Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. / Harvard Univ. Erhard Plödereder
Betreuer:	Dr. Rainer Koschke Dipl.-Inf. Thomas Eisenbarth

Diplomarbeit Nr. 2182

Konzeption und Implementierung einer abstrakten Anfrage- und Manipulationssprache für den Resource-Flow-Graph

Michael Stürmer

Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstr. 38
D-70569 Stuttgart

begonnen am:	1. Februar 2004
beendet am:	2. August 2004
CR-Klassifikation:	D.2.3, D.2.6, D.2.7, D.3.4, F.3.1

Zusammenfassung

Der Resource-Flow-Graph (RFG) ist eine Zwischendarstellung, die im Bauhaus-Projekt eingesetzt wird. Sie enthält aus Quellcode extrahierte Quell- und Architekturinformationen. Die Auswertung dieser Informationen findet momentan entweder manuell oder durch ausprogrammierte Analysen statt.

Diese Arbeit beschreibt eine Skriptsprache, die von Entwicklern und Endanwendern des Bauhaus-Systems zur Formulierung von Analysen und Manipulationen des RFG benutzt werden kann. Zuerst werden Anforderungen an die Skriptsprache formuliert und vorhandene Anfragesprachen auf erwünschte und unerwünschte Eigenschaften untersucht. Es stellt sich heraus, dass eine leicht verständliche Syntax und eine komfortable Visualisierung durch den im Bauhaus-Projekt eingesetzten Grapheneditor wesentliche Merkmale der Sprache sein sollen. Die Konzeption der Sprache, die vorhandenen Datentypen, Anweisungen sowie eingebaute Funktionen werden beschrieben und der daraus entwickelte Entwurf mit Realisierungsalternativen dargestellt. Abschließend wird die Effizienz der Realisierung mit anderen Anfragesprachen verglichen.

Inhaltsverzeichnis

1. Einleitung

1.1. Inhalt der Arbeit

Hintergrund

Der Resource-Flow-Graph (RFG) ist eine Zwischendarstellung, die im Bauhaus-Projekt eingesetzt wird. Sie enthält unter anderem globale Deklarationen, Module und Verzeichnisse sowie deren (im Falle von Modulen und Verzeichnissen aggregierte) Abhängigkeiten, die direkt aus dem Quellcode extrahiert werden. Resource-Flow-Graphen können mit dem Grapheneditor Gravis visualisiert und manipuliert werden.

Um Informationen über den Resource-Flow-Graphen abzufragen stehen in Gravis verschiedene Filter-, Such- und Navigationsmöglichkeiten zur Verfügung. Komplexe und wiederkehrende Anfragen lassen sich jedoch nur umständlich damit erledigen.

Eine (bereits abgeschlossene) Studienarbeit hatte deshalb zum Ziel, eine Skriptsprache (Python) in Gravis einzubetten, die es erlaubt, Informationen über den RFG programmiert abzufragen. Die eingebettete Skriptsprache bietet dem Benutzer jedoch nur relativ primitive Operatoren, die direkt auf der RFG-Bibliothek aufsetzen. Dadurch werden Python-Skripte im Prinzip so aufwändig zu programmieren wie Ada-Programme für den selben Zweck.

Aufgabenstellung

In dieser Diplomarbeit soll eine Skriptsprache für Gravis/RFG geschaffen werden, die sich an bekannten, relativ abstrakten Formalismen für Anfragesprachen an Graphen orientiert. Solche Formalismen sind zum Beispiel die Tarski Algebra, implementiert in Grok (Holt, 1998, 2002), GreQL (Kamp und Kullenbach, 2001) und die RPA (Fejis et al., 1998). Die Anfragesprache soll möglichst einfach von einem Endanwender benutzt werden können, d.h. von einem Programmierer, der Bauhaus für seine Analysen einsetzt und der das dem RFG zu Grunde liegende Schema kennt. Darüber hinaus soll es auch möglich sein, den RFG durch Skripte zu manipulieren.

Die Anfragesprache soll vollständig in Gravis integriert werden. Dazu muss sichergestellt werden, dass die Visualisierung in Gravis effizient während des Ablaufs der

Skripte aktualisiert wird. Aus den Skripten heraus sollen Gravis-Dialoge angesteuert werden können.

Die Effizienz der Realisierung soll mit anderen Werkzeugen durch Messung verglichen werden (Grok, Gupro/GreQL sowie CrocoPat), sofern die Werkzeuge für die Evaluierung zur Verfügung stehen.

Dem Bearbeiter dieser Diplomarbeit ist es überlassen, die bereits existierende Pythonanbindung zu verwenden oder gegebenenfalls die Operatoren in eine andere Skriptsprache (oder auch funktionale Sprache) einzubetten oder auch eine eigene Skriptsprache zu implementieren.

Auf die Qualität der Dokumentation, des Entwurfs und der Implementierung sowie auf einen umfangreichen Test wird großen Wert gelegt. Sämtliche Programmdokumentationen sind in Englisch zu verfassen.

1.2. Gliederung

Kapitel 2 beschreibt den Rahmen des Bauhaus-Projekts, den Resource-Flow-Graph und den Grapheneditor Gravis. In Kapitel 3 werden verschiedene Anfragesprachen mit Anwendungsbeispielen vorgestellt. Kapitel 4 beschreibt die Anforderungen an die Sprache und ihre Konzeption. In Kapitel 5 wird der Aufbau der Sprache sowie die Semantik aller darin enthaltenen Elemente beschrieben. Außerdem wird auf die Einbindung der Sprache in Bauhaus eingegangen. Kapitel 6 stellt den Entwurf sowie die Funktionsweise des Interpreters dar. Abschliessend wird in Kapitel 7 die Effizienz der erstellten Sprache mit anderen Realisierungen verglichen.

2. Grundlagen

In diesem Kapitel wird das Bauhaus-System und der Resource Flow Graph (RFG), die den Rahmen für die Diplomarbeit bilden, beschrieben.

2.1. Bauhaus

Das Bauhaus-System der Abteilung Programmiersprachen und Übersetzerbau des Institut für Softwaretechnologie der Universität Stuttgart stellt Beschreibungsmittel, Analysen und Werkzeuge für die Arbeit mit Quellcode im Kontext des Reverse Engineering und der Wartung zur Verfügung. Es unterstützt die Herleitung von Architekturinformationen aus Quellcode, die Identifikation von Modulen und Komponenten, die Planung von Architektur- und Strukturänderungen und die Abschätzung deren Auswirkungen.

Üblicherweise ist zu Anfang nur der Quellcode eines Systems ohne weitere Informationen vorhanden. Dieser wird Mithilfe eines speziellen Compilers in einen Resource Flow Graph übersetzt. Dies geschieht entweder in einem Schritt oder über die Zwischendarstellung *Intermediate Language* IML [6]. Welcher Weg gewählt wird, hängt von der derzeitigen Unterstützung der einzelnen Quellsprache ab.

Die weiteren Arbeiten finden dann auf dem RFG statt.

2.2. Der Resource-Flow-Graph

Ein Resource-Flow-Graph baut auf einem gerichteten Graphen auf, wurde jedoch um bestimmte Elemente erweitert.

2.2.1. Ein gerichteter Graph

Gewöhnlich wird ein gerichteter Graph als ein Tupel $G = (V, E)$ mit einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$ definiert. Dabei beginnt eine Kante $e = (n_s, n_e)$ beim Startknoten n_s und endet am Endknoten n_e.

2.2.2. Ein Resource-Flow-Graph

Für einen RFG ist diese Definition nicht ausreichend.

Genau wie oben enthält ein RFG eine **Menge von Knoten** N. Jedem Knoten wird ein Typ sowie evtl. weitere Attribute zugeordnet.

Die Abbildung $M_{n,\text{Typ}} : N \times T$ ordnet jedem Knoten $\in N$ einen Typ zu.

Des Weiteren sind **Kanten** in einem RFG nicht alleine durch die mit ihnen verbundenen Knoten definiert. Kanten existieren im RFG als eine Menge E eigenständiger Objekte. Jeder Kante wird ein Startknoten, ein Endknoten, ein Typ sowie evtl. weitere Attribute zugeordnet.

$\sigma : E \times N$ ist eine Abbildung, die jeder Kante einen Startknoten zuordnet.

$\eta : E \times N$ ist eine Abbildung, die jeder Kante einen Endknoten zuordnet.

Die Abbildung $M_{e,\text{Typ}} : E \times T$ ordnet jeder Kante $\in E$ einen Typ zu.

Zusätzlich enthält ein RFG eine Menge V von benannten **Views**, in denen eine Teilmenge der im RFG vorhandenen Knoten und Kanten (im folgenden Elemente genannt) sichtbar gemacht werden können. Dabei gilt, dass zu einer in einer bestimmten View sichtbaren Kante auch deren Start- und Endknoten dort sichtbar sein müssen. Views sind ein effizientes Mittel zur Operation mit Objektmengen, z.B. um unwesentliche Elemente von der weiteren Bearbeitung auszuschliessen.

Die Abbildung $M_{n,V} : N \times V^*$ ordnet jedem Knoten $\in N$ eine Menge von Views zu, in denen er sichtbar ist.

Die Abbildung $M_{e,V} : E \times V^*$ ordnet jeder Kante $\in N$ eine Menge von Views zu, in denen sie sichtbar ist.

Zusätzlich gilt: $v \in M_{e,V}(e) \rightarrow v \in M_{n,V}(\sigma(e)) \wedge v \in M_{n,V}(\eta(e))$
(ist eine Kante e in einer View v sichtbar, müssen auch deren Start- und Endknoten sichtbar sein).

Daraus lässt sich nun die Definition eines RFG $R = (N, E, \sigma, \eta, V, M)$ formulieren, wobei $M = (M_{n,V}, M_{n,\text{Typ}}, M_{e,V}, M_{e,\text{Typ}}, \ldots)$ die Menge von Abbildungen ist, mit denen den einzelnen Elementen Typen, Views sowie weitere Attribute wie Dateiname zugeordnet werden.

2.2.3. Konventionen

Diese Definitionen werden von einigen Konventionen begleitet, die alle architekturverarbeitenden Programme beachten. Diese Konventionen werden im folgenden auszugsweise dargestellt. Eine vollständige Abdeckung ist nicht angestrebt, da sie zum einen ständig im Fluß sind und zum anderen die Skriptsprache so wenig Kenntnis wie möglich vom internen Aufbau des RFGs haben soll.

- Jeder RFG hat eine Base-View, in der alle bei der ursprünglichen RFG-Generierung entstandenen Elementen enthalten sind.
- Es gibt eine Call-View, die Elemente enthält, die mit Aufrufstrukturen zu tun haben.
- Es gibt ein Schema, das festlegt, welche Kantentypen zwischen Knoten bestimmter Typen erlaubt sind. Dieses Schema entsteht momentan implizit durch die Implementierung der Werkzeuge. Es wird jedoch angestrebt, die Validierung eines RFG gegen ein explizit formuliertes Schema zu ermöglichen.
- Views können eine durch einen bestimmten Kantentyp dargestellte Hierarchie haben, z.B. eine part-of Beziehung.

2.2.4. Die RFG-Bibliothek

Eisenbarth [3] beschreibt die von der RFG-Bibliothek angebotenen Datentypen sowie die Operationen für den Zugriff auf und die Manipulation von RFGs. Im folgenden wird ein Überblick über die Datentypen und Operationen gegeben, die auch die Skriptsprache anbieten wird.

Von der zentrale Datenstruktur RFG aus kann auf alle anderen Daten zugegriffen werden. Jedes Element ist mit dem RFG verknüpft und kann nicht ohne Anstrengung in einen anderen RFG verschoben werden.

Über eine View kann auf Informationen wie den Viewnamen zugegriffen werden, sowie Viewmengen zusammengestellt werden. Die Bibliothek stellt momentan bis zu 64 Views zur Verfügung. Da die Zuordnung eines Objekts zu einer View durch ein Bitfeld realisiert ist, lässt sich diese Grenze nicht ohne Aufwand erhöhen.

Es sind Typen für einzelne Elemente definiert. Auch aus diesen können Mengen zusammengestellt werden. Elemente können in Views sichtbar und wieder unsichtbar gemacht werden. Außerdem können sie mit Attributen versehen werden, die jedoch zuvor registriert werden müssen. Einige Attribute wie Objektname oder Zeilennummer sind von vorn herein beim ersten Erstellen eines RFGs registriert.

Zusätzlich zu diesen einfachen Funktionen existieren weitere Pakete, z.B. zum Zugriff auf Nachbarknoten, zur Iteration über Mengen, verschiedene Mengenoperation sowie Analysen.

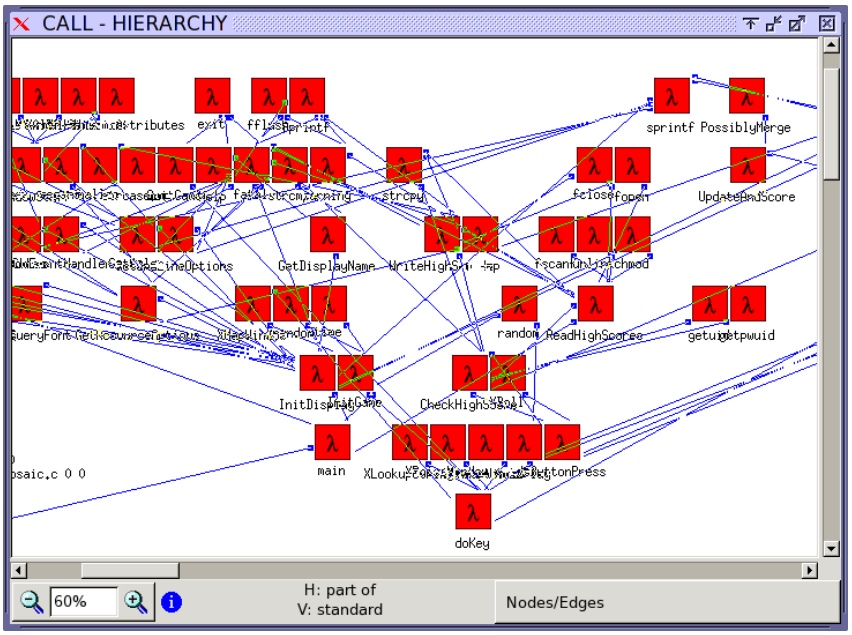
2.3. Graphen-Editor Gravis

Der im Baushaus-System eingebundene Graphen-Editor *Gravis* ermöglicht die Visualisierung und Manipulation von in einem RFG enthaltenen Informationen in Form eines

Graphen.

Gravis stellt Elemente nach Views getrennt dar. Für jede View können eigene Fenster geöffnet werden, die nur die in der entsprechenden View sichtbaren Elemente anzeigt (siehe Abbildung 2.1). In dieser Ansicht einer View können noch Elemente nach verschiedenen Kriterien ausgeblendet werden.

Abbildung 2.1 Grapheneditor Gravis



Elemente können mit der Maus selektiert und manipuliert werden oder nach verschiedenen Kriterien abgefragt werden. Mittels eingebauter Layout-Funktionen können die oftmals unübersichtlichen Graphen auf verschiedene Art angeordnet werden.

Metriken können auf verschiedene Weise visualisiert werden.

Der mit manchen Knoten verknüpfte Quellcode wird durch einen Doppelklick angezeigt.

