Entwicklerhandbuch

Die C# Schnittstelle der Referenzattributgrammatik-gesteuerten Graphersetzungsbibliothek *RACR*

Übersicht, Anwendung und Implementierung

Daniel Langner

s8572327@gmail.com

Editor: Christoff Bürger

RACR .NET Entwicklerhandbuch
RACR Distribution und Homepage: `https://github.com/christoff-buerger/racr`

Danksagung

Ich danke Llewellyn ‘Leppie‘ Pritchard, dem Autor der Scheme-VM IronScheme, ohne welche diese Arbeit niemals hätte entstehen können. Er hatte immer ein offenes Ohr und war stets bereit, sein breites CLI-Wissen zu teilen und bei Problemen mit IronScheme weiterzuhelfen.

Vorwort

Dieses Entwicklerhandbuch beruht auf dem Großem Beleg von Daniel Langner, welcher unter dem Titel *RAG-gesteuerte Graphersetzung in der objektorientierten Programmierung* am 17. November 2015 beim Lehrstuhl Softwaretechnologie, Institut für Software- und Multimediatechnik, Technische Universität Dresden, eingereicht wurde.

Der Große Beleg wurde von Dipl.-inf. Christoff Bürger und Dipl.-Inf. Johannes Mey, unter Aufsicht von Prof. Dr. rer. nat. habil. Uwe Aßmann, betreut und mit *sehr gut* abgeschlossen.

Inhaltsverzeichnis

Abbildungs- und Tabellenverzeichnis

Quelltextverzeichnis

1. Einleitung

Referenzattributgrammatiken [14] (RAG) und Graphersetzung [22, 9] sind namhafte Konzepte im Übersetzerbau. Durch Referenzattribute lassen sich abstrakte Syntaxbäume (AST) deklarativ zu Graphen erweitern und analysieren. Sie eignen sich daher beispielsweise zur Entwicklung semantischer Analysen und Code-Generatoren. Üblicherweise kommen semantische Analysen und Graphersetzungen getrennt in unterschiedlichen Kompilierungsphasen zum Einsatz.

Zur Untersuchung der Vorteile einer engen Integration von Graphanalysen und -Ersetzungen wurde am Lehrstuhl Softwaretechnologie ein Verfahren namens Referenzattributgrammatik-gesteuerte Graphersetzung [4] entwickelt, das effiziente, sich wechselseitig bedingende Graphanalysen und -Ersetzungen ermöglicht (RAG-gesteuerte Graphersetzung). Entscheidungen bezüglich einer Graphersetzung können unter Zuhilfenahme von Attributen getroffen werden. Graphersetzungen ändern die Struktur des AST, wobei die Neuauswertung nur jener Attributen ausgelöst wird, die von der Modifikation betroffen sind. Zur Neuauswertung von Attributen, die von Graphersetzungen beeinflusst werden, wird eine bedarfsgesteuerte, inkrementelle Evaluierungsstrategie eingesetzt, in welche innerhalb der Attributgleichung genommenen Ausführungspfade einfließen. Attributabhängigkeiten werden dabei nur betrachtet, wenn diese zur Laufzeit auftreten, also während der Auswertung einer Attributgleichung relevant sind. Dieser dynamische Ansatz steht im Gegensatz zu bekannten inkrementellen Auswertungsverfahren, aufbauend auf statischen Attributabhängigkeitsanalysen [7, 19].

Eine Referenzimplementierung der RAG-gesteuerten Graphersetzung liegt in Form einer R6RS Scheme-Bibliothek namens RACR[1] vor [2, 4]. Sie umfasst Scheme-Prozeduren, um Grammatiken und Attribute zu spezifizieren, zur Syntaxbaum-Konstruktion, Traversierung, Abfrage von Knoteninformation und zur Attributsauswertung. Zusätzlich werden Prozeduren zur Graphersetzung bereitgestellt.

Die Vorteile der RAG-gesteuerten Graphersetzung wurden bereits in verschiedenen Bereichen demonstriert. Zum Beispiel wurde mittels RACR eine domänenspezifische Sprache[2] zur Generierung interaktiver Fragebögen (Language Workbench Challenge 2013[3] [12]) spezifiziert. Des Weiteren wurde RACR zur Implementierung von Laufzeit-Modellen für cyber-physische Systeme zur effizienten selbst-Optimierung hinsichtlich des Energiebedarfs eingesetzt [5]. Darüber hinaus werden in einer Lösung des Transformation Tool Contest 2015 [20] mittels RACR Transformationen von fUML-Aktivitätsdiagrammen nach ausführbaren Petrinetzen realisiert. Die erzeugten Petrinetze werden hierbei inkrementell von RACR ausgewertet, sodass eine sowohl effiziente als auch wohldefinierte Ausführungssemantik für fUML-Aktivitätsdiagramme gegeben ist [3].

[1] `https://github.com/christoff-buerger/racr`

[2] `https://github.com/christoff-buerger/racr/tree/master/examples/questionnaires`

[3] `http://www.languageworkbenches.net/wp-content/uploads/2013/11/Ql.pdf`

1.1. Aufgabenstellung

Um die Nützlichkeit des Verfahrens der RAG-gesteuerten Graphersetzung in zukünftigen Forschungsprojekten in der objektorientierten Programmierung untersuchen zu können, soll RACR in dieser Arbeit innerhalb einer objektorientierten Sprache über eine entsprechende Schnittstelle verfügbar gemacht werden.

Damit die existierende RACR-Implementierung genutzt werden kann, muss die Möglichkeit geschaffen werden, eine R6RS-konforme virtuelle Maschine von der Host-Sprache aus anzusteuern. Dies beinhaltet, dass Scheme-Prozeduren von der Host-Sprache aus aufgerufen werden können, und umgekehrt (Callback-Funktionen). Die Funktionalitäten der prozeduralen Schnittstelle RACRs müssen auf Objekte und Methoden abgebildet werden. Die entscheidenden Qualitätskriterien der Lösung sind eine benutzerfreundliche, objektorientierte Schnittstelle und geringe zusätzliche Laufzeitkosten im Vergleich zu einer reinen Scheme-Anwendung. Test-Anwendungen sind erforderlich, um die Korrektheit und eine akzeptable Performance der Schnittstelle zu gewährleisten.

Als Host-Sprache wurde C# ausgewählt. Folglich soll eine .NET-Klassenbibliothek namens RACR-NET geschaffen werden, mittels welcher der Funktionsumfang RACRs von C# aus über eine objektorientierte Schnittstelle nutzbar ist. Die Brücke zwischen Scheme und C# soll durch den Einsatz der IronScheme-VM[4] geschaffen werden.

1.2. Struktur der Arbeit

Die vorliegende Arbeit ist folgendermaßen gegliedert: Kapitel 2 gibt einen Überblick über die RAG-gesteuerte Graphersetzung und andere verwendete Technologien. Die tatsächliche Umsetzung erfolgt in zwei Schritten. Zunächst wird eine prozedurale Schnittstelle entworfen, welche in Kapitel 3 beschrieben wird. Die darauf aufbauende objektorientierte Schnittstelle sowie deren Implementierung werden in Kapitel 4 vorgestellt. Anschließend wird das Ergebnis in Kapitel 5 getestet und ausgewertet. Die Arbeit endet mit einer Zusammenfassung und einen Ausblick in Kapitel 6.

[4] `http://ironscheme.codeplex.com/`

2. Konzeptionelle und technische Voraussetzungen

Die Grundlagen von RACR-NET lassen sich in zwei Kategorien scheiden: Übersetzerbaukonzepte zur Spezifikation formaler Sprachen und Generierung entsprechender Sprachprozessoren und Technikräume, welche zur Implementierung genutzt wurden. Die RACR Scheme-Bibliothek setzt hierbei sowohl den konzeptionellen, als auch den technischen Rahmen fest: RAG-gesteuerte Graphersetzung und die Programmiersprache Scheme. Als Technikraum für eine objektorientierte Adaption RACRs wurde C# und somit das .NET-Framework gewählt.

2.1. Überblick der RAG-gesteuerten Graphersetzung

Attributgrammatiken [16, 17, 21] sind eine wichtige Technik im Übersetzerbau [18]. Attribute bilden ein Gleichungssystem zur deklarativen Spezifikation kontextsensitiver Analysen basierend auf einer kontextfreien Grammatik. Somit ermöglichen Attributgrammatiken die Spezifikation der Semantik kontextfreier Sprachen.

Schwächen zeigen Attributgrammatiken bei der Berechnung nichtlokaler, bedingter Relationen und aggregierter Attributwerte [15]. Zum Beispiel müssen für die Namensanalyse typischer Programmiersprachen, wie C++ oder Java, Informationen aus dem Syntaxbaum bezüglich der Deklarationen gesammelt und in den jeweiligen Attributen wiederholt zwischengespeichert werden. Diese Aggregat-Attribute können sehr komplex werden, was die Erweiterbarkeit der Grammatik erschwert – besonders für Sprachen mit komplizierten Sichtbarkeitsbereichen der Variablen, wie zum Beispiel objektorientierten Sprachen.

Referenzattributgrammatiken adressieren dieses Problem durch die Einführung von Referenzattributen – Attribute, deren Wert Referenzen auf beliebige Knoten innerhalb des Syntaxbaums sind [14]. Mittels Referenzattributen wird der Syntaxbaum gewissermaßen mit zusätzlichen Kannten versehen und zu einem Syntaxgraph erweitert. Über Referenzen können auf die Attribute entfernter Knoten zugegriffen werden. Auf diese Weise kann Information von einem Knoten zu einem anderen entfernten Knoten im Syntaxbaum direkt übertragen werden und muss nicht in komplexen Aggregat-Attributen gehalten werden.

Für effiziente inkrementelle Attributsauswertungen ist es entscheidend, dass Attribute nicht unnötig mehrfach berechnet werden [19]. Um Mehrfachauswertungen zu vermeiden, bietet es sich an, einmal berechnete Attributwerte in einem Cache-Speicher zu halten. Veränderungen am abstrakten Syntaxbaum (AST) erfordern jedoch eine Neuberechnung jener Attribute, die direkt oder indirekt von der modifizierten Baumstruktur abhängen. Deshalb müssen gegebenenfalls die Attribut-Caches invalidiert werden. Inkrementelle Attributsauswertung

wurde bislang nur für statische Auswertungsverfahren realisiert [2, 25]. Referenzattributgrammatiken erfordern jedoch eine dynamische Auswertung, da die Abhängigkeiten von auf Referenzattributen basierenden Attributen statisch unbekannt sind. Die inkrementelle Auswertung von Referenzattributgrammatiken war ein bis vor kurzem ungelöstes Problem.

Am Lehrstuhl Softwaretechnologie der Technischen Universität Dresden wurde von Christoff Bürger ein technisches Verfahren namens Referenzattributgrammatik-gesteuerte Graphersetzung [2, 4] entwickelt, das effiziente, inkrementelle Auswertung von Referenzattributen, sowie den Einsatz von Attributgrammatik-basierten Analysen zur Graphersetzungen ermöglicht. Attributsabhängigkeiten werden dynamisch während der Auswertung überwacht. Bei einer Graphtransformation werden lediglich die Caches der Attribute invalidiert, die von der Struktur des geänderten Teilbaumes abhängen. Entsprechend werden in weiteren Analysen nur diese Attribute neu berechnet.

2.2. Scheme

Scheme ist die Implementierungssprache von RACR. Sie ist dynamisch, vorwiegend funktional und bekannt für ihr elegantes, minimalistisches Design.

Es existieren viele Interpreter und Compiler für Scheme, die jedoch häufig von offiziellen Sprach-Standards abweichen. Deshalb wird Scheme oft als Überbegriff für eine ganze Familie von Dialekten angesehen. Das Scheme Steering Committee[1] selbst bezeichnete Scheme als die unportierbarste Sprache der Welt [6].

Die verschiedenen Versionen der Sprache werden Revised^n Report on the Algorithmic Language Scheme (RnRS) genannt. Der R6RS [26] brachte grundlegende, teils kontroverse Änderungen mit sich[2]. Um den Anforderungen der modernen Softwareentwicklung gerecht zu werden, wurde ein geregeltes Modulsystem eingeführt und die Sprache mit Standardbibliotheken ausgestattet. Da die Bibliotheken Teil des Standards selbst sind, ist der R6RS im Vergleich zu seinem Vorgänger sehr viel umfangreicher, was im Widerspruch zu der minimalistischen Philosophie der Sprache steht. Es finden sich nur wenige vollständig R6RS-konforme Scheme-Implementierungen[3].

Der aktuelle R7RS [24] teilt Scheme in zwei getrennte, zueinander kompatible Sprachen auf: eine kleine, pädagogische Sprache mit minimalistischem Charakter und eine moderne, zweckmäßige Programmiersprache zur Softwareentwicklung, die den R6RS ersetzen soll. Die kleine Sprache umfasst die wichtigsten Features des R6RS, wie Record-Typen, ein Bibliothekssystem und Exceptions, verzichtet aber auf viele zusätzliche, umfangreiche Bibliothekteile des R6RS.

Die wichtigsten Spracheigenschaften Schemes sind eine dynamische Typisierung, imperative Schreiboperationen, lexikalische Hüllen und First-Class-Prozeduren, automatische Speicherverwaltung und hygienische Makros. Scheme zwingt Programmierern kein spezifisches Paradigma, wie zum Beispiel logische, funktionale oder objektorientierte Programmierung, auf. Vielmehr

[1] Das Scheme Steering Committee ist ein aus drei Personen bestehendes Gremium, das den Standardisierungsprozess der Sprache übersieht (`http://www.r6rs.org/steering-committee/`).

[2] `http://www.r6rs.org/ratification/electorate.html`

[3] `http://www.r6rs.org/implementations.html`

ist Scheme dank seines mächtigen Makrosystems und der schlichten, prägnanten Syntax eine programmierbare Programmiersprache. Gewünschte Sprachkonzepte und Abstraktionen werden in Scheme selbst *hineinprogrammiert*.

Aufgrund der dynamischen Typisierung und der automatischen Speicherverwaltung Schemes, welche die Freigabe von dynamisch allozierten Speicherplatz erfordert, wenn dieser nicht mehr benötigt wird, stellen Scheme-Implementierungen meist eine eigene Laufzeitumgebung zur Programmausführung bereit (virtuelle Maschine). Scheme-Implementierungen tendiert daher eher zu einem Interpreter- statt Übersetzeransatz.

