schneider 1975, S. 15.
[2] **Quelle fehlt! improve https://link.springer.com/article/10.1007/BF02009622**.
[3] Vgl. Wagenknecht 2014, S. 47.
[4] **Quelle fehlt! Meilensteine der Rechentechnik: Zur Geschichte der Mathematik und der Informatik**.
[5] Vgl. Ullman et al. 2008, S. 1.
[6] Vgl. Ullman et al. 2008, S. 4f.

- Präprozessor: Ein Quellprogramm kann aus mehreren Modulen gegSliedert sein, welche in anderen Dateien gespeichert sind. Die Aufgabe des Präprozessor ist es, diese Dateien zusammenzustellen.
- Assembler: Compiler können die Sprache Assembler als Zielsprache haben, da sich solche Programme leichter auf Fehler untersuchen lassen. Der Assembler übernimmt die Verarbeitung und gibt den entsprechenden Maschinencode aus.
- Linker: Da Umfangreiche Programme häufig in mehreren Teilen kompiliert werden müssen die einzelnen Elemente miteinander verknüpft werden, Der Linker übernimmt diese Aufgaben indem er externe Speicheradressen auflöst.

2.1 Aufgabe

Die Aufgaben des Compilers lassen sich, mit dem Ziel der Übersetzung von Programmiersprachen, in die zwei Unteraufgaben Analyse und Synthese unterteilen.[7]
Bei der Analyse wird das Programm in seine Bestandteile zerlegt und mit einer grammatischen Struktur versehen. Diese wird anschließend verwendet um eine Zwischendarstellung des Quellprogramms zu erstellen. Dabei wird überprüft, ob das Programm syntaktisch oder semantisch nicht wohlgeformt ist, und ob der Programmierer Änderungen vornehmen muss. Außerdem werden bei der Analyse Informationen über das Quellprogramm gesammelt und in einer so genannten Symboltabelle abgelegt.[8]
Bei der Synthese wird aus der Zwischendarstellung un den Informationen aus der Symboltabelle das gewünschte Zielprogramm konstruiert. Der Teil des Compilers, der sich mit der Analyse befasst wird oft als Front-End bezeichnet, derjenige der für die Synthese zustädnig ist als Back-End.[9]

2.2 Phasen

Der Vorgang des Kompilieren lässt sich nach Ullman et al. in mehere Phasen unterteilen, die an dieser Stelle eingeführt werden sollen.[10]

[7] Vgl. Ullman et al. 2008, S. 6.
[8] Vgl. Ullman et al. 2008, S. 6f.
[9] Vgl. Ullman et al. 2008, S. 7.
[10] Vgl. Ullman et al. 2008, S. 6.

2.2.1 Lexikalische Analyse

Die erste Phase eines Compilers ist die sogenannte lexikalische Analyse. Dabei wird der Zeichenstream, der das Quellprogramm bildet in Lexeme gegliedert. Für jedes erzeugte Lexem gibt der lexikalische analysator ein Token in folgender Form aus:[11]

<Name, Attributwert>

Der Name ist dabei ein abstaktes Symbol, das während der nächste Phase, der Syntaxanalyse verwendet wird. Der Attributwert auf einen Eintrag in der symboltabelle für dieses Token zeigt. Diese Informationen werden in den späteren Phasen für die semantische Analyse und die Codegeneriung benötigt.[12]

2.2.2 Syntaxanalyse

Die zweite Phase des Compilers ist die Syntaxanalye. Dafür verwendet der sogenannte Parser die vom lexikalischen Analysator ausgegebenen Token um eine baumartige Zwischendarstellung zu erstellen. die die grammatische Struktur der Tokens zeigt. Die Darstellung wird daher auch häufig als Syntaxbaum bezeichnet. Die Knoten stehen dabei für eine Operation und seine Kindknoten für die Argumente dieser Operation. Die Anordnung er Operationen stimmt mit üblichen arithmentischen Konventionen überein, wie zum Beispiel der vorrang der Multiplikation vor Addition.[13]