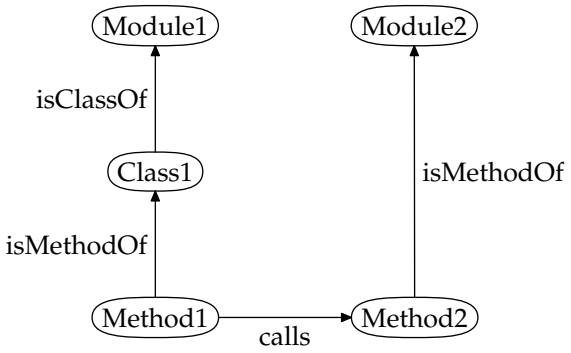
Analysen werden durch Auswahl über die Oberfläche gestartet und reichern den RFG mit zusätzlichen Informationen an.

3. Anfragesprachen

In diesem Kapitel werden verschiedene Anfragesprachen für Graphen vorgestellt.

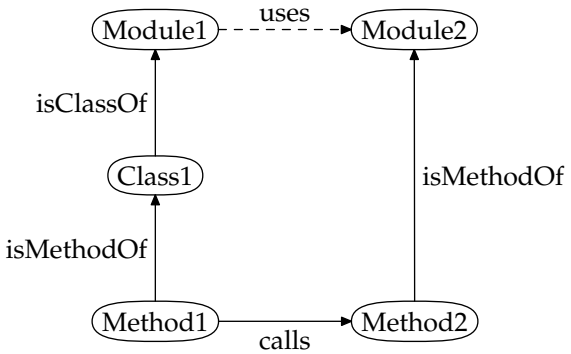
Als Grundlage für die Anwendungsbeispiele für Anfrage soll ein Graph dienen, der u.a. auch Teilgraphen wie in Abbildung 3.1 enthält.

Abbildung 3.1 Uses-Beziehungen (1)



Es sollen jeweils *Uses*-Beziehungen aus den vorhandenen *isClassOf*-, *Calls*- und *isMethodOf*-Kanten gewonnen werden (in Abbildung 3.2 gestrichelt eingezeichnet).

Abbildung 3.2 Uses-Beziehungen (2)



3.1. Relationale Algebra

In diesem Abschnitt wird auf die Grundlagen der relationalen Algebra eingegangen. Sie wird in verschiedenen Anfragesprachen verwendet und wird auch Teil der zu konzipierenden Skriptsprache werden.

3.1.1. Definition

Tarski [8] baut auf Peirce auf, der 1870 die ersten Arbeiten zur relationalen Algebra veröffentlichte. Tarski baut seine Definition der relationale Algebra auf der Booleschen Algebra auf.

Er führt folgende relationalen Konstanten ein:

1	universelle Relation (*universal relation*), sie enthält alle möglichen Kombinationen von Knotenpaaren,
0	Nullrelation (*null relation*),
$1'$	Identitätsrelation (*identity relation*), sie enthält alle Knotenpaare mit identischen Start- und Endknoten
$0'$	Unterschiedsrelation (*diversity relation*), sie enthält alle Knotenpaare mit unterschiedlichichen Start- und Endknoten; $0' = 1 - 1'$.

Außerdem definiert er folgende Operatoren:

$\overline{}$	Komplement (*complement*) $(1 - R)$
$\breve{}$	Umkehrung (*converse*) $((a, b) \rightarrow (b, a))$
$+$	Addition (*addition*), Vereinigung $R_1 \cup R_2$ zweier Relationen
$\cdot$	Multiplikation (*multiplication*), Schnitt $R_1 \cap R_2$ zweier Relationen
$\dagger$	Relative Addition (*relative addition*)
;	Relative Multiplikation (*relative multiplication*), Relationale Komposition $((a, b) \in R_1, (b, c) \in R_2 \rightarrow (a, c))$
$=$	Identität (*identity*)

und verschiedene Axiome.

3.1.2. Anwendung

Die in einem Graph vorhandenen Kanten können als eine binäre Relation zwischen Knoten aufgefasst werden. Sind die Kanten mit Typen markiert, können mehrere Relationen je nach Typ konstruiert werden.

Die Knotenmengen m (*Module*-Knoten) sowie die Relationen M (*isMethodOf*-Relation), L (*isClassOf*-Relation) sowie C (*Calls*-Relation) stehen in diesem Beispiel durch die Typannotation per Definition zur Verfügung.

$$m = \{n \in \text{alle Knoten} | n_{\text{Type}} = \text{Module}\}$$
$$M = \{e \in \text{alle Kanten} | e_{\text{Type}} = \text{isMethodOf}\}$$
$$N = \{(n, n) | n_{\text{Type}} = \text{Module}\}$$
$$L = \{e \in \text{alle Kanten} | e_{\text{Type}} = \text{isClassOf}\}$$
$$C = \{e \in \text{alle Kanten} | e_{\text{Type}} = \text{calls}\}$$

Im nächsten Schritt wird von der Möglichkeit eines eingeschobenen *Class*-Knotens abstrahiert. Die Relation M' enthält nur Kanten, die direkt von einem *Method*-Knoten zu einem *Module*-Knoten führen.

$$M' = \underbrace{(M;N)}_{\substack{\text{Komposition von M und N;} \\ \text{enthält nur Kanten von Methoden,} \\ \text{die in } \textit{Module}\text{-Knoten enden}}} + \underbrace{(M;L)}_{\substack{\text{Komposition von M und L;} \\ \text{enthält nur Kanten von Methoden, die über eine } \textit{isMethodOf}\text{-Kante und eine } \textit{isClassOf}\text{-Kante in } \textit{Module}\text{-Knoten enden}}}$$

Im letzten Schritt wird durch Komposition der M'-Relation und der C-Relation die gesuchte U-Relation berechnet. Da die Komposition auch Knotenpaare mit identischen Start- und Endknoten enthalten, werden diese durch Subtraktion der Identitätsrelation entfernt.

Module-Knotenpaare, die rückwärts über ein (Module, Method)-Knotenpaar, eine *Calls*-Kante und ein weiteres (Module, Method)-Knotenpaar erreichbar sind

$$U = \overbrace{(\underbrace{\breve{M}'}_{\text{Umkehrung von } M'};C;M')} - \underbrace{1'}_{\text{Identitätsrelation}}$$

3.2. Grok

Holt [5] erweitert für das Grok System[1] die relationale Algebra um verschiedene Operatoren und bettet sie in eine Skriptsprache ein.

3.2.1. Universum

Das Grok System arbeitet auf einer Wissensbasis, die aus Dateien eingelesen und gespeichert werden kann.

Relationen sind nach dem Rigi Standard Format (RSF) aufgebaut. Wong [9] beschreibt

[1] es wird auf die Version 83 Bezug genommen

es als eine durch Leerzeichen getrennte Sequenz von Tripeln. Eine Kante wird mit dem Tripel (Relation, Startknoten, Endknoten) beschrieben. Genauso können Attribute als (Objekt, Attributname, Attributwert)-Tupel dargestellt werden.

In Abbildung 3.3 wird ein Auszug aus einer RSF-Datei dargestellt.

Mengen werden durch Strings, die durch Leerzeichen getrennt sind, dargestellt.

Abbildung 3.3 Rigi Standard Format

```
CALL          ASTEvaluationEngine   JavaObjectRuntimeContext
CALL          ASTEvaluationEngine   Message
CALL          ASTEvaluationEngine   RuntimeContext
INHERITANCE   ASTEvaluationEngine   IAstEvaluationEngine
CONTAINMENT   ASTEvaluationEngine   IJavaDebugTarget
CONTAINMENT   ASTEvaluationEngine   IJavaProject
CONTAINMENT   ASTEvaluationEngine   List
```

3.2.2. Relationale Operatoren

Grok stellt u.a. die in Abbildung 3.4 dargestellten relationale Operatoren zur Verfügung, die teilweise schon aus der relationalen Algebra bekannt sind.

Abbildung 3.4 Relationale Operatoren in Grok

`R1 + R2`	Vereinigung R1 $\cup$ R2 zweier Relationen
`R1 - R2`	Differenz R1 $-$ R2 zweier Relationen
`R1 ^ R2`	Schnittmenge R1 $\cap$ R2 zweier Relationen
`R1 o R2`	Relationale Komposition
`inv R`	Inversion von R
`s . R`	Projektion der Menge s durch die Relation R
`R . s`	Umgekehrte Projektion der Menge s durch die Relation R
`R+`	Transitiver Abschluss von R
`R*`	Reflektiver transitiver Abschluß von R
`R1 □ R2`	Vergleichsoperator, □ $\in$ { == oder =, ~=, <, <=, >, >= }
`stc R`	Symmetrischer transitiver Abschluß von R
`# R`	Kardinalität (Anzahl der Tupel) von R
`dom R`	Domain von R
`rng R`	Range von R
`ent R`	Range und Domain von R
`id s`	Identitätsrelation der Menge s
`ID`	Identitätsrelation

3.2.3. Mengenoperatoren

Grok stellt die in Abbildung 3.5 dargestellten Mengenoperatoren zur Verfügung.

Abbildung 3.5 Mengenoperatoren in Grok

`s1 + s2`	Vereinigung von Mengen
`s1 - s2`	Differenz von Mengen
`s1 ^ s2`	Schnittmenge von Mengen
`s1` □ `s2`	Vergleichsoperator, □ ∈ { `==` oder `=`, `~=`, `<`, `<=`, `>`, `>=` }
`# s`	Kardinalität von s
`s1 X s2`	Kreuzprodukt s1 × s2
`{ e1, e2, ... }`	Erzeuge eine Menge aus den Elementen e1, e2, ...

3.2.4. Anweisungen

Es stehen folgende Anweisungen zur Verfügung:

x `:=` *expn*

Zuweisung des Ausdrucks expn zur Variable x

`if` *expn* `then` *statements* `else` *statements* `end if`

Bedingte Ausführung

`loop` *statements* `end loop`

Schleife, wird mit `exit` verlassen

`for` *e* `in` *s* *statements* `end for`

Ausführung der Anweisungen mit jedem Element e aus der Menge s

`exit`

Verlasse `loop` bzw. `for`

`exit when` *expn*

Verlasse `loop` bzw. `for`, wenn der Ausdruck wahr ist

3.2.5. Anwendung

Wie in Abschnitt 3.1.2 sollen *Uses*-Beziehungen aus einem Graphen extrahiert werden.

Dazu werden zuerst aus dem kompletten Graphen die Knoten extrahiert, die den entsprechenden Typ Module, Method oder Class haben. Durch das Laden des entsprechenden Schemas wird die *$INSTANCE*-Relation erzeugt, die jedem Knoten einen Typ des Namens `$_`*Typname* zuordnet.

Die Extraktion geschieht durch Projektion der *$INSTANCE*-Relation mit einer Menge, die nur aus einem Element, dem Typnamen, besteht.

```
modules := $INSTANCE . {"$_Module"}
methods := $INSTANCE . {"$_Method"}
classes := $INSTANCE . {"$_Class"}
```

Im nächsten Schritt wird von der Möglichkeit eines eingeschobenen *Class*-Knotens abstrahiert. Die *inModule*-Relation enthält nur Kanten, die direkt von einem *Method*-Knoten zu einem *Module*-Knoten führen.

```
inModule :=   (isMethodOf o (id modules))
            + (isMethodOf o isClassOf)
```

Im letzten Schritt wird durch Komposition der *inModule*-Relation und der *Calls*-Relation die gesuchte *Uses*-Relation berechnet. Da die Komposition auch Knotenpaare mit identischen Start- und Endknoten enthält, werden diese durch Subtraktion der Identitätsrelation entfernt.

```
uses := ((inv inModule) o calls o inModule)
        - ID
```

Für die Ausgabe können nun noch alle uninteressanten Knoten (und aus ihnen aufgebaute Relationen) aus dem Universum entfernt werden, so dass nur noch *Module*-Knoten übrig bleiben.

```
delset classes
delset methods
```

3.3. GReQL

Die Graph Repository Query Language (GReQL) ist eine SQL-ähnliche, rein deklarative Anfragesprache, die keine Graphmanipulationen erlaubt.

3.3.1. Universum

Anfragen werden gegen ein Repository gestellt, in dem Objekte und deren Beziehungen gespeichert werden. Beziehungen sind gerichtet und können Schlingen bilden. Objekte und Beziehungen können attributiert sein. Beide sind in einer Vererbungshierarchie angeordnet, wobei Mehrfachvererbungen möglich sind. Dabei wird zwischen einer Klasse, die einen Typ und seine Untertypen umfasst, und einem einzelnen Typ unterschieden.

3.3.2. Anfragen

Eine Anfrage wird in Form eines sog. FWR-Ausdrucks gestellt.

Jede Anfrage besteht aus 3 Teilen:

`FROM`
: Deklaration von Graph-Objekten

`WITH`
: optionale Prädikate, die die Objekte erfüllen müssen

`REPORT`
: Beschreibung der Ausgabe der Ergebnisse

Im `WITH`- und `REPORT`-Teil der Anfrage können beliebige GReQL-Ausdrücke verwendet werden. Da FWR-Ausdrücke selbst Ausdrücke sind, kann man sie somit verschachteln.

Im `FROM`-Teil werden Variablen deklariert und ihr Wertebereich definiert. Als Wertebereich sind alle im Schema vorkommenden Typen, Klassen und Beziehungen erlaubt, die mit der Syntax `Vtype{`*`Knotentyp`*`}`, `V{`*`Knotentyp`*`}` und `E{`*`Kantentyp`*`}` angegeben werden.

Neben Booleschen Operatoren und Funktionen sind in Ausdrücken Pfadausdrücke erlaubt. Diese erlauben es, Beziehungen zwischen bestimmten Objekten und Mengen über einen Pfad erreichbarer Objekte zu beschreiben.

Ein einfacher Pfad kann die Form `-->{`*`TypeId`*`}`, `<--{`*`TypeId`*`}` oder `<->{`*`TypeId`*`}` annehmen, wobei das Pfeilsymbol die Richtung der gewünschten Beziehung angibt. Sind beide Richtungen erlaubt, wird dies durch `<->` ausgedrückt. Des Weiteren ist noch die Angabe einer *Goalrestriction* möglich, die den Typ des zweiten Objekts weiter einschränkt. Dies geschieht durch den Anhang `&{`*`TypeId`*`}`.

Beziehungen zwischen Objekten können durch Pfadausdrücke in der Art regulärer Ausdrücke angegeben werden. Dabei gibt es folgende Möglichkeiten:

Sequenz Hintereinanderschreiben von einfachen Pfaden,
z.B. `-->{calls}-->{isMethodOf}`

Alternative Trennung zweier Pfade mit `|`,
z.B. `-->{isMethodOf}|(-->{isClassOf}-->{isMethodOf})`

Option optionale Pfade werden in eckigen Klammern gesetzt

Iteration Iterationen werden durch Anhängen eines `+` oder `*` ermöglicht, wobei die Beziehung bei `+` mindestens ein Mal, bei `*` auch null Mal durchlaufen wird.

Zusätzlich zu der Prädikatform können Pfadausdrücke auch in Form einer Funktion benutzt werden, die dann die Menge über den angegebenen Pfad erreichbarer Objekte zurück liefert.

Abbildung 3.6 zeigt einen Auszug der von der GReQL-Bibliothek bereitgestellten Funktionen.