2.3. Die RACR Scheme-Bibliothek

RACR [2] ist die Referenzimplementierung der RAG-gesteureten Graphersetzung. Hierbei handelt es sich um eine R6RS-konforme Scheme-Bibliothek, welche in dieser Arbeit in C# integriert werden soll. Die in ihr enthaltenen Prozeduren lassen sich in folgende Bereiche aufgliedern:

- Definition von abstrakten Syntaxbaum-Schemata mithilfe von Nichtterminal-Klassen, Kompositen, typisierten Kind-Elementen und Vererbung zwischen Nichtterminal-Klassen (erweiterte Backus-Naur-Form [23], basierend auf Vererbung zwischen Nichtterminalen [13])
- Attribuierung der Nichtterminale unter Nutzung normaler Scheme-Prozeduren als Attributgleichungen
- Erzeugung von Syntaxbäumen für eine bestimmte Sprachspezifikation
- Baum-Traversierung und Abfrage von Attributen und Knoten-Informationen (Die hierfür von RACR bereitgestellten Funktionen werden innerhalb von Attributgleichungen zur Berechnung derselben genutzt.)
- Annotation von AST-Knoten
- Graphersetzung

Eine RACR-Anwendung kann für gewöhnlich in zwei Teile gegliedert werden. Der erst Teil umfasst die Spezifikationsphase der Sprache, in welcher zuerst Grammatikregeln und danach Attribute definiert werden. Attributgleichungen werden von Prozeduren verkörpert, die den Attributwert für einen AST-Knoten errechnen. Bemerkenswerterweise geschieht die Sprachdefinition zur Programmlaufzeit, was die Generierung von Sprachen abhängig von Laufzeitinformationen erlaubt. Insbesondere wird keine Typ-Hierarchie über die Nichtterminale angelegt. Stattdessen bietet RACR für Nichtterminale jedes Typs eine generische Schnittstelle. Attribute sind somit nicht an einen bestimmen Typ gekoppelt. Sie können an verteilten Stellen im Programm definiert werden. Attribute sind dynamische Inter-Typ-Deklarationen gemäß der aspektorientierten Programmierung [1], eine bewährte Technik für die Implementierung erweiterbarer Sprachprozessoren [10].

Nachdem die Sprache spezifiziert wurde, können ASTs instanziiert werden und Attribute von AST-Knoten ausgewertet werden. Die inkrementelle Attributsauswertung, der wichtigste

Vorteil von RACR gegenüber vielen anderen Referenzattributgrammatik=Werkzeugen, kommt bei Graphersetzungen und anschließenden Attributsauswertungen zum Tragen. Gleichzeitig profitiert die Graphersetzung von attributbasierten Analysen. Beispielsweise kann der Wert von Attributen eine Prozedur sein, welche Ersetzungsregel-Anwendungen kapselt. Attribute können also zur Ableitung von Ersetzungen genutzt werden. Ein Vorteil dieses Ansatzes ist, dass die Mustererkennung (das Subgraphisomophismus-Problem [27]) der Graphersetzung in Attribute kodiert ist und damit ebenfalls inkrementell erfolgt.

Das in Abbildung 2.1 gezeigte Zustandsdiagramm stellt alle Prozeduren RACRs in Beziehung.

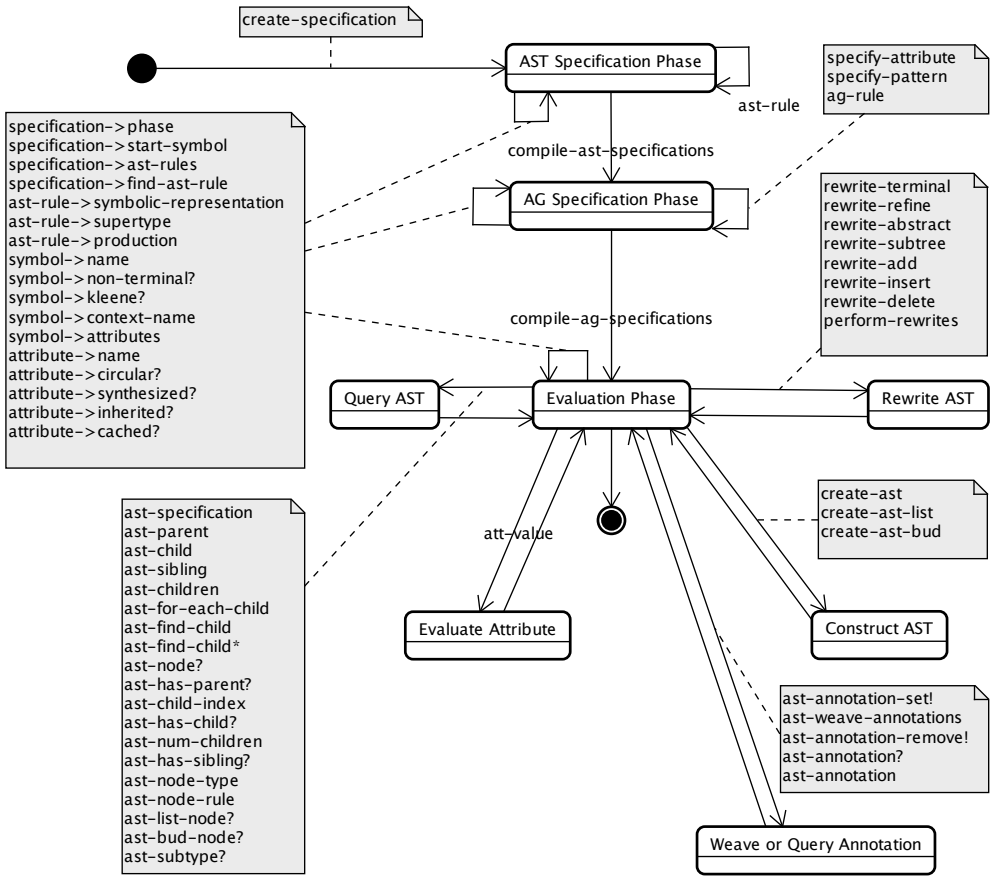

Abbildung 2.1.: RACR API [2]

2.4. Das .NET-Framework und die Common Language Infrastructure

.NET ist ein von Microsoft entwickeltes Software-Framework und eine integrale Komponente des Windows-Betriebssystems. Für .NET entwickelte Programme werden nicht direkt nach Machinencode, sondern nach Bytecode – der Intermediate Language (IL) – übersetzt und

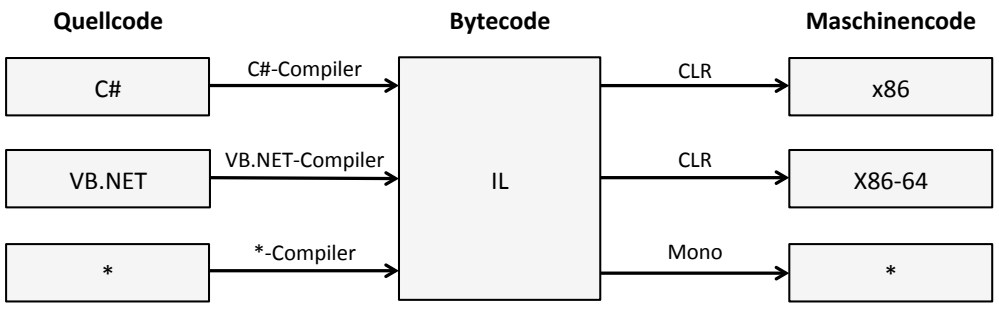

Abbildung 2.2.: Übersetzung von .NET-Sprachen nach Maschinencode

laufen innerhalb eines virtuellen Ausführungssystems. Programmiersprachen, die nach IL kompilieren, werden .NET-Sprachen genannt, wobei C# und Visual Basic .NET zu den wichtigsten Vertretern gehören. Der IL-Code wird in sogenannten Assemblies in Form von EXE-Dateien (Prozess-Assemblies) oder DLL-Dateien (Library-Assemblies) gespeichert.

Die stapelbasierende virtuelle Maschine (VM) von .NET wird innerhalb der Common Language Infrastructure (CLI) [8], einem internationalen Standard, definiert und bietet sprach-neutrale Features, wie zum Beispiel automatische Speicherbereinigung, Exceptions, Typsicherheit, Zugriff auf eine umfangreiche Klassenbibliothek und Just-in-Time-Kompilierung, wobei entsprechend der zu Grunde liegenden Rechnerarchitektur IL-Code bedarfsgesteuert zu nativem Code übersetzt wird. Neben der Common Language Runtime (CLR), der Microsoft-Implementierung der CLI, existieren Portable.NET[4] und das Mono-Projekt[5] als kompatible, quelloffene Alternativen.

Abbildung 2.2 veranschaulicht die Kompillierungsprozesse, die der Ausführung eines .NET-Programms vorhergehen müssen. Alle .NET-Programme haben gleichen Bytecode, unabhängig von der Implementierungssprache oder der Zielarchitektur. Dies hat folgende wichtige Konsequenzen zur Folge:

- .NET-Programme sind nicht architekturgebunden, sondern plattformunabhängig. Ein unter Windows entwickeltes Programm kann nach der Kompilierung unter Linux (mittels Mono) ausgeführt werden und umgekehrt.

- .NET-Sprachen haben zueinander hohe Interoperabilität. Die CLI vereinheitlicht unter anderem Exception-Handling, den Aufbau von und Zugriff auf Klassenbibliotheken und die Interaktion zwischen allen Datentypen. Objektinstanzen können über Sprachgrenzen hinweg ausgetauscht werden.

[4] http://www.dotgnu.org/
[5] http://www.mono-project.com/

2.5. IronScheme

IronScheme[6] ist eine quelloffene, weitgehend R6RS-konforme Scheme-Implementierung für die .NET Software-Plattform. Sie umfasst einen eigenständigen, interaktiven Interpreter zur Ausführung von Scheme-Programmen und eine Klassenbibliothek, mittels welcher IronScheme in andere .NET-Projekte eingebunden werden kann. Diese Klassenbibliothek erlaubt es, .NET–Strings[7] als Scheme-Ausdruck auszuwerten, analog zu der `Eval`-Funktion, wie sie in vielen dynamischen Sprachen zu finden ist. So wird während der Programmausführung aus dem Scheme-Code IL-Code generiert und dieser anschließend ausgeführt. Scheme-Bibliotheken können jedoch auch zu Assemblies vorkompiliert werden.

Scheme	.NET
`fixnum`	`System.Int32`
`flonum`	`System.Double`
`boolean`	`System.Boolean`
`string`	`System.String`
`pair`	`IronScheme.Runtime.Cons`
`'()`	`null`
`hashtable`	`System.Collections.Hashtable`
`symbol`	`Microsoft.Scripting.SymbolId`
`procedure`	`Ironscheme.Runtime.Callable`

Tabelle 2.3.: IronSchemes Abbildung von Scheme-Datentypen auf .NET-Datentypen

Viele wichtige Datentypen sind als Adapter für .NET-Datentypen realisiert, was die Interoperabilität von IronScheme zu anderen .NET-Sprachen erhöht. Andere Datentypen implementieren spezielle Schnittstellen. Die für diese Arbeit relevanten Scheme-Typen und deren zugehörige .NET-Typen sind in Tabelle 2.3 aufgeführt.

[6] `http://ironscheme.codeplex.com/`

[7] Genauer gesagt handelt es sich um den Typ `System.String` – die Klasse `String` im Namensraum `System`.

3. RACR-NET Implementierung: Prozedurale Schnittstelle

Dieses Kapitel beschäftigt sich mit der Umsetzung einer prozeduralen Schnittstelle, welche es ermöglicht, RACR in C# zu nutzen. Zuerst wird gezeigt, wie innerhalb von C# Scheme-Code aufgerufen und wie die RACR-Scheme-Bibliothek effizient geladen werden kann. Nach einer Anforderungsanalyse wird die Implementierung vorgestellt. Die abschließende Evaluation zeigt die Schwächen einer rein prozeduralen Lösung und leitet zur finalen objektorientierten Lösung im folgenden Kapitel über.

3.1. Scheme in C#

Mittels der IronScheme-Klassenbibliothek kann Scheme-Code von C# aus ausgeführt werden. IronScheme bedient sich dazu der Extension-Methoden – ein Feature von C#, das es erlauben, einem existierenden Typen neue Methoden hinzuzufügen, ohne den ursprünglichen Typ zu manipulieren. Extension-Methoden sind spezielle statische Methoden, die vom Nutzer wie Instanz-Methoden aufgerufen werden.

IronSchemes `Eval` ist eine solche Methode. Sie interpretiert .NET-Strings als Scheme-Ausdrücke und gibt einen Wert vom Typ `object` zurück, dem Basistypen aller .NET-Datentypen. Da C# statisch typisiert ist, müssen Rückgabewerte zur sinnvollen Weiterverwendung üblicherweise explizit in einen Subtypen konvertiert werden. Deshalb bietet IronScheme auch eine generische Variante der `Eval`-Methode. Diese akzeptiert einen Typ-Parameter, der den Typ des Rückgabewerts festlegt. Die Typumwandlung erfolgt hierbei innerhalb von `Eval`.

Bei Referenztypkonvertierungen wie dieser kann während der Code-Übersetzung nicht bestimmt werden, ob die Umwandlung gültig ist. Schlägt zur Laufzeit eine Typumwandlungsoperation fehl, wird eine `InvalidCastException` ausgelöst. Quelltext 3.1 veranschaulicht die Arbeitsweise von `Eval`.

```
1 object a = "(+ 1 2)".Eval();
2 int b = (int) a;                    // explizite Umwandlung
3 bool c = "(< 3 4)".Eval<bool>();    // generische Methode
4
5 bool d = (bool) "(* 2 3)".Eval();   // Laufzeitfehler!
6 bool e = "(* 2 3)".Eval<bool>();    // Laufzeitfehler!
```

Listing 3.1: Auswerten von Scheme-Ausdrücken

Alle Scheme-Prozeduren implementieren das `Callable`-Interface, mittels welchem man eine Prozedur ohne mehrfaches Parsen oder Kompilieren wiederholt aufrufen kann. Diese abstrakte Klasse stellt `Call`-Methoden in verschiedenen Ausführungen bereit – variierend über die Anzahl von Parametern, welche wie auch der Rückgabewert stets vom Typ `object` sind. Auf diese Weise wird die dynamische Typisierung von Scheme in C# abgebildet. Beim Aufruf eines `Callable`-Objektes kann der Compiler für die korrekte Stelligkeit sowie die Typisierung der Parameter einer Prozedur nicht garantieren. Typfehler äußern sich erst während der Laufzeit eines Programms. Quelltext 3.2 zeigt das Interface in Aktion.

```
1 Callable sum = "+".Eval<Callable>();
2 int a = (int) sum.Call(1, 2);
3 int b = (int) sum.Call(3, 4, 5);
4 int c = (int) sum.Call(0, false);   // Laufzeitfehler!
```

Listing 3.2: Verwendung des `Callable`-Interfaces

3.2. RACR in C#

3.2.1. Importieren der Scheme-Bibliothek

Der nächste Schritt besteht darin, RACR dem Scheme-Environment bekannt zu machen. RACR laden wir mittels `"(import (racr core))".Eval()`. Dieser Aufruf veranlasst IronScheme dazu, den Quellcode RACRs, der in der Datei `racr/core.sls` residiert, zu kompilieren und alle darin als Export deklarierten Symbole zu registrieren. Dies geschieht mit jedem Neustart des C#-Programms, was dessen Anlaufzeit verlängert.