2.2.3 Semantische Analyse

Bei der semantischen Analyse wird der Syntaxbaum und die Informationen aus der Symboltabelle verwendet um das Quellprogramm auf semantische Konsistenz mit der Sprachdefinition zu überprüfen. Außerdem werden in dieser Typinformationen gesammelt und entweder im Syntaxbaum oder in der Symboltabelle hinterlegt um sie in späteren Phasen zu verwenden. Dabei werden außerdem Typen überprüft, daher analysiert ob jeder Operator die passenden Operanden hat.[14]

[11] Vgl. Ullman et al. 2008, S. 7f.
[12] Vgl. Ullman et al. 2008, S. 7f.
[13] Vgl. Ullman et al. 2008, S. 9.
[14] Vgl. Ullman et al. 2008, S. 9ff.

2.2.4 Zwischencodeerzeugung

Bei der Übersetzung eines Quellprogramms in den Zielcode kann der Compiler mehrere Zwischendarstellungen in verschiedenen Formen erstellen. Syntaxbäume sind beispielsweise eine solche Darstellung. Nach der semantischen Analyse stellen viele Compiler eine maschinennahe Zwischendarstellung die für eine Abstrakte Maschine entworfen wurden.[15]

2.2.5 Codeoptimierung

In dieser Phase wird der maschinenunabhängige Code so optimiert, dass sich darauf ein besserer Zielcode ergibt. Dabei bedeutet besser, schnellerer code oder code der weniger Ressourcen verbraucht. Der Umfang der Codeoptimierung schwankt dabei von Compiler zu Compiler erheblich.[16]

2.2.6 Codeerzeugung

Bei der Codeerzegung werden die Eingaben aus der Zwischendarstellung des Quellprogramms entgegengenommen und auf die Zielsprache abgebildet. Ein entsheidender Aspekt der Codeerzeugung ist die sinnvolle zuweisung von Registern für Variablen, falls es sich bei der Zielsprache um Maschinencode handelt.[17]

2.3 Rekursiver Ansatz

[15] Vgl. Ullman et al. 2008, S. 11.
[16] Vgl. Ullman et al. 2008, S. 11f.
[17] Vgl. Ullman et al. 2008, S. 13.

Literaturverzeichnis

Hunter, Scott (2020). *Introducing .NET Multi-platform App UI*. Website. Online erhältlich unter `https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/`; abgerufen am 28. Oktober 2020.

Joorabchi, Mona Erfani (Apr. 2016). „Mobile App Development: Challenges and Opportunities for Automated Support“. Diss. Vancouver: University of British Columbia.

Schneider, Hans-Jürgen (1975). *Compiler. Aufbau und Arbeitsweise*. Berlin: Walter de Gruyter.

Ullman, Jeffrey D. et al. (2008). *Compiler. Prinzipien, Techniken und Werkzeuge*. 2. Aufl. München: Pearson Studium.

Vollmer, Guy (2017). *Mobile App Engineering. Eine systematische Einführung - von den Requirements zum Go Live*. 1. Aufl. Heidelberg: dpunkt.

Wagenknecht Christian abd Hielscher, Michael (2014). *Formale Sprachen, abstrakte Automaten und Compiler. Lehr- und Arbeitsbuch für Grundstudium*. 2. Aufl. Wiesbaden: Springer.

Wissel, Andreas, Chrsitian Liebel und Thorsten Hans (2017). „Frameworks und Tools für Cross-Plattform-Programmierung“. In: *iX – Magazin für professionelle Informationstechnik* 2.

Eidesstattliche Erklärung

Studierender: Julian Pasqué
Matrikelnummer: 902953

Hiermit erkläre ich, dass ich diese Arbeit selbstständig abgefasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....	.....
Ort, Abgabedatum	Unterschrift (Vor- und Zuname)