Abbildung 3.6 Funktionen in GReQL

`avg`	berechnet das arithmetische Mittel der Werte der Komponenten
`cnt`	berechnet die Anzahl der Komponenten
`sum`	berechnet die Summe der Werte der Komponenten
`degree`, `inDegree`, `outDegree`	Anzahl der Kanten, die den übergebenen Knoten als Start- oder Zielknoten haben
`edgesOf`, `edgesTo`, `edgesFrom`	Liste der Kanten des angegeben Knotens
`edgesOfSet`, `edgesToSet`, `edgesFromSet`	Menge der Kanten des angegeben Knotens
`neighbours`, `inNeighbours`, `outNeighbours`	Liste der Nachbarn des angegeben Knotens
`startVertex`, `targetVertex`	Startknoten bzw. Zielknoten der angegebenen Kante
`concat`	Verbinden der angegebenen Listen
`distinct`	Entfernen aller doppelten Elemente aus der Liste
`nthElement`	liefert das n-te Element einer Liste
`pos`	liefert die Position eines Elements in einer Liste
`theElement`	liefert bei einer ein-elementigen Liste dieses Element
`union`	Vereinigung zweier Mengen
`uUnion`	vereinigt die in einer Menge enthaltenen Mengen
`connects`	prüft, ob die angegebene Kante die angegebenen Knoten verbindet
`hasClass`	prüft, ob das angegebene Objekt von der angegebenen Klasse ist
`hasType`	prüft, ob das angegebene Objekt vom angegebenen Typ ist
`isNeighbourOf`	prüft, ob die angegebenen Knoten benachbart sind
`isIn`	prüft, ob das angegebene Objekt Teil der angegebenen Menge ist

Ausgaben erzeugen i.a. Tabellen, die die Ergebnismengen oder -Listen darstellen.

3.3.3. Anwendung

Es sollen *Uses*-Beziehungen aus einem Graphen extrahiert werden.

Dazu werden zuerst die Variablen deklariert, die die *Module*-Knoten aufnehmen sollen. Dabei wird durch die Angabe eines Wertebereichs der Typ der Knoten eingeschränkt.

```
FROM m1, m2 : V{Module}
```

Nun werden die Prädikate angegeben, die die Knoten erfüllen müssen. Die Pfadausdrücke geben den Pfad an, der zwischen den Knoten existieren muss. Dabei werden die *isClassOf*-Kanten als optional angegeben. Schließlich wird noch der Fall, dass beide *Module*-Knoten identisch sind, ausgeschlossen.

In diesem Beispiel wird auf die Festlegung der Typen der innerhalb des Pfads liegender Objekte keinen Wert gelegt.

```
WITH ( m1 [<-{isClassOf}]
          <-{isMethodOf}
          ->{calls}
          ->{isMethodOf}
          [->{isClassOf}]
       m2 )
     AND (m1 <> m2)
```

Schließlich erfolgt die Ausgabe der Namen beider *Module*-Knoten in einer Tabelle.

```
REPORT m1.name, m2.name
END
```

3.4. Binary Decision Diagrams: CrocoPat

Beyer et al. [2] beschreiben ein auf Binary Decision Diagrams (BDD) aufbauendes Verfahren für die Anfrage an Graphen.

Ein Vorteil dieser Darstellung ist der im Vergleich zu mit Knotenlisten oder -tabellen arbeitenden Systemen geringere Speicherverbrauch und die effizientere Verarbeitung. Außerdem erlaubt sie die Suche nach beliebig großen Teilgraphen.

3.4.1. Universum

CrocoPat arbeitet auf den als Kommandozeilenargumente angegebenen RSF-Dateien. Diese sind jedoch nicht auf die 3-Tupel (Relation, Startknoten, Endknoten) beschränkt,

sondern es ist eine beliebige Anzahl Knoten erlaubt. Allerdings ist darauf zu achten, dass die Tupel in einer Datei immer die gleiche Anzahl Knoten enthalten.

3.4.2. Anweisungen und Ausdrücke

Abbildung 3.7 zeigt die in einem Ausdruck erlaubten Konstrukte.

Es gibt drei verschiedene Anweisungen: Mit der Anweisung `PRINT` wird ein Text auf der Standardausgabe ausgegeben. Die Anweisung `SAVE` speichert den Inhalt der angegebenen relationalen Variable in einer RSF-Datei ab.

Mit einer Zuweisung `R (t1, ...) := exp` wird einer relationalen Variable ein Ausdruck zugewiesen. Dabei gelten folgende Bedingungen: alle relationalen Variablen auf der rechten Seite müssen zuvor definiert worden sein. Dies kann durch das Einlesen von RSF-Dateien oder eine Zuweisung geschehen. Außerdem muss die Anzahl der auf der linken Seite angegebenen Terme $t_1, \ldots$ gleich der Anzahl der freien Variablen auf der rechten Seite sein. Eine Variable ist frei, wenn sie in einem Ausdruck außerhalb eines Quantors mit dieser Variable auftritt.

Abbildung 3.7 Ausdrücke in CrocoPat

`TRUE`	boolesches Literal
`FALSE`	boolesches Literal
`R (t, ...)`	relationale Variable R, die Terme t dürfen nur Konstanten und Variablen enthalten; es gibt keine Funktionen
`t = t`	Identitätsrelation $\{(u,v) \mid u,v \in \text{DOM},\ u = v\}$, wobei DOM die Menge aller Knoten ist
`! exp`	Negation des Ausdrucks *exp*
`exp & exp`	Konjunktion
`exp | exp`	Disjunktion
`exp -> exp`	Implikation
`EX (v, exp)`	Existenzquantor
`FA (v, exp)`	Allquantor
`TC (exp, v1, v2)`	transitiver Abschluss

3.4.3. Anwendung

Um die *Uses*-Beziehung zu extrahieren, reicht eine zweistellige Relation aus.

Zuerst wird mit Hilfe der Existenzquantoren nach Kandidaten für *Method*-Knoten gesucht. Es wird davon ausgegangen, dass das Universum eine *isOfType*-Relation enthält, die jedem Knoten einen Typ zuordnet. Dies lässt eine Einschränkung der Knoten auf die passenden Typen zu.

Darauf folgt die Verknüpfung mit der *isMethodOf*-Relation. Es gibt die Alternative eines eingeschobenen Class Knotens, die durch eine Disjunktion aufgenommen wird. Dies geschieht für beide Enden des gesuchten Pfads.

Schließlich erfolgt die Verknüpfung mit der *Calls*-Relation.

```
Uses (ModuleA, ModuleB) :=
  EX(MeA, EX(MeB,   isOfType (ModuleA, "Module")
                  & isOfType (ModuleB, "Module")
                  & isOfType (MeA, "Method")
                  & isOfType (MeB, "Method")
                  & (isMethodOf (MeA, ModuleA)
                     | EX(ClA, isOfType (ClA, "Class")
                               & isMethodOf (MeA, ClA)
                               & isClassOf (ClA, ModuleA)))
                  & (isMethodOf (MeB, ModuleB)
                     | EX(ClB, isOfType (ClB, "Class")
                               & isMethodOf (MeB, ClB)
                               & isClassOf (ClB, ModuleB)))
                  & calls (MeA, MeB)
  ));
```

3.5. GIANT Scripting Language: GSL

Der IML-Browser GIANT [7] enthält eine Skriptsprache, die es ermöglicht, IML-Teilgraphen und Selektionen aus einem IML-Teilgraph anzufragen, sowie Aktionen auf ihnen auszuführen.

GSL erlaubt die Interaktion mit der Browser-Oberfläche. Es können neue Fenster geöffnet werden, neue Elemente angezeigt oder versteckt werden, sowie die Selektion und IML-Graphen manipuliert werden.

3.5.1. Universum

GSL arbeitet auf einer IML-Graph-Datei, die die Informationen über Elemente und Attribute enthält. Es gibt einen speziell markierten Wurzelknoten.

Attribute werden über Namen referenziert und können eine Source Location, einen Booleschen Wert, eine natürliche Zahl, einen String, eine Folge von Strings oder einen Verweis enthalten. Es kann auf einen anderen Knoten, eine Folge von Verweisen, eine Menge von Verweisen oder auf nichts (Nullverweis) verwiesen werden.

3.5.2. Ausdrücke

Die Ausführung eines GSL-Skripts erfolgt durch das Auswerten von Ausdrücken. Veränderungen am Systemzustand erfolgen dabei durch Nebeneffekte der Ausdrücke.

Abbildung 3.8 zeigt die in Ausdrücken verwendbaren Terme.

Abbildung 3.8 GSL Ausdrücke

`False, True`	Boolesche Literale
Integer	Integer Literal
String	String Literal
`null`	Null Literal
`subgraph .` *Identifier*	der Teilgraph mit dem entsprechenden Bezeichner
`selection .` *Identifier*	die Selektion mit dem entsprechenden Bezeichner
`' subgraph .` *Identifier*	Referenz auf den Teilgraph mit dem entsprechenden Bezeichner
`' selection .` *Identifier*	Referenz auf die Selektion mit dem entsprechenden Bezeichner
Identifier	Variablenzugriff
`'` *Identifier*	Referenz auf eine Variable
`+` *Identifier*	neue Variable erzeugen
`(` *expression*`, ...)`	Liste von Ausdrücken
`[` *expression*`; ...]`	Liste von Sequenzen
`{` *list expression* `}`	Deklaration eines Skripts
expression list	Ausführung eines Skripts

Es stehen Mengenoperationen wie Vereinigung, Schnitt und Differenz für Knoten- und Kantenmengen zur Verfügung. Außerdem ist die Auswahl von Elementen anhand von Attributen und Klassen möglich.

Abbildung 3.9 zeigt die in GSL eingebauten Funktionen. Abbildung 3.10 zeigt die in der Standardbibliothek definierten Operationen.

Abbildung 3.9 GSL Standardbibliothek

`set`	weist einer Variable einen neuen Wert zu
`deref`	dereferenziert eine Referenz
`if` *`cond t_branch f_branch`*	wenn *cond* zu `True` evaluiert, wird *t_branch* zurückgegeben, ansonsten *f_branch*
`loop`	führt das übergebene Skript in einer Schleife aus
`error`	Erzeugt eine Fehlermeldung
`run`	führt das Skript mit dem angegebene Namen aus
`add`	addiert zwei Integer-Werte; hängt neue Elemente an eine Menge an
`sub`	subtrahiert zwei Integer-Werte; entfernt Elemente aus einer Menge
`cat`	verkettet zwei Strings miteinander
`less`	$<$-Operator
`equal`	Gleichheits-Operator
`in_regexp`	prüft, ob der angegebene String zum regulären Ausdruck passt
`empty_node_set`	gibt eine leere Knotenmenge zurück
`empty_edge_set`	gibt eine leere Kantenmenge zurück
`get_first`	gibt das erste Element einer Menge zurück
`is_in`	gibt `True` zurück, wenn das angegebene Element in der Menge vorhanden ist
`size_of`	gibt die Länge einer Menge zurück
`get_entry`	gibt das Element an der angegebenen Position der Liste zurück
`is_nodeid`, `is_edgeid`, ...	prüft den Typ des Arguments
`get_current_window`, `set_current_window`	liefert das aktuelle Fenster zurück, oder ändert es
`root_node`	gibt den Wurzelknoten des Graphen zurück
`all_nodes`	liefert die Menge aller Knoten zurück
`all_edges`	liefert die Menge aller Kanten zurück
`has_attribute`	prüft auf Vorhandensein eines Attributs
`get_attribute`	gibt den Wert eines Attributs zurück
`get_type`	gibt den Typ eines Knotens bzw. einer Kante als String zurück
`get_incoming`	gibt die eingehenden Kanten eines Knotens zurück
`get_outgoing`	gibt die ausgehenden Kanten eines Knotens zurück
`get_source`	gibt den Startknoten einer Kante zurück
`get_target`	gibt den Endknoten einer Kante zurück
`input`	fragt einen Wert vom Benutzer ab

Abbildung 3.10 GSL Standardbibliothek

`not (a)`	Negation
`and_then (a, b)`	Konjunktion
`and (a, b)`	Konjunktion; es ist sichergestellt, dass alle Argumente evaluiert werden
`or_else (a, b)`	Disjunktion
`or (a, b)`	Disjunktion; es ist sichergestellt, dass alle Argumente evaluiert werden
`while (b, c)`	Bedingte Schleife
`repeat (c, until)`	Bedingte Schleife
`union, add`	Vereinigung
`intersection (A, B)`	Schnittmenge
`difference, sub`	Differenz
`select_nodes (from, pred)`	Selektion von Knoten anhand eines Prädikats
`build_nodes (edge_set, pred)`	
`for_each (a_set, action)`	Iterator

3.5.3. Anwendung

Es sollen *Uses*-Beziehungen aus einem Graphen extrahiert werden.

Da dieses Beispiel recht umfangreich ist, wird auf den Abdruck der trivial zu implementierenden Funktionen `nodes_with_type`, die alle Knoten eines bestimmten Typs zurückgibt, `select_nodes_with_type`, die alle Knoten eines bestimmten Typs in der angegebenen Menge zurückgibt, und `select_edges_with_type`, die alle Kanten eines bestimmten Typs in der angegebenen Menge zurückgibt, verzichtet.

Zu Anfang des Skripts werden die Argumente deklariert und die Standardbibliothek geladen. In diesem Beispiel sind keine Argumente notwendig.

```
{
  ()
  [
    run ("standard");
```

Nun wird über alle *Module*-Knoten iteriert und ein geeigneter Pfad zu einem zweiten *Module*-Knoten gesucht. Die Menge uses soll am Schluss alle gefundenen *Uses*-Beziehungen aufnehmen, die Menge module_b_set alle am Ende des Pfads gefundenen *Module*-Knoten.

```
    set (+uses, empty_node_set());
    for_each (nodes_with_type ("Module"), { (+module_a) [
      set (+module_b_set, empty_node_set());
```

Die Method_a_set-Menge soll alle *Method*-Knoten aufnehmen, über die man über die Kombination *isClassOf*-Kante und *isMethodOf*-Kante oder auch direkt über eine *isMethodOf*-Kante gelangt.

```
set (+method_a_set, empty_node_set());
set (+is_class_of_a,
    select_edges_with_type ("isClassOf", get_incoming (module_a)));
for_each (is_class_of_a, { (+e1) [
  set (+class_a_set,
       select_nodes_with_type ("Class", get_source (e1)));
  for_each (class_a_set, { (+class_a) [
    set (+is_method_of_a,
      select_edges_with_type ("isMethodOf", get_incoming (class_a)));
    for_each (is_method_of_a, { (+e2) [
      add ('method_a_set,
           select_nodes_with_type ("Method", get_source (e2)));
    ] });
  ] });
] });
for_each (is_method_of_a, { (+e1) [
  add ('method_a_set,
       select_nodes_with_type ("Method", get_source (e1)));
] });
```

Nun werden alle *Method*-Knoten gesucht, die über eine *Calls*-Kante erreichbar sind.

```
for_each (method_a_set, { (+method_a) [
  set (+call,
       select_edges_with_type ("call", get_outgoing (module_a)));
  for_each (call, { (+e3) [
    set (+method_b_set,
         select_nodes_with_type ("Method", get_target (e3)));
```

Nun werden wie oben *isClassOf*-Kanten und *isMethodOf*-Kanten verfolgt, um einen zweiten *Module*-Knoten zu finden.

```
    for_each (method_b_set, {(+method_b) [
      set (+is_method_of_b,
           select_edges_with_type
             ("isMethodOf", get_outgoing (method_b)));
      for_each (is_method_of_b, { (+e4) [
```

```
            add ('module_b_set,
                 select_nodes_with_type ("Module", get_target (e4)));
            set (+class_b_set,
                 select_nodes_with_type ("Class", get_target (e4)));
            for_each (class_b_set, { (+class_b) [
              set (+is_class_of_b,
                   select_edges_with_type
                     ("isClassOf", get_outgoing (class_b)));
              for_each (is_class_of_b, { (+e5) [
                add ('module_b_set,
                   select_nodes_with_type ("Module", get_target (e5)));
              ] });
            ] });
        ] });
    ] });
```

Nun wird noch sichergestellt, dass sich die beiden *Module*-Knoten unterscheiden.

```
    for_each (module_b_set, { (+module_b) [
      if (Equal (module_a, module_b), null, { () [
        add ('uses, module_a);
        add ('uses, module_b);
      ] });
    ] });
  ] });
```

Schließlich werden die erhaltenen Knoten im Editor selektiert.

```
  set ('Selection.Uses, (uses));
]
}
```

3.6. Python-Anbindung der RFG-Bibliothek

Festi [4] beschreibt die Einbettung des Python-Interpreters in Gravis. Er erstellt dazu ein Binding der RFG-Bibliothek für Python.