Ein weniger dokumentiertes Feature IronSchemes erlaubt es, Scheme-Bibliotheken vorzukompilieren und in Assemblies zu speichern. Auf diese Weise kann das Laden von RACR stark beschleunigt werden. Zum Beispiel lege man eine Datei namens `tmp.scm` mit dem Inhalt `(import (racr core))` an. Der Aufruf von `(compile "tmp.scm")` in der interaktiven IronScheme-Konsole generiert die Datei `racr.core.dll`. Wenn diese Assembly bei der Kompilierung eines C#-Programms referenziert wird, so kompiliert IronScheme RACR mit dem Aufruf von `import` nicht neu, sondern lädt den kompilierten Code aus der DLL-Datei.

3.2.2. Konformitätsprüfung von IronScheme

Wie im vorhergehenden Kapitel beschrieben, unterscheiden sich Scheme-Interpreter bezüglich ihrer R6RS-Konformität. Um die korrekte Funktion von RACR unter Verwendung verschiedener Interpreter zu gewährleisten, beinhaltet RACR eine Test-Umgebung, die dessen wesentlichen Funktionsumfang abdeckt. Diese Tests wurden mit IronScheme ausgeführt. Dabei terminierten vorerst alle Tests mit dem gleichen Fehler: Der Aufruf von `hashtable-set!` mit dem Schlüssel `'()` löst eine `System.ArgumentNullException` aus. Der Fehler wird durch eine R6RS-Unkonformität IronSchemes verursacht: RACR benutzt häufig die leere Liste als Schlüssel, um Daten in einer `hashtable`-Datenstruktur abzulegen. Wie Tabelle 2.3 zu entnehmen,

bildet IronScheme `hashtable` auf den CLI-internen Typ `System.Collections.Hashtable` ab. Des Weiteren wird die leere Liste `'()` auf `null` abgebildet. Laut MSDN[1] darf der Schlüssel eines `Hashtable`-Objektes jedoch nicht `null` sein.

Ein Workaround in Form einer Scheme-Bibliothek soll bijektiv die leere Liste auf einen internen Record abbilden. Quelltext 3.3 zeigt die Definition dieses Records und dessen Verwendung in den Prozeduren `hashtable-ref*` und `hashtable-set!*`, welche das Interface von `hashtable-ref` beziehungsweise `hashtable-set!` reimplementieren.

```
1  (define-record-type nil-record (sealed #t) (opaque #t))
2  (define nil (make-nil-record))
3
4  (define hashtable-ref*
5    (lambda (h key default-value)
6      (hashtable-ref h (if (null? key) nil key) default-value)))
7
8  (define hashtable-set!*
9    (lambda (h key value)
10     (hashtable-set! h (if (null? key) nil key) value)))
```

Listing 3.3: `hashtable`-Workaround

Auf Zeile 1 wird ein neuer Record-Typ angelegt und anschließend auf Zeile 2 instanziiert. In den Definitionen der `hashtable`-Prozeduren wird mittels einer `if`-Anweisung geprüft, ob es sich bei dem Schlüssel um das `null`-Objekt handelt, in welchem Falle der `nil`-Record als Schlüssel benutzt werden soll (Zeilen 6 und 10). Auf ähnliche Weise wurde das Interface folgender weiterer Scheme-Prozeduren reimplementiert: `hashtable-delete!`, `hashtable-contains?` und `hashtable-entries`. Diese Bibliothek wurde in RACR eingebunden und alle Aufrufe der genannten Prozeduren auf die entsprechenden Reimplementierungen umgelenkt. Alle Tests der RACR-Test-Umgebung liefen nun fehlerlos. Somit ist sichergestellt, dass RACR unter IronScheme korrekt ausgeführt wird.

3.3. Anforderungsanalyse

Um die Qualität der RACR-C#-Schnittstelle bewerten zu können, müssen geeignete Kriterien festgelegt werden. Bislang steht dem Programmierer RACR nur als Scheme-Bibliothek zur Verfügung. Folglich müssen sämtliche Aspekte einer Anwendung, die den Funktionsumfang von RACR nutzen sollen, in Scheme programmiert sein. Mit der Möglichkeit, RACR auch in C# nutzen zu können, ergeben sich neue Anwendungsfälle, in denen die verschiedenen Aspekte einer RACR-Anwendung zu variierenden Anteilen in Scheme oder C# implementiert sein können. In einem Beispiel-Szenario werden AST-Regeln und Attribute einer Sprachspezifikation in Scheme definiert. Mittels dieser Spezifikation werden von C# aus ASTs instanziiert und Analysen betrieben. Ein weiterer denkbarer Fall ist, dass eine in Scheme spezifizierte Sprache in C# um zusätzliche Attribute erweitert wird. Abbildung 3.1 visualisiert diese und weitere

[1] `https://msdn.microsoft.com/library/system.collections.hashtable.aspx`

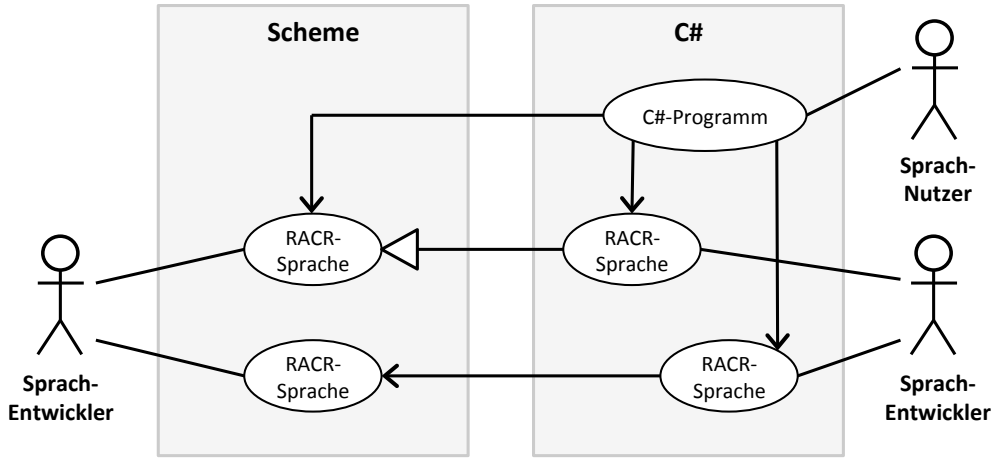

Abbildung 3.1.: Anwendungsfälle von RACR in Scheme und C#

Szenarien. Es ergibt sich eine Vielzahl möglicher Varianten der Sprachnutzung, Kopplung und Vererbung und der Aufteilung deren Implementierungen in Scheme und C#.

Diese Arbeit fokussiert sich auf das Szenario, in dem der Nutzer RACR ausschließlich von C# aus bedient. Angesichts dessen werden folgende Anforderungen an das System gestellt:

A1 – Vollständigkeit: Alle dokumentierten [2] Funktionalitäten, die RACR in Scheme bietet, sollen uneingeschränkt auch von C# aus nutzbar sein.

A2 – Dynamik: Die dynamische Natur RACRs muss erhalten bleiben. Grammatik und Attribute sollen noch während der Programmlaufzeit festgelegt werden können. Dies schließt Code-Generatoren, die anhand einer Spezifikation C#-Code erzeugen, aus.

A3 – Nähe zur originalen Schnittstelle: Der Wechsel zwischen Scheme und C# soll für mit RACR vertraute Programmierer intuitiv sein.

A4 – Entkoppelung von Scheme: Dass im Hintergrund die eigentliche Arbeit in einer Scheme-VM passiert, soll für C#-Nutzer irrelevant sein und sich in keinster Weise in der Schnittstelle widerspiegeln. Insbesondere sollen IronScheme-eigene Datentypen, wie zum Beispiel Paare, nicht in der Schnittstelle vorkommen. Stattdessen sollen äquivalente, .NET-typische Typen zum Einsatz kommen.

Im Nachfolgenden soll eine funktionierende, zunächst prozedurale Schnittstelle geschaffen werden, welche die Anforderungen **A1** bis **A4** erfüllt. Die endgültige objektorientierte Schnittstelle wird in Kapitel 4 behandelt.

3.4. Implementierung der prozeduralen Schnittstelle

Die prozedurale Schnittstelle soll vollständig sein und die Funktionsweise RACRs nicht einschränken. Ein naiver Ansatz zur Implementierung einer solchen Schnittstelle besteht darin,

für jede Scheme-Prozedur in C# ein direktes Gegenstück zu modellieren – in Form einer statischen Methode. Innerhalb dieser Methoden soll unter Verwendung der IronScheme-VM die entsprechende Prozedur aufgerufen werden. Auf diese Weise werden die Anforderungen **A1** und **A2** erfüllt.

Um Anforderung **A3** zu befriedigen, müssen die Methodennamen denen der zugehörigen Prozeduren gleichen. RACR hält sich bei der Benennung von Variablen beziehungsweise Prozeduren an die für Lisp-Sprachen typischen Konventionen und verwendet in Bezeichnern einige Sonderzeichen. Beispielsweise werden innerhalb eines Bezeichners Worte durch Bindestriche getrennt. Bezeichner von Prozeduren, die einen booleschen Wert liefern, enden für gewöhnlich mit einem Fragezeichen. C# hat seine eigenen Namenskonventionen. Sonderzeichen in Bezeichnern sind nicht gestattet. Alle Namen öffentlicher Member, Typen und Namespaces beginnen mit einem Großbuchstaben, Parameternamen jedoch mit einem Kleinbuchstaben. Beiderseits werden innerhalb eines Bezeichners verkettete Wörter durch Binnenmajuskel hervorgehoben.

Bei der Benennung der Methoden der .NET-Schnittstelle für RACR soll die C#-übliche Namenskonvention eingehalten werden ohne den Prozedurnamen zu entfremden. Aus `create-ast` wird `CreateAst`. Alle anderen Benennungen geschehen analog. Somit wird die Realisierung von Anforderung **A3** gewährleistet. Quelltext 3.4 skizziert diesen Ansatz.

```
1  public static class Racr {
2      private static Callable createSpecification;
3      private static Callable astRule;
4      // ...
5      static Racr() {
6          "(import (racr core))".Eval();
7          createSpecification = "create-specification".Eval<Callable>();
8          astRule             = "ast-rule".Eval<Callable>();
9          // ...
10     }
11     public static object CreateSpecification() {
12         return createSpecification.Call();
13     }
14     public static void AstRule(object spec, string rule) {
15         astRule.Call(spec, SymbolTable.StringToObject(rule));
16     }
17 // ...
```

Listing 3.4: Prozedurale C#-Schnittstelle für RACR

Die statische Klasse `Racr` umfasst private `Callable`-Felder, die Referenzen auf die entsprechenden Scheme-Prozeduren RACRs speichern sollen. Im statischen Konstruktor wird zuerst die RACR-Bibliothek geladen (Zeile 6), woraufhin alle `Callable`-Objekte initialisiert werden. Zusätzlich enthält die Klasse für jedes `Callable` eine statisch Methode, die als Adapter fungiert, indem sie ihre Argumente an die `Call`-Methode des entsprechenden `Callable`-Objekt durchreicht und dessen Rückgabewert zurückliefert. Ein triviales Beispiel hierfür ist die parameterlose Methode `CreateSpecification` (Zeile 11).

3.4.1. Entkopplung von Scheme

Um die Anforderung **A4** zu erfüllen, werden gegebenenfalls Umwandlungen Scheme-eigener Datentypen erforderlich. RACR nutzt Symbole, Paare und Listen – sowohl für Funktionsparameter als auch für Rückgabewerte. Des Weiteren erwarten einige Prozeduren als Argument selbst eine Prozedur. Der Umgang mit diesen Scheme-typischen Datentypen von C# aus ist unbequem und ineffizient, weswegen sie in RACRs C#-Schnittstelle vermieden werden sollen, sodass der Nutzer nicht mit ihnen konfrontiert wird. An ihrer statt sollen charakteristischere Typen zum Einsatz kommen. Die Implementierung der Schnittstelle muss für die Übersetzung solcher Typen Sorge tragen. Im Folgenden soll auf jene Datentypen eingegangen werden, die zur Verarbeitung in C# einer Sonderbehandlung bedürfen.

Symbole

Scheme-Symbole tauchen in RACRs Schnittstelle an vielen Stellen auf. Aufseiten von C# sollen stattdessen .NET-Strings zum Einsatz kommen. Symbole werden unter anderem als Zeichenketten für Nichtterminale, ganze AST-Regeln und Attributsnamen verwendet. IronScheme implementiert Symbole mittels der Struktur `SymbolId`, deren `ToString` Methode die String-Repräsentation der jeweiligen Symbol-Instanz liefert. Analog bildet die statische Methode `SymbolTable.StringToObject` Strings auf Symbole ab. Quelltext 3.4 zeigt die Verwendung von `SymbolTable.StringToObject` in der Methode `AstRule` (Zeile 15).

Paare

Paare sind in Scheme ein essenzieller Datentyp. In RACRs Scheme-Schnittstelle kommen Paare in folgenden Prozeduren zum Einsatz: `ast-children`, `ast-for-each-child`, `ast-find-child` und `ast-find-child*`. Diese Prozeduren verlangen neben anderen Arguenten eine unbestimmte Anzahl von sogenannten Kinder-Intervallen – Paare, die in Form von zwei Indices eine untere und ober Grenze enthalten. Mit dem Scheme-Symbol `'*` kann auch eine offene obere Grenze angegeben werden[2]. Paare entsprechen in C# Objekten der Klasse `Cons` aus der IronScheme-Klassenbibliothek. Nutzer sollen mit ihr nicht in Berührung kommen müssen, sondern zur Angaben von Kinder-Intervallen eine für genau diesen Zweck geschaffene Datenstruktur verwenden. Quelltext 3.5 zeigt die dafür konzipierte Struktur `Range`.

```
1 public struct Range {
2     public int min;
3     public int max;
4     public Range(int min, int max=0) {
5         this.min = min;
6         this.max = max;
7     }
8     internal Cons ToCons() {
9         return new Cons(min,
```

[2] https://github.com/christoff-buerger/racr/blob/master/documentation/abstract-syntax-trees.md#ast-children

```
10                          max > 0 ? max : SymbolTable.StringToObject("*"));
11      }
12  }
```

Listing 3.5: Definition der `Range`-Struktur

Zweckmäßig hält die Struktur zwei Felder für die beiden Grenzen, wobei eine obere Grenze mit dem Wert 0 als offen interpretiert wird[3]. Die interne Methode `ToCons` dient dazu, aus dem `Range`- ein `Cons`-Objekt zu konstruieren, das von IronScheme aus weiterverarbeitet werden kann. Sie kommt in den Methoden des Interfaces, welche die oben genannten Prozeduren abbilden sollen, zum Einsatz.