3.6.1. Universum

Die RFG-Bibliothek arbeitet auf einem in sich geschlossenen Graphen, der in einer Binärdarstellung gespeichert wird.

3.6.2. Objekte und Methoden

Das Wurzelobjekt der Schnittstelle ist der RFG. Abbildung 3.11 zeigt die implementierten Attribute und Methoden. Neben dem RFG-Objekt gibt es Objekte für Views, Knoten und Kanten, Knoten- und Kantenmengen, sowie zum Zugriff auf Gravis.

Abbildung 3.11 RFG Attribute und Methoden

`views`	ein Dictionary, das alle Views unter ihrem Namen enthält
`new_view(`*name*`)`	erzeugt eine neue View mit dem angegebenen Namen
`new_node(`*typ*`)`	erzeugt einen neuen Knoten mit dem angegebenen Typ
`new_edge(`*typ*`,`*from*`,`*to*`)`	erzeugt eine neue Kante
`edge_attributes`	ein Dictionary, das die Kantenattribute unter ihrem Namen enthält
`node_attributes`	ein Dictionary, das die Knotenattribute unter ihrem Namen enthält
`new_edge_attribute (`*name*`,`*typ*`,`*per_view*`,`*unique*`)`	erstellt ein neues Kantenattribut
`new_node_attribute (`*name*`,`*typ*`,`*per_view*`,`*unique*`)`	erstellt ein neues Knotenattribut
`insert(`*item*`)`	fügt einen Knoten oder eine Kante in den RFG ein
`remove(`*item*`)`	entfernt einen Knoten oder eine Kante aus dem RFG
`save_plain(`*filename*`)`	speichert den RFG
`load_plain(`*filename*`)`	lädt einen RFG

3.6.3. Anwendung

Es sollen *Uses*-Beziehungen aus einem Graphen extrahiert werden.

Für dieses Beispiel wird angenommen, dass entsprechende Prädikate für Knoten- und Kantentypen definiert sind.

Zu Anfang werden die benötigten Bibliotheken und der zu bearbeitende RFG geladen.

```
import rfgs
from rfgs_predicates import *

RFG = rfgs.open_rfg_plain("example.rfg")
```

Nun wird über alle *Module*-Knoten im Graph iteriert. Die Knotenmenge Methods_A soll alle *Method*-Knoten aufnehmen, die entweder direkt über eine *isMethodOf*-Kante

oder über eine *isClassOf*-Kante und eine *isMethodOf*-Kante erreicht werden.

Die Predecessors- und Successors-Methoden liefern jeweils eine Knotenmenge mit den Vorgänger- bzw. Nachfolgerknoten. Als Argumente können dabei zusätzliche Kanten- und Knotenprädikate angegeben werden, die die Ergebnismenge weiter einschränken.

```
for Module_A in RFG.nodes(Is_Module):
  Methods_A = Module_A.predecessors(is_isMethodOf_edge, Is_Method)
  for Class_A in Module_A.predecessors(is_isClassOf_edge, Is_Class):
    for Method_A in Class_A.predecessors(is_isMethodOf_edge, Is_Method):
      Methods_A.insert(Method_A)
```

Nun wird über alle über eine *Calls*-Kante erreichbaren Nachfolgerknoten der gefundenen *Method*-Knoten iteriert. Die Knotenmenge Modules_B soll alle *Module*-Knoten aufnehmen, die entweder direkt über eine *isMethodOf*-Kante oder über eine *isMethodOf*-Kante und eine *isClassOf*-Kante erreicht werden.

```
  for Method_B in Method_A.successors(is_Calls_edge, Is_Method):
    Modules_B = Method_B.successors(is_isMethodOf_edge, Is_Module)
    for Class_B in Module_B.successors(is_isMethodOf_edge, Is_Class):
      for Module_B in Class_B.successors(is_isClassOf_edge, isModule):
        Modules_B.insert(Module_B)
```

Schließlich wird sichergestellt, das die gefundenen *Module*-Knoten nicht identisch sind und eine entsprechende *Uses*-Kante erzeugt.

```
    if Module_A != Module_B
      RFG.new_edge("uses", Module_A, Module_B)
```

3.7. Bewertung

Grok ist durch die Implementierung der relationalen Algebra sowie die imperativen Kontrollstrukturen sehr mächtig. Allerdings ist der Einarbeitungsaufwand beträchtlich, da Kenntnisse der relationalen Algebra notwendig sind und große Anfragen schwer zu durchschauen sind.

GReQL ist durch seine SQL-ähnliche Anfragesyntax leicht lesbar und auch ohne intime Kenntnisse der Sprache verständlich. Es ist jedoch kein iterativer Aufbau von Datenstrukturen möglich. Außerdem sind keine Manipulationsmöglichkeiten vorgesehen.

Die Ausgabe der Ergebnisse findet in beiden Fällen in Form von Dateien bzw. Text statt.

CrocoPat verwendet eine effiziente und schnelle Datenhaltung für Relationen. Sie lässt sich jedoch nicht einfach mit den Operatoren der relationalen Algebra verknüpfen. Dadurch werden teure Datenstrukturkonvertierungen notwendig.

GSL bietet eine direkte Interaktion und Visualisierung in einem Graph-Editor. Jedoch ist die Syntax nicht auf Anhieb zu verstehen. Es gibt keine relationalen Operatoren. Veränderungen am Graph sind nicht möglich.

Das Python-Binding der RFG-Bibliothek bietet gegenüber einer direkten Implementierung in Ada95 den Vorteil, dass Skripte auch nach dem Kompilieren des Systems erstellt werden können. Obwohl eine Abstraktion der RFG-Bibliothek angestrebt wird, sind viele RFG-Funktionen direkt abgebildet, so dass bei einer Änderung der Bibliothek potentiell alle Skripte zu ändern wären.

Daraus wird klar, dass eine einfache Anfragesyntax von Vorteil ist, aber nicht auf Kontrollstrukturen und Manipulationsmöglichkeiten verzichtet werden kann. Da es im Bauhaus-System die Möglichkeit der direkten graphischen Visualisierung der Ergebnisse gibt, sollte diese genutzt werden. Die Interna der RFG-Bibliothek sollen vor dem Anwender verborgen werden.

4. Sprachkonzeption

In diesem Kapitel werden die aus der Arbeit mit dem Bauhaus-System und der Analyse der Anfragesprachen gewonnenen Anforderungen zusammengestellt und die getroffenen Entwurfsentscheidungen dargelegt.

4.1. Anforderungen

Aus den in Abschnitt 2.3 beschriebenen Arbeitsbedingungen erwächst der Wunsch, momentan manuell ausgeführte Arbeiten zu automatisieren. Auch komplette Analysen sollen nicht wie bisher durch direkte Verwendung der RFG-Bibliothek, sondern in einer abstrakteren Form realisiert werden. Dies bringt u.a. Vorteile für die kommerzielle Nutzung, da der Quellcode des Systems normalerweise nicht verfügbar ist, das System aber trotzem erweitert werden können soll.

Daraus leiten sich folgende Anforderungen an die Skriptsprache ab:

- Es soll einfach möglich sein, Anfragen und komplette Analysen auf dem RFG zu automatisieren. Gerade bei Anfragen, die sich eigentlich einfach programmieren ließen, für die aber der Aufwand für die richtige Integration in das Bauhaus-System zu groß ist, soll die Skriptsprache eingesetzt werden.
- Die Sprache soll von Endanwendern eingesetzt werden können. Diese habe oftmals kein Interesse an der internen Funktionsweise des Systems oder möchten sich die theoretischen Grundlagen nicht erarbeiten. Daher sollen keine schwer verständlichen Formalismen Voraussetzung für den Einsatz der Sprache sein.
- Die Sprache soll direkt in Gravis integriert werden. Damit ist sie in der natürlichen Arbeitsumgebung des Benutzers verfügbar. Außerdem ist beim Einsatz neuer Skripte kein Neucompilieren des Systems notwendig.
- Gebräuchliche Fragestellungen sollen abgedeckt sein. Es muss sich nicht jede Idee verwirklichen lassen, aber die angebotenen Anfragemöglichkeiten sollen auch nicht zu speziell sein.
- Über die Anfragefunktionalität hinaus soll es auch möglich sein, den RFG zu manipulieren.

- Die von Skripten gelieferten Ergebnisse sollen geeignet visualisiert werden, z.B. mit den Möglichkeiten des Grapheneditors. Eine rein textuelle Form ist unbefriedigend.

Zusätzlich sind folgende Eigenschaften für die Integration in das Bauhaus-System wünschenswert:

- Die Sprache soll möglichst abstrakt sein. Sie soll nicht durch Implementierungsdetails des RFGs, wie z.B. Einschränkungen der Attribute oder Views, beeinflusst werden.
- Es müssen jedoch trotzdem die über einen Graphen hinausgehenden Merkmale wie Views benutzbar sein.

4.2. Entscheidungen

4.2.1. Universum

Die Sprache arbeitet immer auf einem einzelnen geladenen RFG als Universum, da ein RFG immer die kompletten Informationen beinhalten soll. Beim Transport einzelner Objekte aus einem RFG in einen anderen würde es zum Verlust der Beziehungsinformationen kommen. Das neue Objekt wäre wertlos. Auch das gleichzeitige Arbeiten auf mehreren RFGs macht wenig Sinn. Schließlich ist auch Gravis nicht dafür ausgelegt, RFG-übergreifend zu arbeiten.

4.2.2. Knoten und Kanten

Die Sprache soll mit Graphen arbeiten können, daher liegt es nahe, Knoten und Kanten als eigenständige Objekte zu halten. Kanten können dabei in Form von Knotenpaaren oder als eigenständige Objekte behandelt werden. Da es im RFG Kanten als eigene Objekte mit Attributen gibt, sollten diese auch behandelbar sein. Dazu ist eine entsprechende Darstellung in der Sprache vorteilhaft.

4.2.3. Pfade

Beim Aufstellen von Algorithmen ist die Vorstellung eines Pfads, über den man bestimmte Knoten erreicht, oft hilfreich. Pfade ermöglichen auch die Angabe eines Ergebnisses, ohne die genaue Anzahl der Knoten auf dem Pfad von vorne herein zu kennen. Daher soll ein Pfad als eigenständiges Beschreibungsmittel in die Sprache aufgenommen werden.

Ein Pfad soll dabei abwechselnd aus Knoten und Kanten bestehen, so dass nie zwei Knoten oder Kanten aufeinander folgen. Es soll nicht festgelegt werden, dass Pfade nur mit Knoten anfangen oder enden, um die Kombination von Teilpfaden nicht zu erschweren und den Aufbau des Pfads durch einzelne Elemente zu ermöglichen.

Dadurch ergibt sich jedoch das Problem, dass nicht von vorn herein feststeht, ob an einem Ende des Pfads ein Knoten oder eine Kante steht. Somit ist der Typ des Elements am Ende nicht bekannt, und es muss Funktionen geben um ihn zu bestimmen.

Es soll möglich sein, die Länge des Pfads abzufragen, sowie einen leeren Pfad zu erzeugen.

Es besteht auch die Möglichkeit, eine Syntax für Pfadliterale einzuführen, so dass durch eine einfache Beschreibung einer neuer Pfad generiert werden kann. Da jedoch Knoten in der Sprache mit RFG-Knoten identisch sein sollen, ist es schwierig, einen einzelnen Knoten im RFG durch ein Literal zu referenzieren, da evtl. umfangreiche Attribute angegeben werden müssen. Auch die Erzeugung neuer Knoten, die in das Schema passen, bedarf diesen Aufwands. Für Kanten gelten die gleichen Überlegungen.

Wenn die Sprache iterative Strukturen zulässt, sind aus orthogonalitätsgründen Iteratoren über Pfade und Funktionen zum iterativen Aufbau von Pfaden zu unterstützen.

4.2.4. Mengen

Zur Zusammenfassung von Elementen sind Mengen sinvoll. Damit sollten auch die Mengenoperationen Vereinigung, Schnitt und Differenz unterstützt werden, sowie eine Abfrage der Kardinalität, der Existenz eines bestimmten Elements in einer Menge und eine Möglichkeit zur Erzeugung leerer Mengen.

Es besteht auch die Möglichkeit, eine Syntax für Mengenliterale einzuführen. Aus denselben Gründen wie bei Pfaden wird davon Abstand genommen.

Wenn die Sprache iterative Strukturen zulässt, sind auch Iteratoren über Mengen und Funktionen zum iterativen Aufbau und Abbau von Mengen zu unterstützen.

Es gibt die Alternativen, einen allgemeinen Mengentyp einzuführen, der beliebige Objekte enthalten darf. Dies würde die Implementierung der Mengenoperationen für verschiedene Typen unnötig machen. Des Weiteren wäre auch die Verschachtelung von Mengen möglich. Allerdings erschwert dies eine Typprüfung vor dem Start des Skripts. Ausserdem gibt es für einzelne Typen effizientere Darstellungen, als es mit einer allgemeinen Menge möglich ist. Daher soll es keine allgemeine Liste geben, sondern jeweils für Knoten, Kanten und Pfade einen eigenen Mengentyp.

4.2.5. Funktionsaufrufe

Funktionen sollen sprechende Parameter haben. Um Skripte auch ohne Kenntnisse der Dokumentation verständlich zu halten, ist es vorteilhaft, die Parameternamen bei Aufruf einer Funktion mit anzugeben.

Da es verschiedene Typen geben kann, die die gleiche Art von Operationen unterstützen sollen, soll es möglich sein Funktionen unter gleichem Namen aber mit verschiedenen Argumenttypen aufzurufen.

4.2.6. Typsystem

Für die Bestimmung des Typs von Funktionen und Variablen gibt es mehrere Alternativen:

- Typen werden dynamisch zur Laufzeit festgelegt. Damit müssen Variablen nicht mit einem Typ versehen werden. Es ist möglich, Funktionen zu definieren, denen der genaue Typ nicht bekannt ist. Dadurch können Mengenoperationen oder Pfadmanipulationen einfacher beschrieben werden.
- Typen sind statisch schon vor der Ausführung des Skripts bekannt. Damit können inkompatible Typen von vorn herein festgestellt und die Ausführung des Skripts unterbunden werden.

Für die Deklaration von Variablen gibt es folgende Alternativen:

- Überhaupt keine Deklaration von Variablen. Variablen werden im Zuge der erstmalige Nutzung erstellt. Variablen können ihren Typ durch neue Zuweisung ändern.
- Keine Deklaration, jedoch legt die erste Zuweisung zur Variable ihren Typ unveränderbar fest.
- Deklaration einer Variablen, aber ohne einen Typ festzulegen. Dabei gibt es wieder die Alternativen, den Typ nach der ersten Zuweisung festzulegen, oder ihn flexibel zu lassen.
- Deklaration einer Variablen und ihres Typs.

Durch die Einführung von Pfaden ist es bei manchen Operationen wie z.B. der Rückgabe des Pfadkopfes nicht möglich, den Typ vor der Ausführung des Skripts genau zu bestimmen. Trotzdem sollen so viele Fehlerquellen wie möglich vor der Ausführung ausgeschlossen werden. Daher sollen Variablen mit Typen im Voraus deklariert werden, so dass in den meisten Fällen eine Prüfung auf korrekte Typverwendung möglich ist. In Fällen, in denen dies nicht möglich ist, müssen die Typen zur Laufzeit geprüft werden.

4.2.7. Ein- und Ausgabe

Die Eingabe der Skriptargumente und die Ausgabe der Ergebnisse sollen über die Oberfläche des Grapheneditors erfolgen. Dabei soll es möglich sein, die Eingabe durch Selektion von Elementen mit der Maus auszuwählen.

Die Ausgabe der Ergebnisse soll im Falle von Graph-Elementen das Selektieren und Markieren dieser Elemente im Grapheneditor unterstützen.

4.2.8. Ausführungsmodell

Es besteht die Möglichkeit, eine rein deklarative Ausführung zu realisieren. Eine imperative Ausführung hat jedoch den Vorteil, dass die Reihenfolge der Ausführung einfacher vom Benutzer bestimmbar ist, und dass Operationen in einzelne Schritte aufgeteilt werden können. Damit ist das Verhalten leichter nachvollziehbar.

5. Sprachaufbau

Dieses Kapitel beschreibt den genauen Aufbau der zu implementierenden, RFGScript genannten Skriptsprache. In den folgenden Abschnitten werden die verfügbaren Datentypen, die Kontrollflusselemente, Ausdrücke und festverdrahteten Operationen definiert und beschrieben. Des Weiteren werden die Benutzeroberfläche und die graphischen Elemente dargestellt.

Um ein Gefühl für die Sprache und den Aufbau eines Skripts zu vermitteln, wird im folgenden ein Beispiel für die Extraktion von *Uses*-Beziehungen aus einem RFG dargestellt. Es gibt mehrere Lösungen für eine solche Anfrage, hier wird die relationale Algebra benutzt.

Zuerst werden die verwendeten Variablen mit ihren Typen deklariert. Dabei wird die Uses-Variable mit `export` markiert, da in ihr die Ergebnisse hinterlegt werden.

```
story Uses is
          Modules      : Node_Set;
          Methods      : Node_Set;
          Classes      : Node_Set;
          In_Module    : Relation;
  Uses         : Relation;
   export Uses_Edges   : Edge_Set;
          Is_Method_Of : Relation;
```

Nun werden aus dem Graph die relevanten Knoten- und Kantenmengen extrahiert.

```
begin
   Modules := Nodes (Type => "Module");
   Methods := Nodes (Type => "Method");
   Classes := Nodes (Type => "Classes");
   Is_Method_Of := Edges (Type => "isMethodOf");
```

Nun wird von der Möglichkeit eines eingeschobenen *Class*-Knotens abstrahiert. Dies geschieht durch Vereinigung der auf Module zeigenden *isMethodOf*-Kanten mit der Komposition der *isMethodOf*-Kanten mit *isClassOf*-Kanten.

```
In_Module :=
   Union (L => Composition (L => Is_Method_Of,
                            R => Identity (Of => Modules)),
          R => Composition (L => Is_Method_Of,
                            R => Edges (Type => "isClassOf")));
```

Durch die Komposition der *In_Module*-Relation mit den *Calls*-Kanten wird nun die *Uses*-Relation berechnet.

```
Uses :=
   Composition (L => Composition (L => Converse (Of   => In_Module)
                                  R => Edges    (Type => "calls")),
                R => In_Module);
```

Nun wird noch die Identitätsrelation der *Module*-Knoten abgezogen, um Kanten mit identischen Start- und Endknoten zu eliminieren.

```
Uses := Difference (L => Uses, R => Identity (Of => Modules));
```

Schließlich werden die berechneten Knotenpaare als Kanten in den RFG eingetragen.

```
   Uses_Edges := Realize (What => Uses, Type => "Uses");
end story;
```

5.1. Aufbau eines Skripts

Der Quelltext eines Skripts beginnt mit dem Schlüsselwort `story`, gefolgt vom Skriptnamen und dem Schlüsselwort `is`.

In der darauf folgenden Liste werden die während der gesamt Laufzeit vorhandenen Variablen mit ihren Typen deklariert. Sind Variablen vor der Ausführung zu belegen, werden sie mit dem Schlüsselwort `import` markiert. Sind Variablen nach der Ausführung auszugeben, werden sie mit dem Schlüsselwort `export` markiert.

Nach dem `begin` Schlüsselwort folgen die Anweisungen des Skripts. Das Schlüsselwort `end story;` schließt das Skript ab.

⟨*story*⟩ ::= 'story' ⟨*identifier*⟩ 'is' [⟨*varlist*⟩] 'begin' ⟨*stmt_list*⟩ 'end story;'
⟨*varlist*⟩ ::= [⟨*varlist*⟩] ⟨*varspec*⟩
⟨*varspec*⟩ ::= ['import' | 'export'] ⟨*identifier*⟩ ':' ⟨*type_identifier*⟩ ';'
⟨*type_identifier*⟩ ::= 'Boolean' | 'String' | 'Integer' | 'Node' | 'Edge' | 'Path' | 'Node_Set' | 'Edge_Set' | 'Path_Set' | 'Relation'

⟨*identifier*⟩ ::= siehe Abschnitt 5.5.1
⟨*stmt_list*⟩ ::= siehe Abschnitt 5.3

5.2. Skript-Schnittstelle

Jedem Skript können eine Menge von Eingabe- und Ausgabeparametern zugewiesen werden. Dazu werden die im Skriptkopf angegebenen Variablen mit den Schlüsselwörtern `import` bzw `export` markiert, wobei eine Variable nicht gleichzeitig Ein- und Ausgabeparameter sein darf.

Eingabeparameter werden beim Start des Skripts mit den angegebenen Werten gefüllt. Diese können z.B. durch Auswertung von Ausdrücken oder auch durch Selektion von Knoten und Kanten in Gravis entstehen. Nach dem Start unterscheiden sich diese Variablen nicht von unmarkierten Variablen.

Bestimmte Typen wie Pfade können außerhalb eines Skripts nicht erzeugt werden, daher sind für Eingabeparameter folgende Typen zulässig: Integer, Boolean, Node, Edge, Node_Set, Edge_Set. Für Ausgabeparameter können folgende Typen verwendet werden: Integer, Boolean, String, Node, Edge, Path, Node_Set, Edge_Set, Relation, Path_Set.

5.3. Anweisungen

Im folgenden Abschnitt werden die von RFGSCRIPT unterstützten Anweisungen beschrieben. Diese erlauben Deklaration und Manipulation von Variablen sowie die Steuerung des Kontrollflusses.

⟨*stmt_list*⟩ ::= [⟨*stmt_list*⟩] ⟨*stmt*⟩
⟨*stmt*⟩ ::= ⟨*expr_stmt*⟩ | ⟨*assignment*⟩ | ⟨*declare*⟩ | ⟨*if*⟩ | ⟨*while*⟩ | ⟨*loop*⟩ | ⟨*break*⟩ | ⟨*match*⟩

5.3.1. Ausdruck

Ein Ausdruck kann auch als alleinstehende Anweisung verwendet werden. Dabei wird der Ausdruck ausgewertet und eventuelle Nebeneffekte ausgeführt. Das Ergebnis des Ausdrucks wird ignoriert. Durch diesen Mechanismus werden auch Funktionsaufrufe ausgeführt.

Der Aufbau eines Ausdrucks wird in Abschnitt 5.6 erläutert.

⟨*expr_stmt*⟩ ::= ⟨*expr*⟩ ';'

Beispiel. `17 + 4;` ∎

5.3.2. Zuweisung

Mittels der Zuweisungs-Anweisung wird einer Variable ein neuer Wert zugewiesen. Die Variable muss zum Zeitpunkt der Zuweisung deklariert sein und vom gleichen Typ wie der zuzuweisende Wert sein. Ist dies nicht der Fall, wird das Skript abgebrochen.

Wird eine Menge oder Relation zugewiesen, werden dabei Kopien aller enthaltenen Werte gemacht, so dass eine Änderung der ursprünglichen Menge keine Änderung der Kopie bewirkt.

⟨*assignment*⟩ ::= ⟨*identifier*⟩ ':=' ⟨*expr*⟩ ';'

Beispiel.
```
X := 2;
```

5.3.3. Deklaration neuer Variablen

Mit der `declare`-Anweisung können auch innerhalb eines Skripts neue Variablen deklariert werden. Die Variablen dürfen Namen von außerhalb deklarierten Variablen verwenden, um diese Variablen zu überdecken. Sie sind nur innerhalb der Deklarationsanweisung sichtbar und werden beim Verlassen der Anweisung gelöscht.

⟨*declare*⟩ ::= 'declare' ⟨*varlist*⟩ 'begin' ⟨*stmt_list*⟩ 'end declare' ';'

Beispiel.
```
declare
  X : Integer;
begin
  X := 5;
end declare;
```

5.3.4. Bedingung

Die `if`-Anweisung macht das Ausführen einer Anweisungsfolge von einer Bedingung abhängig.

Wenn die Auswertung des angegebenen Ausdrucks `True` ergibt, werden die Anweisungen im `then`-Abschnitt der Anweisung ausgeführt. Wenn die Auswertung des angegebenen Ausdrucks `False` ergibt und es einen `else`-Abschnitt gibt, werden die dort enthaltenen Anweisungen ausgeführt.

Danach wird die Ausführung mit der der `if`-Anweisung folgenden Anweisung fortgeführt.

⟨*if*⟩ ::= 'if' ⟨*expr*⟩ 'then' ⟨*stmt_list*⟩ ['else' ⟨*stmt_list*⟩] 'end if' ';'

Beispiel.
```
if X > 5 then
  Y := Y * 2;
else
  Y := Y - 2;
end if;
```
∎

5.3.5. Bedingte Schleife

Eine unbedingte Schleife wird so lange ausgeführt, wie die Auswertung des angegebenen Ausdrucks `True` ergibt. Ergibt die Auswertung `False`, wird die Skriptausführung mit der ersten Anweisung nach der Schleife fortgesetzt.

Der Ausdruck wird vor jedem Schleifendurchlauf neu ausgewertet.

⟨*while*⟩ ::= 'while' ⟨*expr*⟩ 'loop' ⟨*stmt_list*⟩ 'end loop' ';'.

Beispiel.
```
while X > 5 loop
  Y := Y * 2;
end loop;
```
∎

5.3.6. Unbedingte Schleife

Eine unbedingte Schleife wird so lange ausgeführt, bis eine `break`-Anweisung ausgeführt wird. Mit der `break`-Anweisung wird eine `loop`-Schleife abgebrochen. Die Skriptausführung wird mit der ersten Anweisung nach der Schleife fortgesetzt.

`break`-Anweisungen sind nur innerhalb einer Schleife erlaubt, wird sie außerhalb einer Schleife angetroffen, wird das Skript abgebrochen. Es wird nur die innenliegende Schleife abgebrochen.

⟨*loop*⟩ ::= 'loop' ⟨*stmt_list*⟩ 'end loop' ';'.
⟨*break*⟩ ::= 'break'.

Beispiel.
```
loop
  X := X + 1;
  if X > 5 then
    break;
  end if;
end loop;
```
∎

5.3.7. Teilgraphsuche

Mit der `match`-Anweisung wird eine kompaktere Syntax für die Suche nach bestimmten Teilgraphen realisiert.

Diese werden graphisch angegeben und bestehen aus folgenden Teilen:

- ein Start-Knoten-Variable, dem eine Knotenmenge zugewiesen wird,
- eine Menge von Knoten-, Kanten- und Pfad-Variablen (im folgenden *Match-Variablen* genannt) sowie für jede Variable ein Ausdruck, der für gewünschte Belegungen zu `True` evaluieren muss,
- ein zusammenhängender Teilgraph, der Match-Variablen enthält, mit denen die gewünschten Beziehungen der Knoten und Kanten angegeben werden,
- eine geordnete Liste, in der die Reihenfolge der Suche angegeben wird, sowie
- eine Anweisungsliste, die für alle gefundenen Teilgraphen ausgeführt wird.

Am Anfang einer Teilgraphsuche werden für den Start-Knoten alle Werte berechnet, die er potentiell annehmen kann. Nun werden vom Start-Knoten ausgehend entsprechend der vorgegebenen Reihenfolge passende Kanten, Pfade und Knoten gesucht.

Wird ein passender Teilgraph gefunden, wird die angegebene Anweisungsliste ausgeführt.

Der graphische Editor für `match`-Anweisungen wird in Abschnitt 5.8.4 beschrieben.

5.4. Datentypen

Die in RFGScript vorhandenen Datentypen können in folgende Gruppen unterschieden werden:

- einfache Typen wie Integer und String,
- Graphen-Elemente wie Node und Path,
- Relationen.

Die Definition neuer Datentypen ist nicht möglich.

5.4.1. Einfache Typen

Diese Typen wurden in die Sprache aufgenommen, um Objekte wie Mengenelemente zählen und vergleichen zu können und Meldungen anzeigen zu können. Sie bilden außerdem die Basis für logische Ausdrücke, die für Kontrollstrukturen wie `if` und `while` gebraucht werden.

Integer

Variablen vom Typ Integer enthalten vorzeichenbehaftete natürliche Zahlen. Der erlaubte Wertebereich ist gleich dem Wertebereich des Ada95-Typs *Integer*.

Verletzt ein Skript diesen Wertebereich, auch in Zwischenrechnungen, wird es abgebrochen.

Folgende Operatoren sind für Integer-Werte definiert:

`<`	prüft für zwei Zahlen a und b, ob $a < b$
`<=`	prüft für zwei Zahlen a und b, ob $a \leq b$
`=`	prüft für zwei Zahlen a und b, ob $a = b$
`>=`	prüft für zwei Zahlen a und b, ob $a \geq b$
`>`	prüft für zwei Zahlen a und b, ob $a > b$
`/=`	prüft für zwei Zahlen a und b, ob $a \neq b$
`+`	addiert zwei Zahlen
`-`	subtrahiert zwei Zahlen, bzw. gibt bei unärem $-$ $-a$ zurück
`*`	multipliziert zwei Zahlen
`/`	Ganzzahldivision

String

Variablen vom Typ String enthalten Listen von 8-Bit-Zeichen. Die Länge eines Strings ist nicht begrenzt.

Folgende Operatoren sind für String-Werte definiert:

`=`	prüft zwei Strings auf Gleichheit
`/=`	prüft zwei Strings auf Ungleichheit

Boolean

Variablen vom Typ Boolean haben entweder den Wert `True` oder `False`.