Listen

Scheme-Listen sind verschachtelte Scheme-Paare. Sie sind ein Parameter der Prozeduren `create-ast` und `create-ast-list` und der Rückgabetyp der Prozeduren `ast-children` und `rewrite-abstract`. Analog zu `create-ast-list`, erwartet die `create-ast` als letztes Argument eine Liste mit den Kindern des zu erzeugenden AST-Knoten. In der C#-Schnittstelle soll für den Nutzer der Zwischenschritt, erst eine Liste für die Kinder zu konstruieren, übersprungen werden. Stattdessen soll die Methode `CreateAst` ein C#-Array von Kind-Knoten akzeptieren. Unter Verwendung einer einfachen Schleife über das Array soll daraus die Scheme-Liste erzeugt werden, damit sie anschließend beim Aufruf der Scheme-Prozedur übergeben werden kann.

```
1  public static object CreateAst(object spec, string nonTerm,
2                                  params object[] children)
3  {
4      Cons list = null;
5      for (int i = children.Length - 1; i >= 0 ; i--) {
6          list = new Cons(children[i], list);
7      }
8      return createAst.Call(spec, SymbolTable.StringToObject(nonTerm),
9                            list);
10 }
```

Listing 3.6: Listenkonstruktion in `CreateAst`

Quelltext 3.6 zeigt die Implementierung der Methode `CreateAst` der Klasse `Racr` gemäß den oben genannten Vorgaben. Man beachte, dass dem Parameter `children` das Schlüsselwort `params` vorangestellt ist. Es bewirkt, dass beim Aufruf der Methode eine variable Anzahl von Methodenargumenten automatisch zu einem Array zusammengefasst wird, sodass der Nutzer das Array nicht selbst anzulegen braucht. Will man beispielsweise einen Knoten mit drei Kind-Knoten erzeugen, gestaltet sich der Aufruf von `CreateAst` wie folgt:

[3] In RACR werden Indices stets von 1 an gezählt.

```
object node = Racr.CreateAst(spec, "A", childA, childB, childC);
```

Auch als Rückgabewert sollen in der C#-Schnittstelle Scheme-Listen durch Arrays ersetzt werden. Die in Quelltext 3.7 abgebildete Implementierung von `RewriteAbstract` nutzt eine `while`-Schleife, um die Kette von `Cons`-Objekten zu durchwandern und die in der Liste gespeicherten Kind-Knoten einer `List<object>` anzuhängen. Die Klasse `List` ist Teil der .NET-Klassenbibliothek. Im Gegensatz zum Array bietet sie Methoden, um die Anzahl ihrer Elemente dynamisch zu ändern.

```
1  public object[] RewriteAbstract(string supertype) {
2      var list = rewriteAbstract.Call(ast,
3                 SymbolTable.StringToObject(supertype)) as Cons;
4      var children = new List<object>();
5      while (list != null) {
6          children.Add(list.car);
7          list = list.cdr as Cons;
8      }
9      return children.ToArray();
10 }
```

Listing 3.7: Arraykonstruktion in `RewriteAbstract`

Prozeduren

Schemes Prozeduren sind First-Class-Objekte und können Funktionen als Argumente überreicht werden. Insofern ist Scheme eine höhere, funktionale Sprache. Wie schon erwähnt, sind Prozeduren aus der Sicht von .NET Subtypen der abstrakten Klasse `Callable`. C# unterstützt First-Class-Funktionen in Form von Delegaten – Referenztypen, mit denen eine benannte oder anonyme Methode gekapselt werden kann. Sie haben Ähnlichkeit mit Funktionszeigern in C. Der Typ eines Delegat-Objektes entspricht einer bestimmten Methodensignatur, die sich aus der Anzahl und den Typen der Parameter und dem Rückgabetyp zusammensetzt. Um C#-Methoden an RACR zu übergeben, müssen diese zuerst in ein `Callable` umgewandelt werden. IronScheme bietet zu diesem Zweck die Extension-Methode `ToSchemeProcedure` der Klasse `Delegate`, die Basisklasse aller Delegat-Typen. Innerhalb von RACR erwarten folgende Prozeduren eine Funktion als Argument: `ast-for-each-child`, `ast-find-child`, `ast-find-child*` und `specify-attribute`. Dabei gleichen sich die ersten drei insofern, dass die Signatur der zu übergebenden Prozedur von RACR vorgegeben ist, woraus sich für die C#-Schnittstelle ein entsprechender Delegat-Typ ergibt.

```
1  public static object AstFindChild(object node, Func<int,object,bool> f,
2                                    params Range[] bs)
3  {
4      object[] args = new object[2 + bs.Length];
5      args[0] = f.ToSchemeProcedure();
6      args[1] = node;
```

```
7       for (int i = 0; i < bs.Length; i++) args[i + 2] = bs[i].ToCons();
8       object res = astFindChild.Call(args);
9       if (res is bool && (bool) res == false) return null;
10      return res;
11  }
```

Listing 3.8: Delegat-Parameter in `AstFindChild`

Quelltext 3.8 zeigt die Implementierung von `AstFindChild`. Die Methode ist variadisch, genau wie die Scheme-Prozedur, an die sie ihre Argumente weiterleiten muss. Um eine unbestimmte Anzahl von Argumenten an den Aufruf eines `Callable`-Objekts weiterzugeben, müssen diese in ein Objekt-Array geschrieben werden, welches als einziger Parameter an `Call` übergeben werden muss. Zeile 4 deklariert dieses Array und die drei darauffolgenden Zeilen befüllen es. Dem Delegat-Typen `Func<int,object,bool>` zufolge muss die zu übergebende Funktion zwei Parameter mit den Typen `int` und `object` für den Index des Knoten beziehungsweise den Knoten selbst habe, nebst dem Rückgabetypen `bool`. `ToSchemeProcedure` nutzt die durch Introspektion zugänglichen Typinformationen, um ein `Callable`-Objekt zu generieren, in welchem der Delegat gekapselt wird. Die Methode `AstFindChild` weicht in ihrem Verhalten von `ast-find-child` ab, indem sie bei einer missglückten Suche statt `false` `null` zurückgibt (Zeile 9). Dies entspricht der in C# üblichen Weise, das Fehlen eines Objektes zu kennzeichnen.

Der Aufruf von `AstFindChild` gestaltet sich unter Benutzung von C#'s Lambda-Ausdrücken elegant. Hier ein Beispiel:

```
object child = Racr.AstFindChild(parent, (i, n) => {
        bool success;
        // setze success anhand von Attributsauswertungen, etc.
        return success;
    }, new Racr.Range(2, 7));
```

Mit `specify-attribute` lassen sich (gegebenenfalls Referenz-) Attribute definieren. Dieser Prozedur muss dabei unter anderem eine Attributgleichung (in Form eines Delegaten) übergeben werden. RACR unterstützt parametrisierte Attribute, was sich darin äußert, dass die Attributgleichungsfunktion neben einem Argument für den AST-Knoten noch weitere Argumente akzeptiert, mit welchem die auszuwertende Instanz des Attributes assoziiert ist. Bei der Attributsauswertung mittels `att-value` muss die entsprechende Anzahl an Argumenten des Attributs mit übergeben werden. Für die Korrektheit der Argumenttypen ist stets der Programmierer allein verantwortlich.

Aus der Sicht einer statisch typisierten Sprache wie C# stellt diese Dynamik in der Attributsspezifikation und -auswertung eine Herausforderung dar, da der exakte Typ der Attributgleichungsfunktion beinahe beliebig sein kann: Nur der erste Parameter wird von RACR als AST-Knote vorgegeben. Die Implementierung der Methodengruppe `SpecifyAttribute`[4] ist in Quelltext 3.9 abgebildet. Die erste Methode besitzt für die Attributgleichung einen Parameter vom Typ `Delegate`. Lambda-Ausdrücke werden vom Compiler jedoch nicht implizit nach

[4] Der Einfachheit halber ist hier eine Implementierung gezeigt, die keine zyklischen Attribute unterstützt.

`Delegate` umgewandelt. Um den Nutzer dennoch von verbosen Typumwandlungen zu entlasten, decken die zwei zusätzlichen generischen Ausführungen von `SpecifyAttribute` den Fall für parameterlose (`Func<object,R>`) beziehungsweise mit einem Parameter parametrisierte (`Func<object,T,R>`) Attribute ab.

```
public static void SpecifyAttribute(object spec, string name,
        string nonTerm, string context, bool cached, Delegate eq)
{
    specifyAttribute.Call(spec,
                          SymbolTable.StringToObject(name),
                          SymbolTable.StringToObject(nonTerm),
                          SymbolTable.StringToObject(context),
                          cached,
                          eq.ToSchemeProcedure(),
                          false);
}
public static void SpecifyAttribute<R>(object spec, string name,
        string nonTerm, string context, bool cached, Func<object,R> eq)
{
    SpecifyAttribute(spec, name, nonTerm, context, cached, (Delegate) eq);
}
public static void SpecifyAttribute<T,R>(object spec, string name,
        string nonTerm, string context, bool cached, Func<object,T,R> eq)
{
    SpecifyAttribute(spec, name, nonTerm, context, cached, (Delegate) eq);
}
```

Listing 3.9: Methodengruppe `SpecifyAttribute`

Um den Einsatz von `SpecifyAttribute` vorzuführen, definiert folgender C#-Code das parameterlose Attribut `value` für den Knotentypen `Addition`:

```
Racr.SpecifyAttribute(spec, "Eval", "Addition", "*", true, (n) => {
    return (int) Racr.AttValue(Racr.AstChild(n, 1), "value")
         + (int) Racr.AttValue(Racr.AstChild(n, 2), "value");
});
```

Dies entspricht folgender Attributspezifikation in Scheme:

```
(with-specification spec
  (ag-rule
    value
    (Addition
      (lambda (n)
        (+ (att-value 'value (ast-child 1 n))
           (att-value 'value (ast-child 2 n)))))))
```

3.4.2. Evaluation der Schnittstelle

Die resultierende Schnittstelle deckt den vollständigen Funktionsumfang RACRs ab (**A1**). Sprachspezifikationen können zur Programmlaufzeit dynamisch definiert werden (**A2**). Obschon die Schnittstelle sehr dem Original ähnelt (**A3**), wird der Nutzer zu keiner Zeit mit konkreten Scheme-Typen konfrontiert (**A4**).

Die Schnittstelle hat jedoch zwei entscheidende Schwächen: Für Sprachspezifikationen und AST-Knoten fehlen konkrete Typen. Die entsprechenden RACR-spezifischen Objekttypen, auf denen alle RACR-Prozeduren arbeiten, sind Scheme-Records mit den Namen `ast-specification` beziehungsweise `node`. Diese Datentypen werden erst zur Programmlaufzeit generiert. IronScheme kann für Instanzen solcher Typen keinen präziseren Subtypen als `object` geben. Effektiv handelt es sich hierbei um opake Handler, ähnliche den `void`–Zeigern in C. Deshalb sind Spezifikationen sowie AST-Knoten in der Schnittstelle stets als `object` typisiert. Konsequenterweise ist aus der Signatur einer RACR-Methode allein nicht klar ersichtlich, welcher tatsächliche Objekttyp erwartet wird, was eine Quelle von Typfehlern darstellt, die während der Übersetzung nicht erkannt werden können.

Eine weitere Schwachstelle besteht in der Benutzerfreundlichkeit der Schnittstelle, die dem Nutzer einen prozeduralen Programmierstil aufdrängt. Objektorientierung, die Grundcharakteristik von C#, wird nicht wirkungsvoll eingesetzt.

Aus den vorgestellten Nachteilen ergeben sich die folgenden zusätzlichen Anforderungen an die bereits in Kapitel 3.3 präsentierten:

A5 – Entkoppelung von der RACR-Implementierung: Der Nutzer soll nicht direkt mit den Handlern für `ast-specification` und `node` arbeiten. Opake RACR-Datentypen sollen in Stellvertreter-Objekten gekapselt werden.

A6 – Objektorientierung: Alle wesentlichen Prozeduren von RACR können entweder Spezifikationen oder AST-Knoten zugeordnet werden. Die C#-Schnittstelle soll diese Prozeduren als Methoden der entsprechenden Objekte bereitstellen.

4. RACR-NET Implementierung: Objektorientierte Schnittstelle

Dieses Kapitel beschäftigt sich mit der Verwirklichung einer objektorientierten C#-Schnittstelle für RACR: RACR-NET[1]. Dabei sollen die in Kapitel 3.4.2 für verbesserungswürdig befundenen Probleme der prozeduralen Schnittstelle gelöst werden. Erreicht wird dies durch Refactoring der prozeduralen Schnittstelle. Die Idee hierbei ist, dass dessen Funktionalitäten entweder für RACR-Spezifikationen oder AST-Knoten definiert sind, sich also in zwei entsprechende Klassen partitionieren lassen (`RACR.Specification` und `RACR.AstNode`). Jede statische Methode der prozeduralen Schnittstelle (außer dem statischen `Racr`-Konstruktor) wird in eine Objektmethode einer der beiden Klassen umgeformt.

Die folgenden Unterkapitel stellen zunächst die resultierende objektorientierte Schnittstelle vor und demonstrieren deren Verwendung anhand eines Beispiels. Danach wird auf Details der Implementierung eingegangen. Eine Evaluation der Schnittstelle erfolgt in Kapitel 6.

4.1. Überblick über die objektorientierte Schittstelle

Im Folgenden wird lediglich die Schnittstelle für Sprachspezifikationen und AST-Knoten eingeführt. Auf konkrete Details ihrer Implementierung wird in Kapitel 4.4 eingegangen.

4.1.1. Spezifikationsschnittstelle

`RACR.Specification` ist die Hüllklasse für Sprach-Spezifikationen. Prozeduren, die auf Spezifikationen arbeiten, muss als erstes Argument stets eine `ast-specification` übergeben werden. Diese Prozeduren sollen nun über entsprechende Methoden der Klasse `Racr.Specification` aufgerufen werden. Deren Name und Signatur zeigt Quelltext 4.1. Die Klasse umfasst Methoden zur Spezifikation eines AST-Schemas (`AstRule`) und zur Attribuierung (`SpecifyAttribute`) sowie Fabrikmethoden zur Erzeugung von AST-Knoten (`CreateAst`, `CreateAstList` und `CreateAstBud`).

```
1 interface ISpecification {
2     public void AstRule(string rule);
3     public void CompileAstSpecifications(string start);
4     public void CompileAgSpecifications();
5     public void SpecifyAttribute(string name, string nonTerm,
```

[1] Die Quellen von RACR-NET sind bereits in denen von RACR aufgenommen und stehen unter MIT-Lizenz zur Verfügung.

```
6                           string context, bool cached, Delegate eq);
7       public void SpecifyAttribute<R>(string name, string nonTerm,
8                           string context, bool cached, Func<AstNode,R> eq);
9       public void SpecifyAttribute<R,T>(string name, string nonTerm,
10                          string context, bool cached, Func<AstNode,R,T> eq);
11      public AstNode CreateAst(string nonTerm, params object[] children);
12      public AstNode CreateAstList(params object[] children);
13      public AstNode CreateAstBud();
14  }
```

Listing 4.1: Schnittstelle der Klasse `Racr.Specification`

Der entscheidende Unterschied zur prozeduralen Schnittstelle besteht darin, dass der erste Parameter einer Attributgleichungsfunktion mit dem konkreten Typ `Racr.AstNode` versehen ist (Zeilen 8 und 10). Zugriffe auf dem `node`-Record sollen auch innerhalb von Attributgleichungen über die Klasse `Racr.AstNode` geschehen. Auf diese Weise soll die Verwendung des vieldeutigen Typs `object` weitestgehend eingeschränkt werden. Allgemein ist der Verzicht von `object` für Kind-Knoten jedoch nicht möglich, da Terminale beliebigen Typs sein können, weswegen die Methoden `CreateAst` und `CreateAstList` genau wie in der prozeduralen Schnittstelle ein `object`-Array als Argument für die Kind-Knoten erwarten.

4.1.2. Schnittstelle zum Abfragen von AST-Information

Mittels der Klasse `RACR.AstNode` soll dem Nutzer ein objektorientierter Zugriff auf AST-Knoten geboten werden. Quelltext 4.2 zeigt die Schnittstelle der Klasse. Methoden zur Graphersetzung und zum Setzen und Lesen von Annotationen wurden der Kürze halber ausgelassen.

```
1  interface IAstNode {
2      public AstNode Parent();
3      public AstNode Child(int index);        // Nichtterminal
4      public AstNode Child(string name);      // Nichtterminal
5      public AstNode Sibling(int index);      // Nichtterminal
6      public AstNode Sibling(string name);    // Nichtterminal
7      public T Child<T>(int index);           // Terminal
8      public T Child<T>(string name);         // Terminal
9      public T Sibling<T>(int index);         // Terminal
10     public T Sibling<T>(string name);       // Terminal
11     public bool IsNode();
12     public bool HasParent();
13     public int ChildIndex();
14     public bool HasChild(string name);
15     public int NumChildren();
16     public bool HasSibling(string name);
17     public string NodeType();
18     public bool IsListNode();
19     public bool IsBudNode();
```

```
20      public virtual object[] Children(params Range[] bounds);
21      public virtual void ForEachChild(Action<int,object> f,
22                                       params Range[] bounds);
23      public virtual object FindChild(Func<int,object,bool> f,
24                                      params Range[] bounds);
25      public virtual object FindChildA(Func<int,object,object> f,
26                                       params Range[] bounds);
27      public object AttValue(string attName, params object[] args);
28      public T AttValue<T>(string attName, params object[] args);
29      // ...
```

Listing 4.2: Schnittstelle der Klasse `Racr.AstNode`

In anderen Referenzattributgrammatik=Werkzeugen, wie JastAdd [11], wird für jeden Knotentyp einer Grammatik eine eigene Klasse mit maßgeschneiderten Methoden zum Zugriff auf Kind-Knoten und Attribute geboten. Anforderung **A2** in Kapitel 3.3 schließt den Einsatz derartiger Code-Generation jedoch aus, da hierfür die Grammatik schon zur Übersetzungszeit bekannt sein muss. Stattdessen wird für alle Knotentypen eine einheitliche, generische Schnittstelle bereitgestellt.

In Scheme werden Prozeduren oft mit Präfixen versehen, um auf die Objekt-Typen hinzudeuten, auf die sie angewendet werden. Außerdem werden auf diese Weise Namenskollisionen vermieden. In RACR ist dem Namen von Prozeduren, die zur Abfrage von AST-Information dienen, der Präfix `ast-` vorangestellt. Für die entsprechenden Methoden von `Racr.AstNode` wurde zur Verbesserung der Benutzerfreundlichkeit jedoch auf Präfixe verzichtet. Andernfalls wird Anforderung **A3** auch hier erfüllt.

Die Schnittstelle unterscheidet beim Zugriff auf Kind- und Geschwister-Knoten explizit zwischen Terminalen und Nichtterminalen. Die Methodengruppen `Child` und `Sibling` umfassen jeweils vier Methoden. Die nicht-generischen Ausführungen (Zeile 3 bis 6) beziehen sich auf Nichtterminale. Entsprechend ist der Rückgabewert jener Methoden vom Typ `Racr.AstNode`. Terminale können beliebigen Typs sein, weshalb die generischen Methoden (Zeilen 7 bis 10) einen Typ-Parameter erwarten, der den Typ des zu liefernden Terminals entsprechen muss. Somit wird die Verwendung des Typs `object` unterbunden. Die Typumwandlungen erfolgen innerhalb der Implementierung der Schnittstelle. Fehler bei der Angabe der Typ-Parameter führen dazu, dass währen der Programmlaufzeit Exceptions (`System.InvalidCastException`) ausgelöst werden.

Auch `AttValue` hat eine generische Umsetzung. Analog zu den Akzessoren für Terminale, spezifiziert der Typ-Parameter den Typ des Attributwertes.

Die vorgestellten Typ-parametrisierten Methoden können im Zusammenwirken mit sprachspezifischen zusätzlichen Hilfsfunktionen gekapselt werden, um den Zugriff auf Attribute sowie Kind- und Geschwister-Knoten zu vereinfachen und zu verkürzen. Außerdem wird dadurch eine konsistente Typisierung erreicht. Das folgende Kapitel demonstriert diesen Ansatz anhand eines Beispiels.

4.2. Anwendungsbeispiel

Die gewünschte Arbeitsweise der C#-Schnittstelle für RACR soll anhand eines Beispiels veranschaulicht werden. Quelltext 4.3 definiert die Grammatik einer Sprache für einfache arithmetische Ausdrücke. Sie beinhaltet eine Liste von Konstanten-Definitionen (`"Def"`) und einen beliebig tiefen AST, der den zu berechnenden Ausdruck (`"Exp"`) repräsentiert. Ausdrücke können Zahlen (`"Number"`), Konstanten (`"Const"`) sowie Additionen (`"AddExp"`) und Multiplikation (`"MulExp"`) weiterer Ausdrücke sein.

```
1  var spec = new Racr.Specification();
2  spec.AstRule("Root->Def*<Defs-Exp");
3  spec.AstRule("Def->name-value");
4  spec.AstRule("Exp->");
5  spec.AstRule("BinExp:Exp->Exp<A-Exp<B");
6  spec.AstRule("AddExp:BinExp->");
7  spec.AstRule("MulExp:BinExp->");
8  spec.AstRule("Number:Exp->value");
9  spec.AstRule("Const:Exp->name");
10 spec.CompileAstSpecifications("Root");
```

Listing 4.3: Spezifikation von AST-Regeln

Zuerst wird die Klasse `Racr.Specification` instanziiert. Auf dem erzeugten Objekt werden anschließend via `AstRule` die AST-Regeln der Sprache definiert. Die Spezifikationsphase des AST-Schemas wird durch einen Aufruf von `CompileAstSpecifications` abgeschlossen.

Diese Sprache soll um Attribute zur Berechnung von Ausdrücken ergänzt werden. Für gewöhnlich wird innerhalb einer Attributsfunktion auf AST-Knoten relativ zu dem Knoten, dessen Attribut ausgewertet werden soll, sowie auf weitere Attribute zugegriffen. Um diese Zugriffe zu vereinfachen, werden in RACR-Anwendungen typischerweise sprachspezifische Akzessor-Prozeduren definiert, welche statt den von RACR bereitgestellten generischen Methoden genutzt werden können. Der Name solcher sprachspezifischen Akzessor-Prozeduren spiegelt das abzufragende Kind beziehungsweise Attribut wider. Die explizite Angabe eines Kind- oder Attributnamens, wie von den generischen Akzessor-Prozeduren RACRs benötigt, entfällt somit, was die Lesbarkeit, Wartbarkeit und Prägnanz von Sprachspezifikationen erhöht. In C# erfüllen solche Hilfsfunktionen den zusätzlichen Zweck, die dynamischen Typumwandlungen bei Zugriffen auf Terminale oder Attribute auszulagern.

```
1  static class Accessors {
2      // Kind-Knoten
3      public static Racr.AstNode GetExp(this Racr.AstNode n) {
4          return n.Child("Exp");
5      }
6      public static Racr.AstNode GetDefs(this Racr.AstNode n) {
7          return n.Child("Defs");
8      }
9      public static Racr.AstNode GetA(this Racr.AstNode n) {
10         return n.Child("A");
```

```
11      }
12      public static Racr.AstNode GetB(this Racr.AstNode n) {
13          return n.Child("B");
14      }
15      public static double GetValue(this Racr.AstNode n) {
16          return n.Child<double>("value");
17      }
18      public static string GetName(this Racr.AstNode n) {
19          return n.Child<string>("name");
20      }
21      // Attribute
22      public static double Eval(this Racr.AstNode n) {
23          return n.AttValue<double>("Eval");
24      }
25      public static Racr.AstNode Lookup(this Racr.AstNode n, string name) {
26          return n.AttValue<Racr.AstNode>("Lookup", name);
27      }
28  }
```

Listing 4.4: Hilfsmethoden für den Zugriff auf Kind-Knoten und Attribute

Quelltext 4.4 definiert Extension-Methoden, die für die obige Beispielsprache als Akzessoren agieren. Die Typen der Terminale `"value"` und `"name"` sind explizit als `double` (Zeile 15) beziehungsweise `string` (Zeile 18) angegeben. Logischerweise müssen die gleichen Datentypen bei der Attributsspezifikation und AST-Konstruktion zum Einsatz kommen.

Die Methoden `Eval` und `Lookup` dienen zum Zugriff auf gleichnamige Attribute. Das Attribut `"Eval"` berechnet den Wert eines Ausdrucks. `"Lookup"` ist ein parametrisiertes Referenzattribut, das den zugehörigen Definitionsknoten einer Konstanten liefert.

```
1   spec.SpecifyAttribute("Eval", "Root", "*", true, (n) =>
2       n.GetExp().Eval());
3
4   spec.SpecifyAttribute("Eval", "AddExp", "*", true, (n) =>
5       n.GetA().Eval() + n.GetB().Eval());
6
7   spec.SpecifyAttribute("Eval", "MulExp", "*", true, (n) =>
8       n.GetA().Eval() * n.GetB().Eval());
9
10  spec.SpecifyAttribute("Eval", "Number", "*", true, (n) =>
11      n.GetValue());
12
13  spec.SpecifyAttribute("Eval", "Const", "*", true, (n) =>
14      n.Lookup(n.GetName()).GetValue());
15
16  spec.SpecifyAttribute("Lookup", "Root", "*", true,
17                        (Racr.AstNode n, string name) =>
18      (Racr.AstNode) n.GetDefs().FindChild((i, d) =>
19          ((Racr.AstNode) d).GetName() == name));
20
```

```
21 spec.CompileAgSpecifications();
```

Listing 4.5: Attributsspezifikationen

Quelltext 4.5 zeigt die Spezifikation der genannten Attribute unter Verwendung der Akzessoren aus Quelltext 4.4. Das Attribut `"Eval"` wird für die Nichtterminale `"Root"`, `"AddExp"`, `"MulExp"`, `"Number"` und `"Const"` definiert. Interessant ist die Verwendung von `Lookup` auf dem `"Const"`-Knoten selbst (Zeile 14), obwohl das Attribut `"Lookup"` nur für den `"Root"`-Knoten definiert wird. RACR implementiert Attribut-Broadcasting: Wenn ein Attribut für den Typ eines AST-Knotes, auf dem es aufgerufen wird, nicht definiert ist, wird der Attributsaufruf an den Eltern-Knoten weitergeleitet. Die Typsignatur der Attributsfunktion von `"Lookup"` entspricht der Signatur der entsprechenden Akzessor-Methode. Der zusätzliche Parameter hält den Namen der Konstanten. `FindChild` durchwandert linear die Liste von `"Def"`-Knoten und terminiert bei Namensübereinstimmung mit dem gefundenen Knoten. Bei erfolgloser Suche gibt die Methode `null` zurück.

```
1  var spec = new MySpec();
2  var defs = spec.CreateAstList(
3                  spec.CreateAst("Def", "e", 2.718),
4                  spec.CreateAst("Def", "pi", 3.142));
5  var exp = spec.CreateAst("AddExp",
6                 spec.CreateAst("Number", 5.0),
7                 spec.CreateAst("MulExp",
8                      spec.CreateAst("Const", "pi"),
9                      spec.CreateAst("Number", 2.0)));
10 var root = spec.CreateAst("Root", defs, exp);
11 Console.WriteLine("Eval: {0}", root.Eval());
```

Listing 4.6: AST-Konstruktion

In Quelltext 4.6 wird ein AST der Sprache unter Verwendung der Fabrikmethoden von `Racr.Specification` konstruiert. Der Ausdruck entspricht dabei $5+(\pi\times 2)$. Das Ergebnis des Ausdrucks wird mittels `Eval` auf dem Wurzelknoten berechnet und anschließend ausgegeben (Zeile 11).