Folgende Operatoren sind für Boolean-Werte definiert:

`=`	prüft zwei Booleans auf Gleichheit
`/=`	prüft zwei Booleans auf Ungleichheit
`and`	berechnet $a \wedge b$ für zwei Boolean-Werte a und b
`or`	berechnet $a \vee b$ für zwei Boolean-Werte a und b
`xor`	berechnet $a \oplus b$ für zwei Boolean-Werte a und b
`not`	berechnet $\overline{a}$ für einen Boolean-Werte a

Operatorpriorität

Operatoren mit niedrigerer Priorität stehen weiter oben, Tokens sind mit | getrennt):

`and | or | xor`	logische Operatoren
`= | /= | < | <= | > | >=`	Vergleichsoperatoren
`+ | -`	binäre Additionsoperatoren
`-`	unäre Additionsoperatoren
`* | /`	Multiplikationsoperatoren
`not`	

Stehen zwei Operatoren der gleichen Kategorie nebeneinander, wird der linke Operand zuerst evaluiert. Eine andere Evaluierungsreihenfolge kann durch Klammerung erreicht werden.

5.4.2. Graph-Elemente

Diese Typen wurden in die Sprache aufgenommen, um mit Graph-Elementen arbeiten zu können.

Für keinen dieser Typen wird gespeichert, aus welcher im RFG vorhandenen View die referenzierten Objekte stammen.

Node

Variablen vom Typ Node enthalten eine Referenz auf einen in einem RFG existierenden Knoten.

Edge

Variablen vom Typ Edge enthalten eine Referenz auf eine in einem RFG existierende Kante. Dabei wird die Kante als eigenständiges Objekt referenziert, nicht nur der Start- und Endknoten.

Path

Variablen vom Typ Path enthalten eine Liste von alternierenden Knoten- und Kantenreferenzen. Diese stellen einen Pfad in einem RFG dar.

Pfade sind nicht darauf beschränkt, mit einem Knoten zu beginnen und zu enden, um eine programmatische Konstruktion des Pfads zu ermöglichen.

Nodeset

Variablen vom Typ Node_Set enthalten eine Menge von Knotenreferenzen.

Edgeset

Variablen vom Typ Edge_Set enthalten eine Menge von Kantenreferenzen.

Relation

Variablen vom Typ Relation enthalten eine Menge von Knotenpaar-Referenzen.

Pathset

Variablen vom Typ Path_Set enthalten eine Menge von Pfadreferenzen.

View

Variablen vom Typ View enthalten eine Referenz auf eine in einem RFG existierende View.

5.5. Reservierte Worte

Die Namen der im Kapitel 5.7 beschriebenen Funktionen, die Typnamen sowie folgende, in der Sprache selbst verwendeten Wörter können nicht als Bezeichner verwendet werden: `and`, `begin`, `break`, `declare`, `else`, `end`, `export`, `if`, `import`, `is`, `loop`, `match`, `not`, `or`, `script`, `story`, `then`, `while` und `xor`.

5.5.1. Bezeichner

Variablen und Funktionen teilen sich denselben Namensraum. Ihre Namen beginnen mit einem Buchstaben und können ansonsten aus Buchstaben, Ziffern und Unterstrichen bestehen. Reservierte Worte können nicht als Bezeichner verwendet werden.

⟨*letter*⟩ ::= 'A' | ... | 'Z' | 'a' | ... | 'z'
⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*number*⟩ | '_')*

Beispiel. `Print(What => "Beeblebrox");` ∎

5.6. Ausdrücke

Ausdrücke können aus Literalen, Operatoren und Funktionsaufrufen aufgebaut sein.

5.6.1. Literale

Boolesche Literale

Die zwei Literale „`True`"und „`False`" werden als Boolesche Literale mit dem Typ Boolean erkannt.

⟨*boolean_literal*⟩ ::= 'True' | 'False'

Beispiel. `X := True;` ∎

Numerische Literale

Es gibt nur den numerischen Typ Integer in RFGScript. Daher dürfen numerische Literale nur aus den Ziffern „`0`" - „`9`" sowie optional einem führenden „−" bestehen.

⟨*numeric_literal*⟩ ::= ['-'] ⟨*digits*⟩
⟨*digits*⟩ ::= [⟨*digits*⟩] ⟨*digit*⟩
⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Beispiel. `N := 42;` ∎

Stringliterale

Stringliterale können aus 8-Bit-Zeichen, allerdings ohne das doppelten Anführungszeichen, bestehen und müssen in doppelten Anführungszeichen eingeschlossen sein.

⟨*string_literal*⟩ ::= '"' ⟨*character_list*⟩ '"'
⟨*character_list*⟩ ::= [⟨*character_list*⟩] ⟨*character*⟩
⟨*character*⟩ ::= 8-Bit-Zeichen ohne '"'

Beispiel. `S := "Zaphod";` ∎

5.6.2. Funktionsaufrufe

Funktionen werden über ihren Bezeichner aufgerufen. Alle Parameter einer Funktion sind benamt. Diese Parameternamen dürfen keine reservierten Worte sein, was bei den eingebauten Funktionen sichergestellt ist.

⟨*function_call*⟩ ::= ⟨*identifier*⟩ ['(' ⟨*arg_list*⟩ ')']
⟨*arg_list*⟩ ::= [⟨*arg_list*⟩ ','] ⟨*arg*⟩
⟨*arg*⟩ ::= ⟨*identifier*⟩ '=>' ⟨*expr*⟩

Beispiel. `Print(What => "Beeblebrox");` ∎

5.6.3. Operatoren

Nicht jeder Operator ist für alle verfügbaren Datentypen definiert.

Im folgenden Abschnitt werden die definierten Operatoren und ihre Semantik beschrieben.

Logische Operatoren *and, or, xor, not*

Diese Operatoren lassen sich auf boolesche Werte anwenden und liefern auch wieder boolesche Werte zurück.

Vergleichsoperatoren

Die Vergleichsoperatoren „<“, „<=“, „>“, „>=“, „=“ und „/=“ lassen sich auf numerische Werte anwenden und liefern boolesche Werte zurück. Die Vergleichsoperatoren „=“ und „/=“ können dazu noch auf Strings, Booleans, Nodes, Edges, Paths sowie Mengen und Relationen angewandt werden.

5.7. Funktionen

Zur Modularisierung von Skripten und zur Erleichterung von Wiederverwendung ist es sinnvoll, die Definition von Funktionen zu ermöglichen. In RFGScript wäre die Einführung einer speziellen Syntax zur Definition von Funktionen oder den Aufruf von anderen Skripten über ihren Namen denkbar. Auch der Aufruf von in Ada95 implementierten Analysen wäre wünschenswert. Aus Zeitgründen wird dies zunächst nicht realisiert.

Es gibt jedoch eine Anzahl vordefinierter Funktionen, die in Skripten und Ausdrücken zur Verfügung stehen. Im folgenden werden diese Funktionen und die Typen, auf denen sie arbeiten, dargestellt.

Da es momentan keine spezifizierte Grammatik für die Definition von Funktionen in RFGScript gibt, werden die Signaturen mit mathematischen Symbolen angegeben, wobei n für einen Knoten, N für eine Knotenmenge, e für eine Kante, E für eine Kantenmenge, p für einen Pfad, P für eine Pfadmenge, R für eine Relation, S für einen String, B für einen Booleschen Wert und I für einen Integer-Wert verwendet wird. Da in RFGScript nicht die Position des Arguments sondern sein Name relevant ist, wird dieser zusätzlich tiefgestellt angegeben.

Die Signatur „Append : $N_{To} \times n_{What} \rightarrow N$“ definiert somit eine Funktion Append mit den Argumenten To vom Typ Knotenmenge, What vom Typ Knoten und dem Ergebnistyp Knotenmenge.

Append

Mit Append können Elemente in eine Menge eingefügt werden. Dabei muss der Elementtyp zum Typ der Menge passen.

Append : $N_{\text{To}} \times n_{\text{What}} \rightarrow N$

Append : $E_{\text{To}} \times e_{\text{What}} \rightarrow E$

Append : $P_{\text{To}} \times p_{\text{What}} \rightarrow P$

Da die zu bearbeitenden Mengen potentiell sehr groß werden, verändert die Append-Funktion die als Argument übergebene Menge direkt, ohne eine Kopie anzulegen.

Die Append-Funktion ist auch für Pfade definiert. Sie fügt die angegebene Kante oder den Knoten an das Ende des Pfads an. Dabei ist zu beachten, dass auf einen Knoten kein weiterer Knoten folgen darf, genauso darf auf eine Kante keine weitere Kante folgen. Des Weiteren muss ein im Pfad vor einer Kante stehender Knoten auch tatsächlich der Quellknoten der Kante im RFG sein, genauso muss ein im Pfad nach einer Kante stehender Knoten der Zielknoten der Kante sein. Da der Zustand des Pfads erst zur Laufzeit bekannt ist, kann vor der Ausführung des Skripts nicht entschieden

werden, ob der Typ des anzufügenden Elements korrekt ist. Ist der Typ nicht korrekt, oder passt die Kante nicht zu den anliegenden Knoten, wird das Skript abgebrochen.

Append : $p_{\text{To}} \times n_{\text{What}} \rightarrow p$

Append : $p_{\text{To}} \times e_{\text{What}} \rightarrow p$

Aus Konsistenzgründen verändert die Append-Funktion auch bei Pfaden direkt den als Argument übergebenen Pfad.

Beispiel. `Append (To => Path, What => E);` ∎

At_Pos

Gibt das an der angegebenen Position vorhandene Objekt des Pfads zurück.

Ist die angegebene Position grösser als die Länge des Pfads, wird das Skript abgebrochen.

Das erste Element des Pfads hat die Position 1.

At_Pos : $p_{\text{Of}} \times I_{\text{Pos}} \rightarrow e$

At_Pos : $p_{\text{Of}} \times I_{\text{Pos}} \rightarrow n$

Da der Zustand des Pfads erst zur Laufzeit bekannt ist, kann vor der Ausführung des Skripts der Ergebnistyp der Funktion nicht bestimmt werden.

Beispiel. `N := At_Pos (Of => Path, Pos => 3);` ∎

Closure

Die Closure-Funktion berechnet den transitiven Abschluß einer Kantenmenge oder Relation. Da dabei Kanten entstehen können, die nicht im RFG vorhanden sind, wird das Ergebnis als Relation zurückgegeben. Mit der Realize-Funktion können diese Kanten dann im RFG erzeugt werden.

Closure : $E_{\text{Of}} \rightarrow R$

Closure : $R_{\text{Of}} \rightarrow R$

Beispiel. `C := Closure (Of => R);` ∎

Complement

Die Complement-Funktion berechnet das Komplement $(1 - R)$ einer Kantenmenge oder Relation. Dazu werden alle im RFG vorhandenen Kanten einbezogen.

Complement : $E_{\text{Of}} \rightarrow R$

Complement : $R_{\text{Of}} \to R$

Beispiel. `C := Complement (Of => R);` ∎

Composition

Die Composition-Funktion berechnet die relationale Komposition ($(a,b) \in R_1, (b,c) \in R_2 \to (a,c)$) zweier Relationen oder Kantenmengen.

Composition : $R_{\text{L}} \times R_{\text{R}} \to R$

Composition : $R_{\text{L}} \times E_{\text{R}} \to R$

Composition : $E_{\text{L}} \times R_{\text{R}} \to R$

Composition : $E_{\text{L}} \times E_{\text{R}} \to R$

Beispiel. `C := Composition (L => R1, R => R2);` ∎

Concat

Die Concat-Funktion verkettet zwei Pfade miteinander und fügt optional entweder einen Knoten oder eine Kante ein. Der resultierende Pfad darf die bei der Append-Funktion beschriebenen Bedingungen für gültige Pfade nicht verletzen. Da der Zustand der Pfade erst zur Laufzeit bekannt ist, wird das Skript im Fall einer Verletzung abgebrochen.

Concat : $p_{\text{Head}} \times p_{\text{Tail}} \to p$

Concat : $p_{\text{Head}} \times n_{\text{Node}} \times p_{\text{Tail}} \to p$

Concat : $p_{\text{Head}} \times e_{\text{Edge}} \times p_{\text{Tail}} \to p$

Beispiel. `P := Concat (Head => P1, Edge => E, Tail => P2);` ∎

Converse

Die Converse-Funktion berechnet die Umkehrung ($(a,b) \to (b,a)$) einer Relation oder Kantenmenge.

Converse : $R_{\text{Of}} \to R$

Converse : $E_{\text{Of}} \to R$

Beispiel. `C := Converse (Of => R);` ∎

Create_Edge

Erzeugt eine neue Kante im RFG.

Create_Edge : $n_{\text{Source}} \times n_{\text{Target}} \times S_{\text{Type}} \rightarrow e$

Beispiel.
```
E := Create_Edge (Type   => "Routine",
                  Source => N1,
                  Target => N2);
```
∎

Create_Node

Erzeugt einen neuen Knoten im RFG.

Create_Node : $S_{\text{Type}} \rightarrow n$

Beispiel.
```
N := Create_Node (Type => "Routine");
```
∎

Cross

Die Cross-Funktion berechnet das Kreuzprodukt zweier Mengen, d.h. die erzeugte Relation besteht aus der Kombination jedes Knotens der ersten Menge mit jedem Knoten der zweiten Menge.

Cross : $N_{\text{L}} \times N_{\text{R}} \rightarrow R$

Beispiel.
```
C := Cross (L => Nodes (View => "BASE"),
            R => Nodes (View => "CALL"));
```
∎

Delete_Edge

Löscht eine Kante im RFG.

Delete_Edge : $e_{\text{What}} \rightarrow B$

Beispiel.
```
Delete_Edge (What => E);
```
∎

Delete_Node

Löscht einen Knoten im RFG.

Delete_Node : $S_{\text{What}} \rightarrow B$

Beispiel.
```
Delete_Node (What => N);
```
∎

Difference

Die Difference-Funktion berechnet die Differenz $D = L - R$ zweier Mengen L und R. Sie kann auf Knoten-, Kanten- und Pfadmengen sowie Relationen angewandt werden und liefert ein Ergebnis gleichen Typs zurück.

Difference : $N_{\mathrm{L}} \times N_{\mathrm{R}} \to N$

Difference : $E_{\mathrm{L}} \times E_{\mathrm{R}} \to E$

Difference : $P_{\mathrm{L}} \times P_{\mathrm{R}} \to P$

Difference : $R_{\mathrm{L}} \times R_{\mathrm{R}} \to R$

Difference : $R_{\mathrm{L}} \times E_{\mathrm{R}} \to R$

Difference : $E_{\mathrm{L}} \times R_{\mathrm{R}} \to R$

Beispiel.
```
D := Difference (L => Nodes (View => "BASE"),
                 R => Nodes (View => "CALL")); ▮
```

Domain

Die Domain-Funktion gibt die Startknoten der übergebenen Kantenmenge oder Relation als Knotenmenge zurück.

Domain : $E_{\mathrm{Of}} \to N$

Domain : $R_{\mathrm{Of}} \to N$

Beispiel.
```
N := Domain (Of => Edges); ▮
```

Edges

Die Edges-Funktion erzeugt eine Kantenliste aus den im RFG vorhandenen Kanten und gibt sie zurück.

Ist ein Typname angegeben, werden nur Kanten zurückgegeben, die den angegebenen Typ haben. Ist ein Viewname angegeben, werden nur Kanten zurückgegeben, die in der angegebenen View sichtbar sind.

Edges : $\to E$

Edges : $S_{\mathrm{View}} \to E$

Edges : $S_{\mathrm{Type}} \to E$

Edges : $S_{\mathrm{Type}} \times S_{\mathrm{View}} \to E$

Beispiel.
```
E := Edges;
C := Edges (View => "CALL");
D := Edges (Type => "Call"); ▮
```

Empty_Edge_Set

Gibt eine leere Kantenmenge als Ergebnis zurück.