4.3. Herausforderungen bei der Implementierung

Um das vorgestellte Verhalten der Schnittstelle zu realisieren, sind diverse programmiertechnische Hürden zu überwinden, die in diesem Kapitel nähergebracht werden sollen. Wichtig ist hierbei, dass die inkrementelle Auswertungssemantik RACRs für C#-Nutzer erhalten bleiben soll. Statt RACR in C# nachzubauen, soll deshalb vielmehr die existierende Implementierung genutzt werden. Das impliziert, dass die RACR-Scheme-Implementierung alle AST- und Attributdaten hält und Zugriffe auf diese protokolliert, um einen dynamischen Abhängigkeitsgraphen zur inkrementellen Auswertung aufzubauen. Jeder Zugriff von C# auf AST-

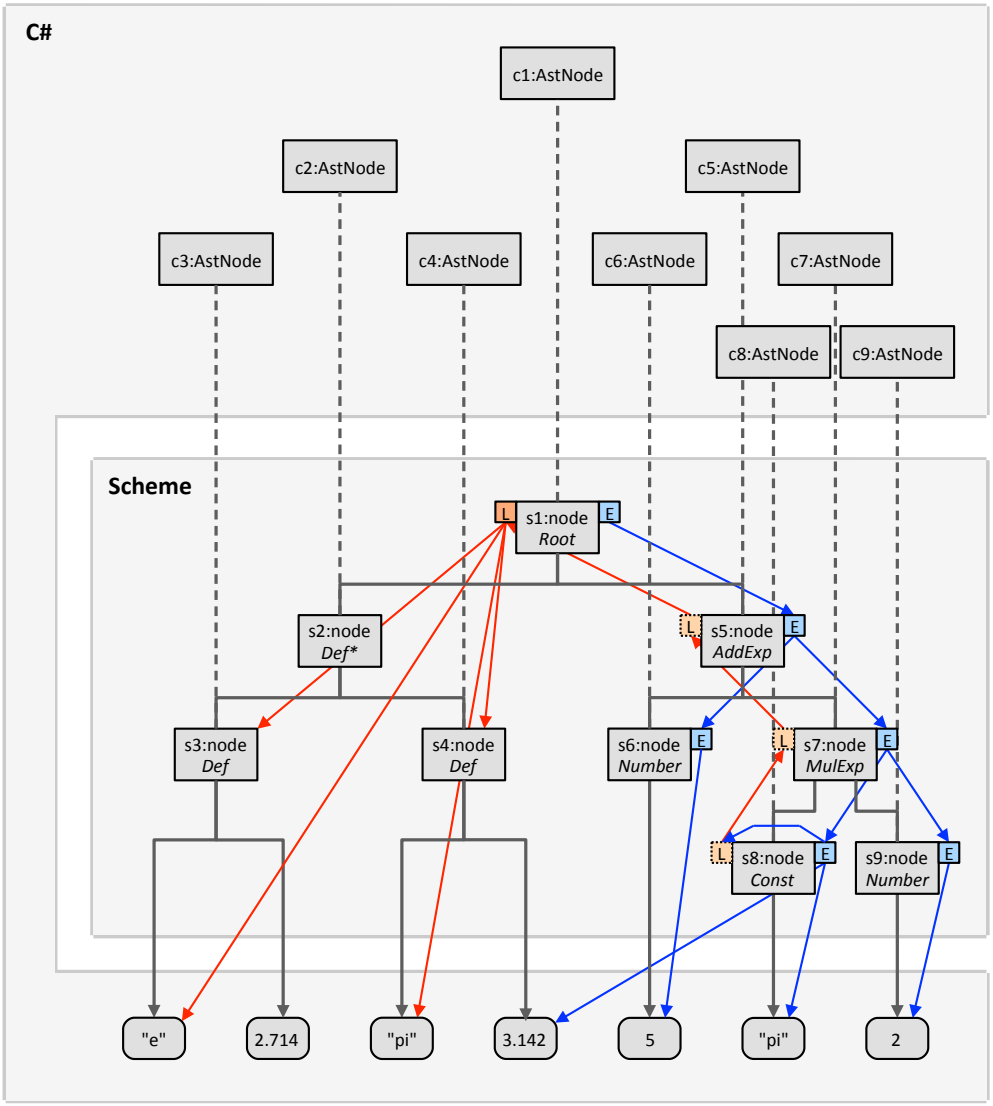

Abbildung 4.1.: Beziehungen zwischen Adapter-Objekten und `node`-Records im AST

und Attributdaten muss entsprechend auf RACR abgebildet werden. Andererseits sollen C# Nutzer mit der Scheme-Implementierung nicht in Berührung kommen, das heißt, nur mit nativen C#-Objekten arbeiten.

Um die Konsequenzen dieser Anforderungen bezüglich der Implementierung der Schnittstelle zu erläutern, soll auf das Anwendungsbeispiel aus Kapitel 4.2 Bezug genommen werden. Abbildung 4.1 stellt die dort angelegten Objekte des ASTs und deren Beziehungen zueinander dar. Die schwarzen, durchgezogenen Linien und Pfeile repräsentieren die von RACR bei der AST-Konstruktion erzeugen Assoziationen der `node`-Records zueinander beziehungsweise zu Terminalen. Die bijektiven Assoziationen zwischen einer `Racr.AstNode`-Instanz und dem zugehörigen `node`-Record sind als gestrichelte Linien dargestellt. Die kleinen Quadrate reprä-

sentieren die Attribute `'Eval` (E) und `'Lookup` (L). Die blauen beziehungsweise roten Pfeile kennzeichnen deren Abhängigkeiten auf weitere Attribute und Kind-Knoten. Der resultierende Graph wird von RACR zur inkrementellen Auswertung aufgebaut.

Entscheidend ist, dass `Racr.AstNode`-Instanzen keine Aggregationen zueinander besitzen. Sie fungieren lediglich als Proxy für die `node`-Objekte. Die eigentlichen Daten werden von RACR in Scheme gehalten. Jede AST- und Attributabfrage erfordert deshalb mehrfache Kontextwechsel zwischen C# und Scheme. Während der Attributsauswertung registriert RACR diese Abfragen und baut zur Realisierung der inkrementellen Auswertung einen dynamischen Abhängigkeitsgraphen für das jeweilige Attribut auf.

Abbildung 4.2 veranschaulicht die zeitliche Abfolge von Methodenaufrufen der objektorientierten Schnittstelle (links) und parallel dazu die Aufrufe der Scheme-Prozeduren von RACR (rechts), die bei der Auswertung des Attributs `"Eval"` auf dem Wurzelknoten ausgelöst werden. Rote Pfeile repräsentieren die Kontextwechsel bei Funktionsaufrufen. Für terminierende Funktionen sind die Kontextwechsel als blaue Pfeile dargestellt. Außerdem kennzeichnen Pfeile den Datenfluss von AST-Knoten, wobei zwischen Nichtterminalen und Terminalen unterschieden wird. Für die Fälle, wo Nichtterminale übergeben werden, sind die Pfeile mit dem Namen der `Racr.AstNode`-Instanz und des `node`-Records aus Abbildung 4.1 beschriftet.

Der Aufruf von `AttValue` auf der `Racr.AstNode`-Instanz `c1` soll an RACR weitergeleitet werden, indem für den zugehörigen `node`-Record die Prozedur `att-value` aufgerufen wird. Da die Attributcaches noch leer sind, muss die Attributgleichung für das Attribut `'Eval` und dem Knotentyp `'Root` ausgewertet werden. RACR ruft die zuvor via `specify-attribute` angegebene Prozedur mit `s1` als Argument auf. Diese Prozedur ist ein `Callable`-Objekt, in welchem das `Delegate`-Objekt von Quelltext 4.5 (Zeile 2) gekapselt werden muss. Der Aufruf von `Call` mit einem `node`-Record als Argument muss in dem Aufruf des Delegaten mit der entsprechenden `Racr.AstNode`-Instanz als Argument resultieren. Innerhalb des Delegaten wird auf `c1` die Methode `Child` aufgerufen, wo für den zugehörigen `node`-Record die Prozedur `ast-child` aufgerufen wird. Diese gibt den Record `s5` zurück. Der ursprüngliche `Child`-Aufruf muss diesen Record auf seine entsprechende `Racr.AstNode`-Instanz `c5` abbilden. Es ist nicht hinreichend, wenn `Child` einfach den `node`-Record zurückgäbe, da dies Anforderung **A5** verletzten würde. Die Wechsel zwischen C# und Scheme setzt sich fort, bis der Ausdruck schließlich berechnet ist.

Es geht hervor, dass ein ständiger Kontextwechsel zwischen C# und Scheme besteht. Zur Realisierung der Kontextwechsel von C# zu Scheme bedarf es einer Abbildung von `Racr.AstNode`-Instanz auf `node`-Record. Das gleiche gilt entsprechend umgekehrt für Kontextwechsel von Scheme zu C#. Methoden, die diese Abbildungen ausführen, müssen vorzugsweise leichtgewichtige Implementierungen haben, da sie notwendigerweise häufig aufgerufen werden. Im Wesentlichen ergeben sich also die drei folgenden Probleme:

Abbildung zwischen `Racr.AstNode` und `node`-Record: Es muss die Möglichkeit geschaffen werden, zwischen der Scheme- und der C#-Identität eines AST-Knotens zu wechseln. Diese Abbildung muss in allen Methoden, die auf `Racr.AstNode`-Instanzen arbeiten, zum Tragen kommen.

Unterscheidung bei Kind-Knoten zwischen Terminalen und Nichtterminalen: In eini-

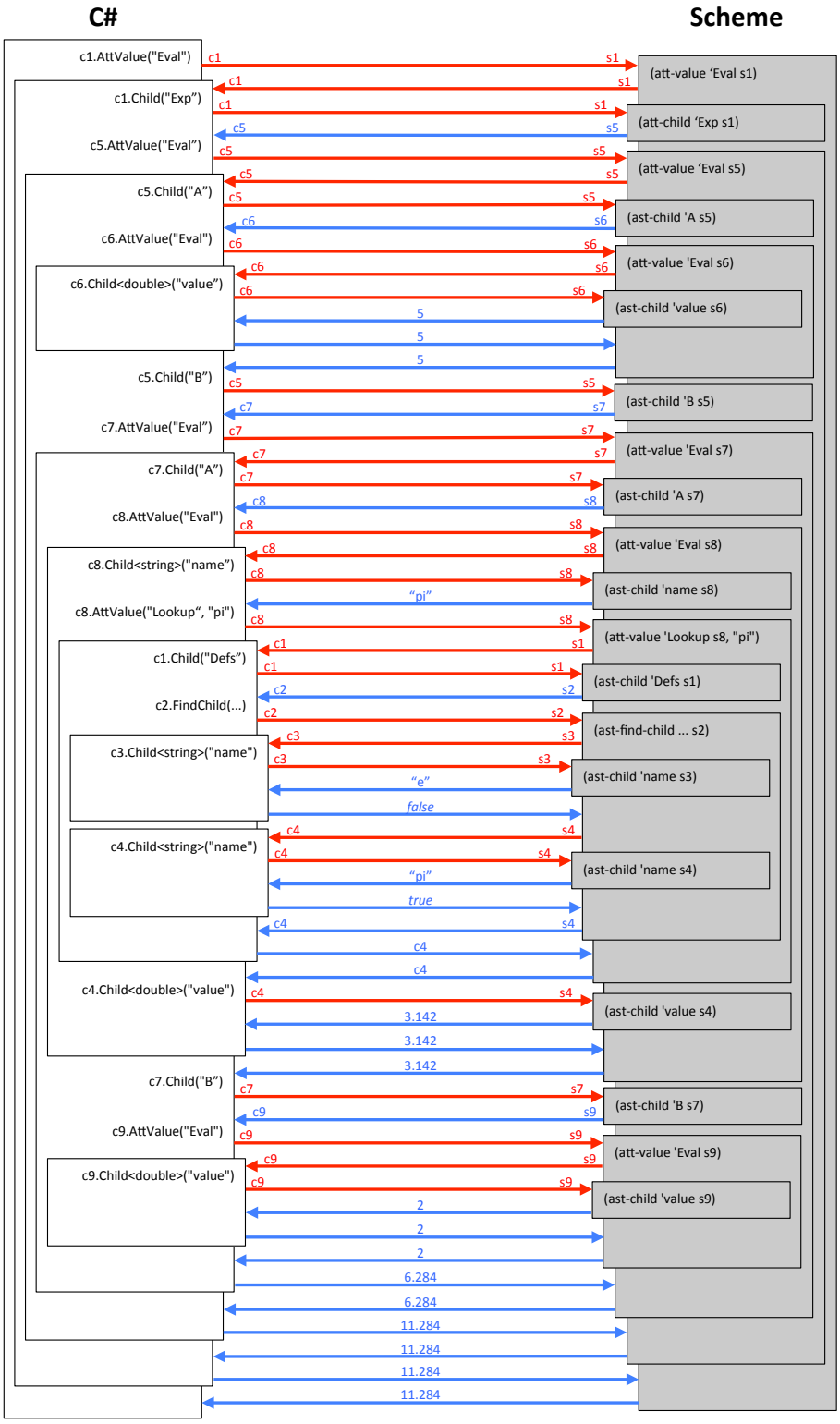

Abbildung 4.2.: Zeitliche Abfolge von API-Aufrufen

gen Methoden, die uniform auf Terminalen wie Nichtterminalen angewendet werden, darf die besagte Abbildung nur bedingt eingesetzt werden, da in RACR Terminale als ungekapselte Wert-Objekte gehandhabt werden.

Wrapping der Attributsfunktionen: Der erste Parameter einer Attributsfunktion ist der AST-Knoten, dessen Attribut ausgewertet werden soll. Aus der Sicht des C#-Nutzers handelt es sich um eine `Racr.AstNode`-Instanz. RACR übergibt zur Attributauswertung jedoch keine `Racr.AstNode`-Instanz, sondern den Handler des `node`-Records an das `Callable`-Objekt. Die Delegaten der Attributsgleichung müssen daher in einer Methode gekapselt werden, welche den `node`-Record auf dessen entsprechende `Racr.AstNode`-Instanz abbildet.

Die Lösung dieser Probleme ist Gegenstand des folgenden Kapitels.

4.4. Implementierung

In diesem Kapitel werden die Lösungen der in Kapitel 4.3 angeführten Probleme der Implementierung präsentiert.

4.4.1. Brücke zwischen C#- und Scheme-Objekten

Instanzen von `Racr.Specification` und `Racr.AstNode` müssen notwendigerweise den Handler des jeweiligen Scheme-Records halten, um bei Methodenaufrufen diesen an die zugehörige Scheme-Prozedur weiterleiten zu können. Im Konstruktor von `Racr.Specification` wird via `createSpecification.Call` ein neuer `ast-spezifikation`-Record angelegt, dessen Referenz in einem privaten Feld der Klasse, `internal object handler`, gespeichert wird. Alle weiteren Methoden leiten analog zum prozeduralen Ansatz die erhaltenen Daten an Aufrufe der zugehörigen `Callable`-Objekte weiter, jedoch unter der Benutzung von `handler` als erstes Argument.