Empty_Edge_Set : $\rightarrow E$

Empty_Node_Set

Gibt eine leere Knotenmenge als Ergebnis zurück.

Empty_Node_Set : $\rightarrow N$

Empty_Path

Gibt einen leeren Pfad als Ergebnis zurück.

Empty_Path : $\rightarrow p$

Empty_Path_Set

Gibt eine leere Pfadmenge als Ergebnis zurück.

Empty_Path_Set : $\rightarrow P$

Empty_Relation

Gibt eine leere Relation als Ergebnis zurück.

Empty_Relation : $\rightarrow R$

Get_Attr

Gibt den Inhalt des angegeben Attributs eines Knotens oder einer Kante zurück. Da sich der Typ eines Attributs je nach Schema des RFGs ändert, steht der Rückgabetyp der Get_Attr-Funktion erst zur Laufzeit fest.

Get_Attr : $n_{\text{Of}} \times S_{\text{Name}} \rightarrow S$

Get_Attr : $n_{\text{Of}} \times S_{\text{Name}} \rightarrow I$

Get_Attr : $n_{\text{Of}} \times S_{\text{Name}} \rightarrow B$

Get_Attr : $e_{\text{Of}} \times S_{\text{Name}} \rightarrow S$

Get_Attr : $e_{\text{Of}} \times S_{\text{Name}} \rightarrow I$

Get_Attr : $e_{\text{Of}} \times S_{\text{Name}} \rightarrow B$

Beispiel. `Filename := Get_Attr (Of => N, Name => "File_Name");` ∎

Gravis_Mark

Diese Funktion markiert alle übergebenen Elemente in Gravis und gibt sie wieder zurück.

Gravis_Mark : $e_{\text{What}} \rightarrow e$

Gravis_Mark : $n_{\text{What}} \rightarrow n$

Gravis_Mark : $p_{\text{What}} \rightarrow p$

Gravis_Mark : $E_{\text{What}} \rightarrow E$

Gravis_Mark : $N_{\text{What}} \rightarrow N$

Gravis_Mark : $P_{\text{What}} \rightarrow P$

Beispiel. `Gravis_Mark (What => Nodes (Type => "Routine"));` ∎

Head

Entfernt das erste Objekt der als Argument angegebenen Knoten-, Kanten- oder Pfadmenge, Relation oder Pfad und gibt es zurück.

Head : $p_{\text{Of}} \rightarrow n$

Head : $p_{\text{Of}} \rightarrow e$

Da Mengen und Relationen keine Ordnung haben, ist dort nicht definiert, welches Element zurückgegeben wird.

Head : $N_{\text{Of}} \rightarrow n$

Head : $E_{\text{Of}} \rightarrow e$

Head : $P_{\text{Of}} \rightarrow p$

Da es für eine 1-elementige Relation keinen eigenen Datentyp gibt, wird das Element als Relation zurückgegeben.

Head : $R_{\text{Of}} \rightarrow R$

Beispiel. `N := Head (Of => Nodes);` ∎

Head_Is_Node

Gibt `True` zurück, wenn der angegebene Pfad mit einem Knoten beginnt.

Head_Is_True : $p_{\text{Of}} \rightarrow B$

Beispiel.
```
if Head_Is_Node (Of => Path) then
   N := Head (Of => Path);
else
   E := Head (Of => Path);
end if;
```
∎

Identity

Die Identity-Funktion erzeugt aus einer Knotenmenge die Identitätsrelation.

Identity : $N_{\text{Of}} \rightarrow R$

Beispiel. `R := Identity (Of => Nodes);` ∎

Incoming_Edges

Gibt die eingehenden Kanten des übergebenen Knoten zurück.

Incoming_Edges : $n_{\text{Of}} \rightarrow E$

Beispiel. `E := Incoming_Edges (Of => N);` ∎

Intersection

Die Intersection-Funktion berechnet den Schnitt $I = L \cap R$ zweier Mengen L und R. Sie kann auf Knoten-, Kanten- und Pfadmengen sowie Relationen angewandt werden und liefert ein Ergebnis gleichen Typs zurück.

Intersection : $N_{\text{L}} \times N_{\text{R}} \rightarrow N$

Intersection : $E_{\text{L}} \times E_{\text{R}} \rightarrow E$

Intersection : $P_{\text{L}} \times P_{\text{R}} \rightarrow P$

Intersection : $R_{\text{L}} \times R_{\text{R}} \rightarrow R$

Intersection : $R_{\text{L}} \times E_{\text{R}} \rightarrow R$

Intersection : $E_{\text{L}} \times R_{\text{R}} \rightarrow R$

Beispiel.
```
D := Intersection (L => Nodes (View => "CALL"),
                   R => Nodes (View => "FILE"));
```
∎

Is_In

Die Is_In-Funktion gibt `True` zurück, wenn das angegebene Element in der angegebenen Menge enthalten ist. Ansonsten wird `False` zurückgegeben.

Is_In : $n_{\text{What}} \times N_{\text{In}} \rightarrow B$

Is_In : $e_{\text{What}} \times E_{\text{In}} \rightarrow B$

Is_In : $p_{\text{What}} \times P_{\text{In}} \rightarrow B$

Is_In : $e_{\text{What}} \times R_{\text{In}} \rightarrow B$

Beispiel.
```
X := Is_In (What => N, In => M);
```
▮

Latent

Die Latent-Funktion extrahiert die Knotenpaare aus einer Relation, für die es keine Kante im RFG gibt, und gibt sie als Relation zurück.

Latent : $R_{\text{In}} \rightarrow R$

Ist ein Typnamen angegeben, werden nur Kanten überprüft, die den angegebenen Typ haben. Ist ein Viewnamen angegeben, werden nur Kanten überprüft, die in der angegebenen View sichtbar sind.

Latent : $R_{\text{In}} \times S_{\text{Type}} \rightarrow R$

Latent : $R_{\text{In}} \times S_{\text{View}} \rightarrow R$

Latent : $R_{\text{In}} \times S_{\text{Type}} \times S_{\text{View}} \rightarrow R$

Beispiel.
```
L := Latent (In => R, Type => "Call");
```
▮

Make_Invisible

Die Make_Invisible-Funktion macht die übergebenen Knoten oder Kanten in der angegebenen View unsichtbar.

Make_Invisible : $N_{\text{What}} \times S_{\text{View}} \rightarrow N$

Make_Invisible : $E_{\text{What}} \times S_{\text{View}} \rightarrow E$

Beispiel.
```
M := Make_Invisible (What => N,
                     View => "Call");
```
▮

Make_Visible

Die Make_Visible-Funktion macht die übergebenen Knoten oder Kanten in der angegebenen View sichtbar.

Make_Visible : $N_{\text{What}} \times S_{\text{View}} \rightarrow N$

Make_Visible : $E_{\text{What}} \times S_{\text{View}} \rightarrow E$

Beispiel.
```
M := Make_Visible (What => N,
                   View => "Call"); ∎
```

Node_At_Pos

Gibt `True` zurück, wenn der angegebene Pfad an der angegebenen Stelle einen Knoten beinhaltet.

Ist der Pfad kürzer als die angegebene Position, wird das Skript abgebrochen.

Das erste Element des Pfads hat die Position 1.

Node_At_Pos : $p_{\text{Of}} \times I_{\text{Pos}} \rightarrow B$

Beispiel.
```
if Node_At_Pos (Of => Path, Pos => 2) then
   ...
end if;                                    ∎
```

Nodes

Die Nodes-Funktion erzeugt eine Knotenliste aus den im RFG vorhandenen Knoten und gibt sie zurück.

Ist ein Typname angegeben, werden nur Knoten zurückgegeben, die den angegebenen Typ haben. Ist ein Viewname angegeben, werden nur Knoten zurückgegeben, die in der angegebenen View sichtbar sind.

Nodes : $\rightarrow B$

Nodes : $S_{\text{View}} \rightarrow N$

Nodes : $S_{\text{Type}} \rightarrow N$

Nodes : $S_{\text{View}} \times S_{\text{Type}} \rightarrow N$

Beispiel.
```
N := Nodes;
C := Nodes (View => "CALL"); ∎
```

Outgoing_Edges

Gibt die ausgehenden Kanten des übergebenen Knoten zurück.

Outgoing_Edges : $n_{\text{Of}} \rightarrow E$

Beispiel.
```
E := Outgoing_Edges (Of => N); ∎
```

Prepend

Fügt die angegebene Kante oder Knoten am Anfang des Pfads an.

Es gelten die bei der Append-Funktion beschriebenen Konsistenzbedingungen für Pfade.

Prepend : $p_{\text{To}} \times n_{\text{What}} \rightarrow p$

Prepend : $p_{\text{To}} \times e_{\text{What}} \rightarrow p$

Aus Gründen der Konsistenz mit der Append-Funktion verändert die Prepend-Funktion bei Pfaden direkt den als Argument übergebenen Pfad.

Beispiel.
```
Prepend (To => Path, What => N);
```
▮

Project

Die Project-Funktion projeziert eine Knotenmenge durch eine Relation.

Project : $N_{\text{What}} \times R_{\text{Through}} \rightarrow N$

Beispiel.
```
M := Project (What    => Nodes (View => "CALL"),
              Through => Edges (View => "CALL"));
```
▮

Project_Backwards

Die Project-Funktion projeziert eine Knotenmenge rückwärts durch eine Relation.

Project_Backwards : $N_{\text{What}} \times R_{\text{Through}} \rightarrow N$

Beispiel.
```
M := Project_Backwards (What    => Nodes (View => "CALL"),
                        Through => Edges (View => "CALL"));
```
▮

Range

Die Range-Funktion gibt die Endknoten der übergebenen Kantenmenge oder Relation als Knotenmenge zurück.

Range : $E_{\text{Of}} \rightarrow N$

Range : $R_{\text{Of}} \rightarrow N$

Beispiel.
```
N := Range (Of => Edges);
```
▮

Realize

Die Realize-Funktion erzeugt aus der angegebenen Relation Kanten im RFG und gibt sie zurück. Existiert schon eine Kanten mit dem angegebenen Typ, wird diese zurückgegeben. Die Kanten werden automatisch in der Base_View sichtbar, in anderen Views müssen sie extra sichtbar gemacht werden.

Realize : $R_{\text{What}} \times S_{\text{Type}} \rightarrow E$

Beispiel.
```
E := Realize (What => R, Type => "Call");
```

Select

Die Select-Funktion gibt die Teilmenge der übergebenen Knoten- oder Kantenmenge zurück, die genau die Knoten oder Kanten enthält, deren Attribut den angegebenen Wert enthält.

Select : $N_{\text{From}} \times S_{\text{Where}} \times S_{\text{Is}} \rightarrow N$

Select : $N_{\text{From}} \times S_{\text{Where}} \times I_{\text{Is}} \rightarrow N$

Select : $N_{\text{From}} \times S_{\text{Where}} \times B_{\text{Is}} \rightarrow N$

Select : $E_{\text{From}} \times S_{\text{Where}} \times S_{\text{Is}} \rightarrow E$

Select : $E_{\text{From}} \times S_{\text{Where}} \times I_{\text{Is}} \rightarrow E$

Select : $E_{\text{From}} \times S_{\text{Where}} \times B_{\text{Is}} \rightarrow E$

Beispiel.
```
N := Select (From  => Nodes,
             Where => "File_Name",
             Is    => "main.c");
```

Set_Attr

Setzt den Inhalt des angegeben Attributs.

Set_Attr : $n_{\text{Of}} \times S_{\text{Name}} \times S_{\text{To}} \rightarrow n$

Set_Attr : $n_{\text{Of}} \times S_{\text{Name}} \times I_{\text{To}} \rightarrow n$

Set_Attr : $n_{\text{Of}} \times S_{\text{Name}} \times B_{\text{To}} \rightarrow n$

Set_Attr : $e_{\text{Of}} \times S_{\text{Name}} \times S_{\text{To}} \rightarrow e$

Set_Attr : $e_{\text{Of}} \times S_{\text{Name}} \times I_{\text{To}} \rightarrow e$

Set_Attr : $e_{\text{Of}} \times S_{\text{Name}} \times B_{\text{To}} \rightarrow e$

Beispiel.
```
N := Set_Attr (Of   => N,
               Name => "Object_Name",
               To   => "Better");
```

Size

Die Size-Funktion gibt die Größe der übergebenen Menge oder Relation oder die Länge des Pfads als Integer zurück.

Size : $N_{\text{Of}} \rightarrow I$

Size : $E_{\text{Of}} \rightarrow I$

Size : $P_{\text{Of}} \rightarrow I$

Size : $R_{\text{Of}} \rightarrow I$

Size : $p_{\text{Of}} \rightarrow I$

Beispiel.
```
S := Size (Of => Nodes);
T := Size (Of => P);
```
∎

Source

Gibt den Startknoten der übergebenen Kante oder 1-elementiger Relation zurück.

Source : $e_{\text{Of}} \rightarrow n$

Source : $R_{\text{Of}} \rightarrow n$

Beispiel.
```
N := Source (Of => E);
```
∎

Tail

Entfernt das letzte Objekt des Pfades und gibt es zurück.

Tail : $p_{\text{Of}} \rightarrow n$

Tail : $p_{\text{Of}} \rightarrow e$

Beispiel.
```
N := Tail (Of => Path);
```
∎

Tail_Is_Node

Gibt `True` zurück, wenn der angegebene Pfad mit einem Knoten endet.

Tail_Is_Node : $p_{\text{Of}} \rightarrow B$

Beispiel.
```
if Tail_Is_Node (Of => Path) then
   ...
end if;
```
∎

Target

Gibt den Endknoten der übergebenen Kante oder 1-elementiger Relation zurück.

Target : $e_{\text{Of}} \rightarrow n$

Target : $R_{\text{Of}} \rightarrow n$

Beispiel. `N := Target (Of => E);` ∎

Type

Gibt den Typ des Elements als String zurück.

Type : $n_{\text{Of}} \rightarrow S$

Type : $e_{\text{Of}} \rightarrow S$

Beispiel. `S := Type (Of => E);` ∎

Union

Die Union-Funktion berechnet die Vereinigung $U = L \cup R$ zweier Mengen L und R. Sie kann auf Knoten-, Kanten- und Pfadmengen sowie Relationen angewandt werden und liefert ein Ergebnis gleichen Typs zurück.