Gleichsam verhält es sich mit der Klasse `Racr.AstNode`. Die Abbildung von einer `Racr.AstNode`-Instanz auf deren zugehörigen `node`-Record ist schlicht mit dem Zugriff auf `handler` realisiert. Wie der Abbildung 4.2 zu entnehmen, ist jedoch eine unidirektionale Abbildung von `Racr.AstNode`-Instanzen auf `node`-Records nicht ausreichend. Eine Abbildung von `node`-Record auf `Racr.AstNode`-Instanz ist außerdem von Nöten. Dazu wird der Record um ein Feld ergänzt, das eine Referenz auf die Instanz hält.

```
1 (define-record-type node
2   (fields
3    (mutable evaluator-state)
4    (mutable ast-rule)
5    (mutable parent)
6    (mutable children)
7    (mutable attributes)
8    (mutable cache-influences)
9    (mutable annotations)
```

```
10     (mutable csh-instance))      ; das neue Feld
11    (opaque #t)(sealed #t)
12    (protocol
13     (lambda (new)
14       (lambda (ast-rule parent children)
15         (new #f ast-rule parent children
16              (list) (list) (list)
17              #f)))))             ; und dessen Initialwert
```

Listing 4.7: Angepasste Definition des `node`-Records

Quelltext 4.7 zeigt die angepasste Definition des `node`-Records in `racr/core.sls`. Auf Zeile 10 wird der Record um das Feld `csh-instance` erweitert. Im Record-Konstruktor wird dieses vorerst auf `#f` initialisiert. Des Weiteren müssen die Prozeduren zum Setzen (`node-csh-instance-set!`) und Lesen (`node-csh-instance`) des Felds aus der Scheme-Bibliothek exportiert werden und im statischen Konstruktor von `Racr` entsprechende `Callable`-Objekte initialisiert werden.

Nach der Erzeugung des `node`-Records im Konstruktor von `Racr.AstNode` muss die `this`-Referenz in `csh-instance` gespeichert werden. Die Hilfsfunktion `GetNode` extrahiert die Referenz von dem Handler.

```
1 private static AstNode GetNode(object ast) {
2     return nodeCshInstance.Call(ast) as AstNode;
3 }
```

Mittels `GetNode` und dem Zugriff auf `handler` ist somit eine bidirektionale Abbildung zwischen `node`-Record und `Racr.AstNode` geschaffen.

4.4.2. Berücksichtigung von Terminalen

Aufgrund der dynamischen Typisierung Schemes bietet RACR dem Nutzer eine generische Schnittstelle zum Zugriff auf Kinder- und Geschwister-Knoten – sowohl Terminale als auch Nichtterminale. In Kapitel 4.1.2 wurde gezeigt, dass im Gegensatz dazu der Klasse `Racr.AstNode` für den Zugriff auf Kind- und Geschwister-Knoten für Terminale und Nichtterminale jeweils unterschiedliche Methoden bereitgestellt werden: Methoden für den Zugriff auf Terminale erwarten einen zusätzlichen Typ-Parameter. Aus der Typsignatur einiger Methoden ist jedoch nicht ablesbar, ob es sich bei den übergebenen oder zu liefernden Knoten um Terminale oder Nichtterminale handelt. Nutzer müssen sich an die Grammatik der Spezifikation halten und die korrekten Akzessoren, mit gegebenenfalls passenden Typ-Parameter, nutzen.

Auch in der Implementierung der Schnittstelle muss innerhalb dieser Methoden bei Kind-Knoten zwischen Nichtterminalen und Terminalen unterschieden werden können, um entsprechend die Abbildungsfunktionen einzusetzen beziehungsweise davon abzusehen, da die Abbildung nur bei Nichtterminalen zum Tragen kommen darf. Dies betrifft folgende Methoden von `Racr.AstNode`: den Konstruktor, `Children`, `ForEachChild`, `FindChild`, `FindChildA`,

`RewriteRefine` und `RewriteAbstract`. Diese Methoden realisieren die RACR-Prozeduren `ast-children`, `ast-for-each-child`, `ast-find-child`, `ast-find-child*`, `rewrite-refine` und `rewrite-abstract`. RACR bietet eine Reihe undokumentierter Prozeduren zur Abfrage von Meta-Information bezüglich der AST-Schemas und Attribuierung für Spezifikationen eines ASTs. Folgende sind für den vorgestellten Zweck erforderlich:

- `(specification->find-ast-rule spec non-term)` liefert die einem Nichtterminal zugehörige AST-Regel.
- `(ast-rule->production rule)` liefert eine Liste mit den Produktionssymbolen einer AST-Regel.
- `(symbol->non-terminal? symbol)` bestimmt, ob ein Symbol ein Nichtterminal ist.

Die Objekte für AST-Regeln und Symbole sind RACR-interne Scheme-Records. Mittels dieser Prozeduren wird im Konstruktor von `Racr.AstNode` für jeden Kind-Knoten bestimmt, ob er ein Nichtterminal ist. Diese Information wird aus Gründen der Effizienz direkt in der Klassen-Instanz gespeichert, um gegebenenfalls wiederverwendet zu werden.

```
1  public class AstNode {
2      internal object handler;
3      private bool[] ntChilren;
4      public AstNode(Specification spec, string nonTerm,
5                     params object[] children)
6      {
7          var nt = SymbolTable.StringToObject(nonTerm);
8          var rule = specificationFindAstRule.Call(spec.handler, nt);
9          var symbols = astRuleProduction.Call(rule) as Cons;
10         ntChilren = new bool[children.Length];
11         Cons list = null;
12         Cons marker = null;
13         for (int i = 0; i < children.Length; i++) {
14             symbols = symbols.cdr as Cons;
15             ntChilren[i] = symbolIsNonTerminal.Call<bool>(symbols.car);
16             var child = ntChilren[i] ? (children[i] as AstNode).handler
17                                      : children[i];
18             var cons = new Cons(child);
19             if (list == null) list = marker = cons;
20             else {
21                 marker.cdr = cons;
22                 marker = cons;
23             }
24         }
25         handler = createAst.Call(spec.handler, nt, list);
26         nodeCshInstanceSet.Call(handler, this);
27     }
```

Listing 4.8: Initialisierungen im Konstruktor von `Racr.AstNode`

Quelltext 4.8 zeigt diesen Vorgang. Der Konstruktor enthält eine `for`-Schleife, die das übergebene Kind-Knoten-Array traversiert. Parallel wird die Scheme-Liste der Produktionssymbole durchwandert und die Information, ob ein Kind ein Nichtterminal ist, in `ntChildren` gespeichert (Zeile 15). Zusätzlich wird eine Scheme-Liste zur Weitergabe der Knote an `create-ast` konstruiert: Für Nichtterminal-Kinder wird aus der `Racr.AstNode`-Instanz der Handler auf den `node`-Record extrahiert (Zeile 16). Terminale werden direkt übergeben (Zeile 17). Interessant sind auch Zeilen 25 und 26, in denen `handler` beziehungsweise `csh-instance` gesetzt werden, wie in Kapitel 4.4.1 beschrieben.

In der Implementierung der übrigen, oben genannten Methoden, kann zur Unterscheidung von Terminalen und Nichtterminalen für den n-ten Kind-Knoten das entsprechende Element in `ntChildren` konsultiert werden.

4.4.3. Wrapping der Attributsfunktionen

Attributsfunktionen werden in Form von Delegaten an die Methode `SpecifyAttribute` der Klasse `Racr.Specification` übergeben. Quelltext 4.9 zeigt eine naive Implementierung, in welcher analog zur Implementierung der prozeduralen Schnittstelle auf dem `Delegate`-Objekt IronSchemes die Extension-Methode `ToSchemeProcedure` aufgerufen wird um ein `Callable`-Objekt zu erzeugen, das anschließend an den Aufruf der Scheme-Prozedur `specify-attribute` übergeben wird.

```
1  public void SpecifyAttribute(string name, string nonTerm, string context,
2                               bool cached, Delegate equation)
3  {
4      specifyAttribute.Call(
5          handler,
6          SymbolTable.StringToObject(name),
7          SymbolTable.StringToObject(nonTerm),
8          SymbolTable.StringToObject(context),
9          cached,
10         equation.ToSchemeProzedure(),
11         false);
12 }
```

Listing 4.9: Naive Implementierung von `SpecifyAttribute`

Wie in Kapitel 4.1.1 vorgestellt, soll der erste Parameter einer Attributsfunktion vom Typ `Racr.AstNode` sein. Der Attributauswerter RACRs (die Prozedur `att-value`) erwartet jedoch Attributsfunktionen, deren erstes Argument ein RACR `node`-Record ist — der AST Knoten mit welchem die auszuwertende Attributsinstanz assoziiert ist. Die Auswertung einer in C# definierten Attributsfunktionn löst demnach einen Laufzeitfehler aus, da innerhalb des `Callable` der Versuch, den `node`-Record des AST-Knoten in eine `Racr.AstNode`-Instanz umzuwandeln scheitert. Auf dem `Delegate`-Objekt, das die Attributsgeichung hält, kann `ToSchemeProcedure` zur Erzeugung eines `Callable`-Objekts nicht direkt angewandt werden. Die Delegaten müssen in einer weiteren Methode gekapselt werden, innerhalb welcher die

Abbildung von `node`-Record auf `Racr.AstNode`-Instanz zum Einsatz kommen muss. Analog zur prozeduralen Schnittstelle, müssen beliebig typisierte Rückgabewerte sowie etwaige weitere Parameter unterstützt werden (siehe Kapitel 3.4.1). Da die Typsignatur des Delegaten während der Übersetzung unbekannt ist, muss mithilfe von Reflexion eine passende Wrapper-Methode zur Laufzeit generiert werden.

Wegen der dynamischen Typisierung Schemes sind Parameter und Rückgabewert der Methode `Call` der `Callable`-Schnittstelle allesamt mit `object` typisiert. Für Delegaten, die von Scheme aus aufgerufen werden sollen, sich jedoch in der Typsignatur von `Call` unterscheiden, wird in der Methode `ToSchemeProcedure` eine Wrapper-Methode erzeugt, in welcher entsprechende Typumwandlungen ausgeführt werden. Für mit einem Werttypen typisierte Parameter des Delegaten muss eine Konvertierung von `object` zu diesem Werttypen realisiert werden (Unboxing). Ähnlich verhält es sich mit den Rückgabewerten, wobei gegebenenfalls von Werttyp nach `object` konvertiert wird (Boxing). Derartige Typumwandlungen sind rechentechnisch aufwendig. Ein doppeltes Methoden-Wrapping ist daher zu vermeiden. Es soll eine dynamische Methode erzeugt werden, in welcher einerseits die Abbildungsfunktion auf den AST-Knoten, also den ersten Parameter des Delegaten, angewendet wird und zusätzlich für alle übrigen Parameter und den Rückgabewert das Unboxing beziehungsweise Boxing ausgeführt wird, unter der Bedingung, dass es sich jeweils um einen Werttypen handeln.

C# umfasst eine Schnittstelle, um zur Laufzeit dynamische Methoden zu erzeugen. Damit in Verbindung bietet IronScheme die leichtgewichtige Fabrikmethode `Closure.Create`, welche ein `Delegate`-Objekt erwartet, dessen Parameter und Rückgabewert mit `object` typisiert sind, und ein `Callable` liefert. Dieses Objekt hält im Wesentlichen eine Referenz auf den Delegaten. Der entscheidende Unterschied von `Closure.Create` zu `ToSchemeProcedure` besteht darin, dass dabei kein Wrapping des `Delegate` (Unboxing und Boxing) ausgeführt werden muss, da die Signatur des Delegaten bereits in der notwendigen Form vorliegt.

```
1  static Callable WrapToCallable(this Delegate equation) {
2      MethodInfo method = equation.Method;
3      Type[] paramTypes = method.GetParameters()
4      .Select(p => p.ParameterType).ToArray();
5      if (paramTypes.Length == 0
6      || !typeof(AstNode).IsAssignableFrom(paramTypes[0])) {
7          throw new ArgumentException(
8                  "type of delegate's first argument must be AstNode.");
9      }
10
11     var dynmeth = new DynamicMethod("", typeof(object),
12                         paramTypesArray[paramTypes.Length], true);
13     var gen = dynmeth.GetILGenerator();
14
15     gen.Emit(OpCodes.Ldarg_0);
16     var getNodeInfo = ((Delegate)(Func<object, AstNode>)GetNode).Method;
17     gen.Emit(OpCodes.Call, getNodeInfo);
18
19     for (int i = 1; i < paramTypes.Length; i++) {
20         gen.Emit(OpCodes.Ldarg_S, i);
21         if (paramTypes[i].IsValueType) {
```

```
22              gen.Emit(OpCodes.Unbox_Any, paramTypes[i]);
23          }
24          else {
25              gen.Emit(OpCodes.Castclass, paramTypes[i]);
26          }
27      }
28      gen.Emit(OpCodes.Call, method);
29      if (method.ReturnType.IsValueType) {
30          gen.Emit(OpCodes.Box, method.ReturnType);
31      }
32      gen.Emit(OpCodes.Ret);
33
34      Type outType = callTargets[paramTypes.Length];
35      return Closure.Create(dynmeth.CreateDelegate(outType),
36                            paramTypes.Length);
37  }
```

Listing 4.10: Dynamische Methodengenerierung und Typzuordnung

Quelltext 4.10 zeigt die Implementierung der statischen Methode `WrapToCallable`, welche `ToSchemeProzedure` in Quelltext 4.9 (Zeile 10) ersetzen soll. Der wichtigste Teil der Methode umfasst die Zeilen 15 bis 32. Hier wird der IL-Code der dynamischen Methode festgelegt. Dabei werden Opcodes für das Laden von Argumenten, Methodenaufrufe, Unboxing und Boxing, sowie das Beenden der Methode eingesetzt. Folgende Schritte sollen in der zu erzeugenden Wrapper-Methode ausgeführt werden: Zuerst wird der erste Parameter (der AST-Knoten) geladen und anschließend `GetNode` aufgerufen (Zeile 15 bis 17). Als nächstes werden alle weiteren Parameter geladen (Zeile 20). Falls es sich bei deren Typ um Werttypen handelt, wird Unboxing ausgeführt (Zeile 22). Für Referenztypen muss stattdessen eine dynamische Typumwandlung erfolgen (Zeile 25). Schließlich wird das `Delegat`-Objekt aufgerufen, innerhalb welchem der Attributwert berechnet werden soll (Zeile 28). Wenn der Attributwert mit einem Werttyp typisiert ist, so wird Boxing angewendet (Zeile 29 bis 31). Damit terminiert die dynamische Methode (Zeile 32).

Nun wird aus dem `DynamicMethod`-Objekt mittels `CreateDelegate` ein `Delegat` erzeugt, das anschießend an `Closure.Create` übergeben wird (Zeile 32 und 33). Das resultierende `Callable` ist eine Attributsfunktion, welche einerseits in C# definiert wurde (und daher nur unter Beachtung der objektorientierten Schnittstelle von RACR-NET implementiert wurde) und andererseits vom Attributsauswerter der existierenden RACRSchemeImplementierung verarbeitet werden kann. Somit ist das generierte `Callable` eine korrekte C#-Attributsfunktion für `specify-attribute`.

5. Evaluation

Es wurde bereits an dem Anwendungsbeispiel in Kapitel 4.2 gezeigt, dass die an die RACR-NET Schnittstelle gesetzten funktionalen Anforderungen von der vorgestellten Implementierung erfüllt werden, insbesondere bezüglich einer benutzerfreundlichen objektorientierten Schnittstelle. Der Zugriff auf Spezifikation und AST-Knoten erfolgt ausschließlich über Instanzen von `Racr.Specification` beziehungsweise `Racr.AstNode` und alle Aufrufe von RACR sind in Methoden dieser Stellvertreter-Objekte gekapselt.

Im Folgenden soll gezeigt werden, dass RACR-NET nicht nur benutzerfreundlich, sondern auch korrekt und effizient ist.

5.1. Testen der Schnittstelle

Zum Testen von RACR-NET wurde eine umfangreiche, bereits existierende Anwendung[1], die den wesentlichen Funktionsumfang RACRs abdeckt, in C# reimplementiert. Das Beispiel ist eine Lösung des Language Workbench Challenge 2013 [12], dessen Aufgabe[2] darin bestand, eine domänenspezifische Sprache zu schaffen, mittels welcher interaktive Fragebögen zur Datenerfassung auf einfache Weise beschrieben und ausgewertet werden können. Die Anwendung bedient sich aller Mechanismen RACRs mit Ausnahme von komplexeren Graphersetzungen. Um auch die funktionelle Korrektheit der Graphersetzungsmethoden sicherzustellen, wurden diese in einem eigenen NUnit[3]-Test erfasst.

5.2. Performance-Messungen und -Vergleiche

Die Erzeugung von Adapter-Objekten und der indirekte Zugriff auf RACRs Funktionalitäten über jene Objekte erzeugt einen Laufzeit-Overhead. Dieser wurde für eine RACR-Anwendung ermittelt, die im Folgenden beschrieben wird.

Unter Verwendung der in Kapitel 4.2 gegebenen Sprachspezifikation wurden arithmetische Ausdrücke für verschiedene Konstantenbelegungen berechnet. Um die Ausführungszeiten des C#-Programms gegenüber denen des Scheme-Programms vergleichen zu können, wurde die Anwendung in beiden Sprachen, Scheme (unter der Verwendung der RACR Scheme-Bibliothek) und C# (mittels der objektorientierten Schnittstelle), implementiert. Der Laufzeit-Overhead ist die Differenz der Ausführungszeiten beider Implementierungen in IronScheme.

[1] `https://github.com/christoff-buerger/racr/tree/master/examples/questionnaires`
[2] `http://www.languageworkbenches.net/wp-content/uploads/2013/11/Ql.pdf`
[3] `http://www.nunit.org/`

Die den zu berechnenden Ausdruck repräsentierenden ASTs wurden mithilfe eines Python-Skripts generiert und enthalten jeweils 5.000, 10.000 und 20.000 Binär-Operationen und ebenso viele Blatt-Knoten. Die Hälfte der Blatt-Knoten sind Konstanten, wobei insgesamt 26 verschiedene Konstanten-Definitionen benutzt werden.

			Laufzeit in s		
Operationen	Evals	Rewrites	C#	IronScheme	Racket
5.000	1	0	1,43	1,36	0,45
10.000	1	0	2,93	2,92	0,98
20.000	1	0	6,21	6,03	2,17
5.000	1.000	1.000	72,44	72,20	24,31
10.000	1.000	1.000	150,14	145,81	71,08
20.000	1.000	1.000	330,58	316,97	217,79

Tabelle 5.1.: Performance-Messungen

Alle Läufe wurden auf einem Rechner mit einem Intel Core i5-3350P Vier-Kern-Prozessor und 16 GB RAM unter Windows 8.1 gemessen. Das verwendete IronScheme Release war 115404, 32-Bit, vom 29. Oktober 2015. Als .NET VM wurde das Microsoft .NET Framework 4.0 verwendet. Von zwanzig Messungen pro Lauf wurde die beste Zeit genommen. Tabelle 5.1 zeigt die Messergebnisse für eine einzelne Berechnung des Attributs `'Eval` (Zeile 1 bis 3). Ferner wurden innerhalb einer Schleife jeweils der Wert einer Konstanten in deren Definition modifiziert und anschließend `'Eval` für den Wurzel-Knoten ausgewertet (Zeile 4 bis 6).

Um die Performance von IronScheme gegenüber anderen Scheme-VMs abschätzen zu können, wurden die Tests ebenfalls auf der Racket[4] Scheme-VM durchgeführt (Racket Version 6.2). Zur Erfassung des durch die objektorientierte Schnittstelle generierten Overheads sind jedoch lediglich die für IronScheme gemessenen Ausführungszeiten relevant.

Die Messungen ergeben, dass die RACR-NET-Lösung erwartungsgemäß etwas langsamer ausführt als eine reine Scheme-Lösung. Der Performance-Overhead beträgt für 20.000 Knoten und 1.000 Auswertungen und Graphersetzungen circa 4,3 % und ist damit durchaus akzeptabel. Des Weiteren wird ersichtlich, dass IronScheme zwar langsamer ist als Racket, der Faktor jedoch keine Größenordnung beträgt (Im Gegensatz zu Racket ist IronScheme ein Ein-Man-Projekt, bei dem Performance nicht im Mittelpunkt steht).

Bei dem gewählten Beispiel handelt es sich um eine Referenzattributgrammatik, deren Attributsgleichungen auf simple Addition und Multiplikation zweier Fließkommazahlen beschränkt sind. Es bleibt daher zu untersuchen, wie stark der Geschwindigkeitsvorteil von C# gegenüber IronScheme in komplexeren Gleichungen zum Tragen kommt und, ob dieser den Overhead von RACR-NET wohl möglich kompensiert.

Die Messergebnisse unterstreichen die Vorteile der RAG-gesteuerten Graphersetzung. Eine einmalige Auswertung von `'Eval` ist relativ teuer (1,43 Sekunden mittels RACR-NET), weil anfangs alle Attribut-Caches leer sind und so `'Eval` erst für jeden Teilausdruck berechnet

[4] `http://racket-lang.org/`

werden muss. Da die vorgenommenen Graphersetzungen nur jeweils einen Anteil der zuvor berechneten Attributwerte invalidieren, begünstigt die inkrementelle Auswertung alle nachträglichen Berechnungen. Eintausend Berechnungen mit intermediären Graphersetzungen benötigen 330,58 Sekunden – nur circa 20 % der tausendfachen Dauer der initiale Berechnung.

6. Zusammenfassung und Ausblick

Abschließend wird der Beitrag dieser Arbeit noch einmal zusammengefasst und ein Ausblick auf offene Fragen und Erweiterungsmöglichkeiten gegeben.

6.1. Eine objektorientierte Bibliothek für RAG-gesteuerte Graphersetzung

In dieser Arbeit wurde die Umsetzung einer objektorientierten Schnittstelle von RACR, der Scheme-Bibliothek zur Referenzattributgrammatik-gesteuerten Graphersetzung, für die Software-Plattform .NET präsentiert. Diese Schnittstelle namens RACR-NET ermöglicht es, RACR-Anwendungen in C# zu programmieren.

IronScheme, eine Scheme-Implementierung für .NET, wurde als virtuelle Maschine eingesetzt, um innerhalb der Methoden der Schnittstelle die entsprechenden RACR-Prozeduren aufzurufen. Da sich IronScheme zur Ausführung von RACR-Anwendungen als nicht hinreichend R6RS-konform herausstellte, wurden geringfügige Veränderungen am Quellcode RACRs bezüglich der `hashtable`-Prozeduren vorgenommen, um Grenzfälle zu umgehen. Unter Verwendung der IronScheme-Klassenbibliothek wurde erst eine imperative Schnittstelle geschaffen. Darauf aufbauend entstand eine benutzerfreundliche, objektorientierte Schnittstelle, die den vollständigen Funktionsumfang RACRs über Stellvertreter-Objekte für Spezifikationen und AST-Knoten abbildet.

In der Schnittstelle kommen keine Scheme-spezifischen Datentypen zum Einsatz, sodass Nutzer RACR-NET ohne jegliche Scheme-Kenntnisse einsetzen können. Gleichzeitig bleibt die Nähe zu RACRs originalen Scheme-Schnittstelle bewahrt. Der indirekte Zugriff auf AST-Knoten erzeugt in RACR-NET-Anwendungen einen Performance-Overhead gegenüber äquivalenten, via IronScheme ausgeführten RACR-Anwendungen. Dieser wurde für eine Beispiel-Anwendung bestimmt und beträgt 4,3 %.

6.2. Zukünftige Arbeiten

RACR-NET dient dem Zweck, die RAG-gesteuerte Graphersetzung von dem gewählten Technikraum .NET aus nutzen zu können. Bisher wurde nur gezeigt, wie die Schnittstelle von C# aus eingesetzt wird. Es existiert jedoch eine Vielzahl von .NET-Sprachen, innerhalb welcher RACR-NET eingesetzt werden kann. Inwieweit dies für solche Sprachen effektiv umgesetzt werden kann, die neben Objektorientierung andere Programmierparadigmen verfolgen (zum Beispiel F# mit der funktionale Programmierung), muss noch untersucht werden.

Ein interessanter Anwendungsfall besteht darin, bereits bestehende in Scheme implementierte RACR-Anwendungen von C# aus über RACR-NET zu erweitern. Dies schließt die Möglichkeit ein, AST-Schemas um zusätzliche AST-Regeln zu ergänzen und weitere Attribute zu definieren. Um die Erweiterbarkeit einer via Scheme spezifizierten Sprache in C# zu ermöglichen und Kopplung und Vererbung von Attributgrammatiken über Sprachgrenzen hinaus zu realisieren, bedarf es weiterer Arbeit. Eine wichtige Aufgabe liegt dabei in der Sonderbehandlung des Falls, dass in Scheme definierte Referenzattribute in C# ausgewertet werden, da diese als AST-Knoten statt eines Stellvertreter-Objekts ein RACR-internes Objekt liefern. Der korrekte Umgang mit diesen Datentypen setzt Scheme-Kenntnisse voraus, die RACR-NET in den jetzigen Anwendungsszenarien nicht verlangt.

Anhänge

A. Literaturverzeichnis

[1] Pavel Avgustinov, Torbjörn Ekman und Julian Tibble. "Modularity first: a case for mixing AOP and attribute grammars". In: *Proceedings of the 7th international conference on Aspect-oriented software development*. ACM. 2008, S. 25–35.

[2] Christoff Bürger. *RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting*. Techn. Ber. Technische Universität Dresden, Institut für Software- und Multimediatechnik, Lehrstuhl Softwaretechnologie, 2012.

[3] Christoff Bürger. "fUML Activity Diagrams in RACR: A RACR Solution of The TTC 2015 Model Execution Case". In: Software Technologies: Applications und Foundations (STAF). 2015.

[4] Christoff Bürger. "Reference attribute grammar controlled graph rewriting: motivation and overview". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM. 2015, S. 89–100.

[5] Christoff Bürger u. a. "Using Reference Attribute Grammar-Controlled Rewriting for Energy Auto-Tuning". In: *10th International Workshop on Models@run.time*. CEUR Workshop Proceedings (CEUR-WS. org). 2015.

[6] Will Clinger u. a. *Scheme Steering Committee Position Statement (Draft)*. Abgerufen am 7. Juni 2015. Aug. 2009. URL: `http://scheme-reports.org/2009/position-statement.html`.

[7] Alan Demers, Thomas Reps und Tim Teitelbaum. "Incremental evaluation for attribute grammars with application to syntax-directed editors". In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 1981, S. 105–116.

[8] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. Microsoft, Juni 2012. URL: `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[9] Hartmut Ehrig. *Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools*. Bd. 2. World Scientific, 1999.

[10] Torbjörn Ekman. "Extensible compiler construction". Diss. 2006.

[11] Torbjörn Ekman und Görel Hedin. "The JastAdd system: modular extensible compiler construction". In: *Science of Computer Programming* 69.1–3 (2007), S. 14–26.

[12] Sebastian Erdweg u. a. "The State of the Art in Language Workbenches". In: *Software Language Engineering*. Hrsg. von Martin Erwig, Richard F. Paige und Eric Van Wyk. Bd. 8225. Lecture Notes in Computer Science. Springer, 2013, S. 197–217.

[13] Görel Hedin. "An object-oriented notation for attribute grammars". In: *ECOOP'89: Proceedings of the 1989 European Conference on Object-Oriented Programming*. 1989, S. 329–345.

[14] Görel Hedin. "Reference attributed grammars". In: *Informatica (Slovenia)* 24.3 (2000), S. 301–317.

[15] Roger Hoover und Tim Teitelbaum. "Efficient incremental evaluation of aggregate values in attribute grammars". In: *ACM SIGPLAN Notices* 21.7 (1986), S. 39–50.

[16] Donald E. Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), S. 127–145.

[17] Donald E. Knuth. "Semantics of context-free languages: Correction". In: *Theory of Computing Systems* 5.2 (1971), S. 95–96.

[18] Monica Lam u. a. *Compilers: Principles, techniques and tools*. Pearson Education, 2006.

[19] William H. Maddox III. "Incremental static semantic analysis". Diss. DTIC Document, 1997.

[20] Tanja Mayerhofer und Manuel Wimmer. "The TTC 2015 Model Execution Case". In: *8th Transformation Tool Contest, CEUR* (2015).

[21] Jukka Paakki. "Attribute grammar paradigms—a high-level methodology in language implementation". In: *ACM Computing Surveys (CSUR)* 27.2 (1995), S. 196–255.

[22] Grzegorz Rozenberg. *Handbook of Graph Grammars and Comp.* Bd. 1. World scientific, 1997.

[23] Roger S. Scowen. *Extended BNF-a generic base standard*. Techn. Ber. Technical report, ISO/IEC 14977, 1998.

[24] Alex Shinn, John Cowan und Artur A. Gleckler. *Revised^7 Report on the Algorithmic Language Scheme*. Techn. Ber. Juli 2013. URL: `http://trac.sacrideo.us/wg/raw-attachment/wiki/WikiStart/r7rs.pdf`.

[25] Emma Söderberg und Görel Hedin. *Incremental evaluation of reference attribute grammars using dynamic dependency tracking*. Techn. Ber. 2012.

[26] Michael Sperber u. a. "Revised^6 Report on the Algorithmic Language Scheme". In: *Journal of Functional Programming* 19.S1 (2009), S. 1–301.

[27] Julian R. Ullmann. "An algorithm for subgraph isomorphism". In: *Journal of the ACM (JACM)* 23.1 (1976), S. 31–42.

B. MIT Lizens