Union : $N_{\text{L}} \times N_{\text{R}} \rightarrow N$

Union : $E_{\text{L}} \times E_{\text{R}} \rightarrow E$

Union : $P_{\text{L}} \times P_{\text{R}} \rightarrow P$

Union : $R_{\text{L}} \times R_{\text{R}} \rightarrow R$

Union : $R_{\text{L}} \times E_{\text{R}} \rightarrow R$

Union : $E_{\text{L}} \times R_{\text{R}} \rightarrow R$

Beispiel.
```
D := Union (L => Nodes (View => "CALL"),
            R => Nodes (View => "FILE"));
```
∎

5.8. Benutzerschnittstelle

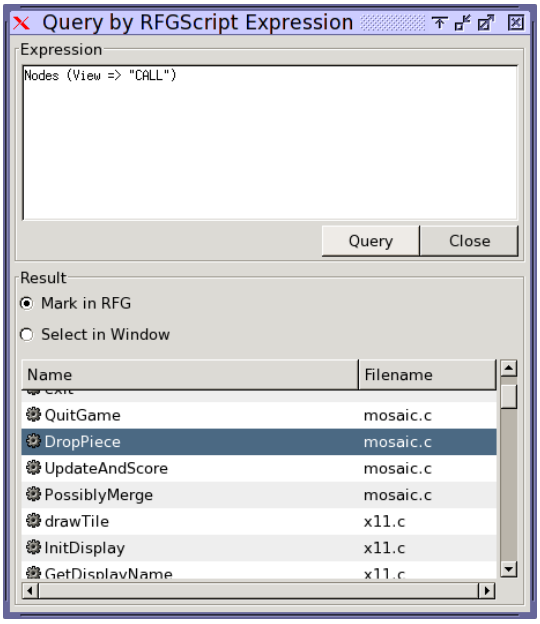
Dieser Abschnitt beschreibt die Benutzerschnittstelle, die von Gravis für den Gebrauch von RFGScript angeboten wird.

5.8.1. Einstieg

Folgende Einstiegspunkte für die Ausführung von RFGScripts sind vorgesehen:

- Query Expression. Dieses Dialogfenster erlaubt die Eingabe eines Ausdrucks, der auf Knopfdruck evaluiert wird, ohne dass ein komplettes Skript geschrieben werden muss. Das Ergebnis wird in Form von Text, bzw. einer Knoten- oder Kantenliste ausgegeben.
- Query Script. Dieses Dialogfenster erlaubt die Auswahl eines Skripts, das editiert oder gestartet werden kann.

Abbildung 5.1 „Query by RFGScript Expression“-Dialog



5.8.2. „Query by RFGScript Expression“-Dialog

Das Dialogfenster ist in ein Eingabefeld im oberen Bereich und eine Ausgabetabelle im unteren Bereich aufgeteilt (siehe Abbildung 5.1).

In das Eingabefeld können beliebige Ausdrücke eingegeben werden. Durch anklicken des Query-Buttons wird der Ausdruck evaluiert und das Ergebnis in der Ausgabetabelle angegeben.

Besteht das Ergebnis aus Knoten, Kanten oder Pfaden, können diese im Grapheneditor durch Markierung im gesamten RFG oder Selektion in einem Fenster visualisiert werden. Dazu wird zunächst die gewünschte Aktion ausgewählt und dann mit einem Doppelklick die in der Tabelle selektierten Elemente entsprechend visualisiert.

5.8.3. „Query by RFGScript"-Dialog

Das Dialogfenster des „Query by RFGScript Expression"-Dialog (siehe Abbildung 5.2) ist folgendermaßen aufgebaut:

- Zuoberst kann das auszuführende Skript ausgewählt werden. Es besteht auch die Möglichkeit, das aktuelle Skript im Editor zu öffnen oder ein neues Skript anzulegen.

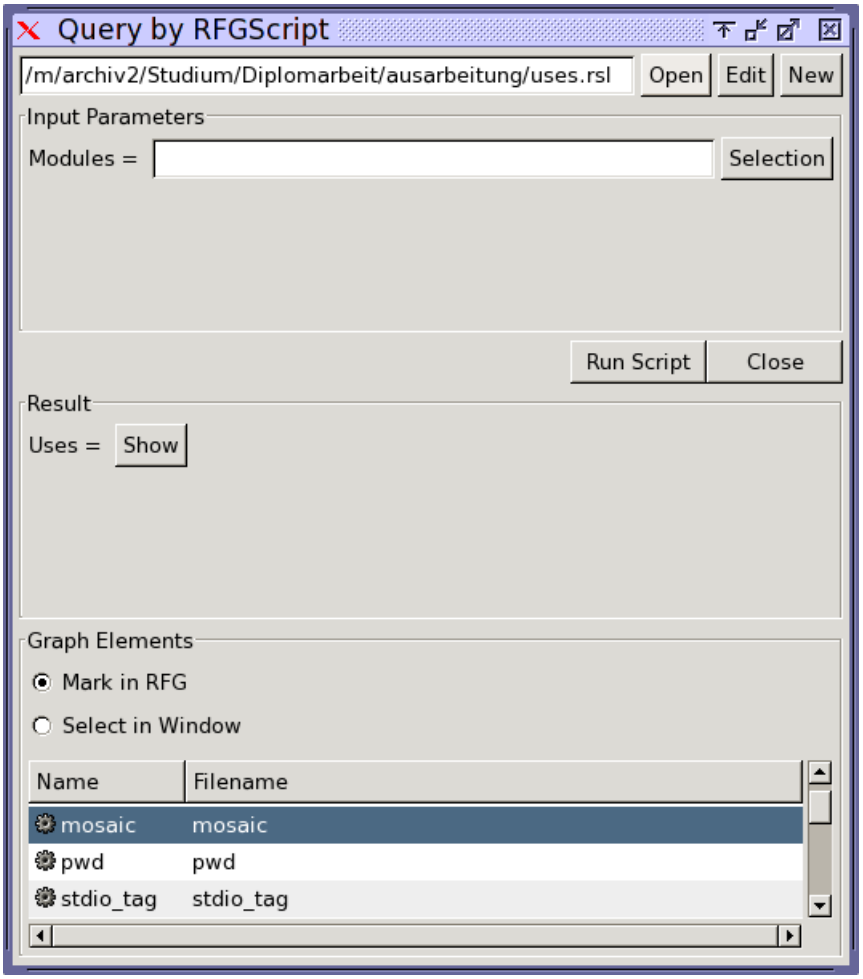
- In der darunterliegenden Tabelle werden die Eingabeparameter des aktuellen Skripts angezeigt. Es können beliebige Ausdrücke angegeben werden. Ist der Eingabeparameter vom Typ Knoten oder Kante, Knotenmenge, Kantenmenge oder Relation, können die momentan in einem Fenster selektierten Elemente durch Klick auf den Selection-Button als Wert für den Parameter übernommen werden.

- Mit dem „Run Script"-Button wird das Skript ausgeführt. Der Close-Button schliesst das Dialogfenster.

- Nach der erfolgreichen Ausführung des Skripts werden unterhalb der Buttons die Ergebnisse in einer Tabelle angezeigt. Ist ein Ergebniswert vom Typ Knoten oder Kante, Knotenmenge oder Kantenmenge, wird durch Klick auf den Show-Button in der untersten Tabelle der Wert ausführlicher dargestellt.

- Die Bedienung der ausführlichen Darstellung eines Werts geschieht analog zum Ergebnisfeld des „Query by RFGScript Expression"-Dialog im vorigen Abschnitt.

Abbildung 5.2 „Query by RFGScript“-Dialog



5.8.4. Skript-Editor

Der Skript-Editor (siehe Abbildung 5.3) ermöglicht die Änderung des Skript-Quelltexts. Er ermöglicht ausserdem das Drucken und Ausführen des aktuellen Skripts.

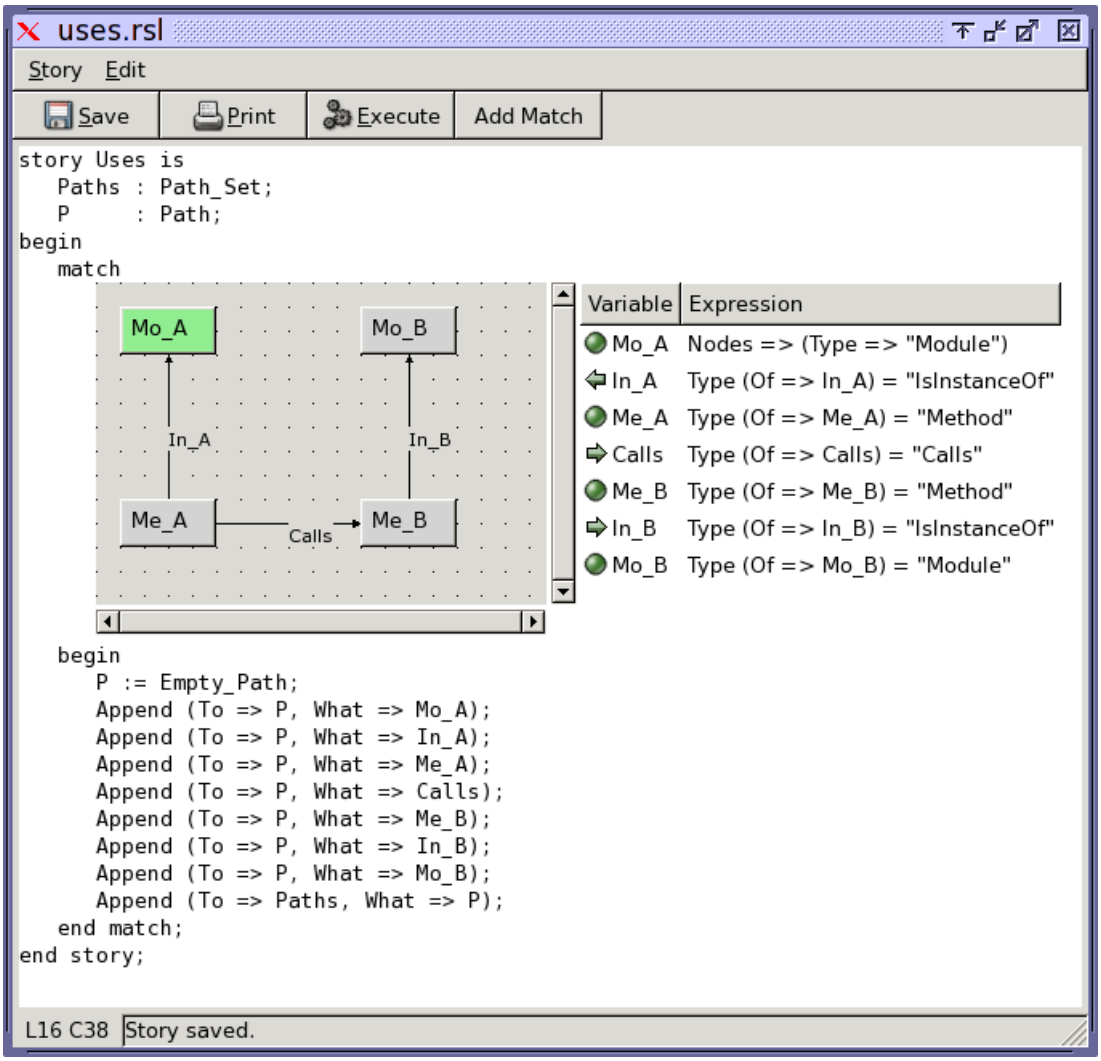
Der Vorteil des Skript-Editors gegenüber anderen Editoren ist die integrierte graphische Manipulationsmöglichkeit für Match-Anweisungen. Diese werden durck Klick auf den „Add Match“-Button an der gewünschten Stelle in den Text integriert.

Im linken Teil der graphischen Match-Anweisung wird der gesuchte Teilgraph angegeben. Knoten und Kanten können durch das Kontextmenü hinzugefügt, geändert und gelöscht werden.

Jedes hinzugefügte Element erscheint im rechten Teil in der Reihenfolgeliste. In die-

ser Liste werden die Variablennamen, die Belegung für den Startknoten sowie die Prädikate für Match-Variablen angegeben. Die Reihenfolge der Elemente ist per Drag-and-Drop änderbar. Ungültige Reihenfolgen werden vom Editor abgefangen.

Abbildung 5.3 Skript-Editor



6. Entwurf und Implementierung

In diesem Kapitel wird der Entwurf des Intepreters und dessen Implementierung beschrieben. Zu Anfang werden verschiedene Realisierungsalternativen für den Interpreter sowie die Art der Editierung diskutiert. Anschließend werden die beim Ablauf eines Skripts stattfindenden Vorgänge und die dafür gewählten Realisierungen dargestellt.

Abschließend wird auf den Test des Interpreters eingegangen.

6.1. Realisierungsalternativen

Aufgrund der ursprünglichen Aufgabe stand die Erweiterung einer vorhandenen Skriptsprache sowie die komplette Eigenentwicklung einer Sprache zur Wahl. Festi [4] beschreibt die Einbettung des Python-Interpreters in Gravis. Die graphischen Elemente der Skriptspezifikation und die freiere Auswahl der Datenstrukturen legten jedoch die Eigenentwicklung eines Interpreters nahe.

Ein erster Ansatz sah das Parsen des Quelltexts in eine zur Bearbeitung und Ausführung geeignete Datenstruktur vor, so dass der Anwender den rohen Quelltext gar nicht zu Gesicht bekommt. Dies hätte den Vorteil, dass Konsistenzprüfungen sofort vorgenommen werden können und ungültige Konstrukte gar nicht erzeugt werden können. Allerdings stellt sich damit das Problem der geeigneten Visualisierung des Skripts auf dem Bildschirm und auf Papier. Ein weiteres Problem war die fehlende Vertrautheit und vermutete Umständlichkeit eines vorwiegend graphischen Ansatzes, das zu Aktzeptanzproblemen vor allem bei erfahrenen Anwendern führt.

Daher wurde als Alternative eine Text-zentrierte Manipulation des Quelltexts gewählt, in die nur bei Bedarf graphische Elemente eingebunden werden. Dies hat allerdings die Auswirkung, dass die Konsistenz des Quellcodes nicht während des Editierens sichergestellt werden kann. Daher wird das Skript vor der Ausführung auf Konsistenzprobleme wie z.B. falsche Typen untersucht.

Um auch die Möglichkeit des zuerst genannten Ansatzes offen zu halten, wurde der Entwurf so ausgelegt, dass die vom Parser erzeugten Datenstrukturen trotzdem noch ohne großen Aufwand programmatisch geändert werden könnten.

6.2. Scanner und Parser

Es stand zur Wahl, einen geeigneten Scanner und Parser selbst zu implementieren. Da im Bauhaus-Projekt allerdings schon für Ada95 geeignete Scanner- und Parsergeneratoren eingesetzt werden, lag deren Benutzung nahe. Es handelt sich bei ihnen um „aflex“ und „ayacc“, deren Eingabeformat an das in Compilers: Principles, Techniques and Tools [1] beschriebene Format angelehnt ist.

Aus der Spezifikation wurde die im Anhang A abgedruckte Grammatik entwickelt und mit besagten Werkzeugen ein fertiger Scanner sowie ein Parser generiert.

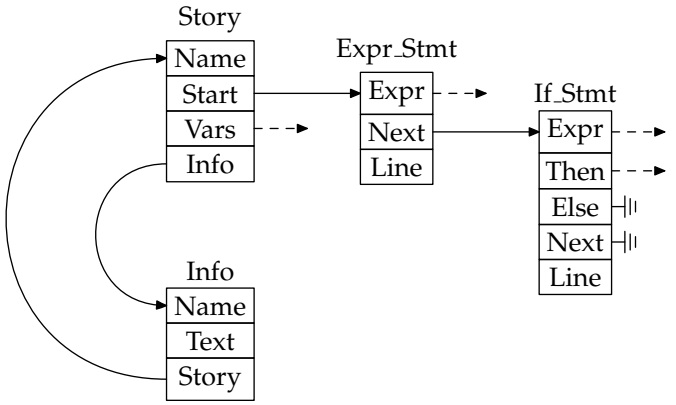
Die generierten Pakete existieren jeweils in einer Variante zum Parsen eines einzelnen Ausdrucks und eines kompletten Skripts.

6.3. Laufzeitumgebung

6.3.1. Aufbau der Skript-Datenstrukturen

Die Datenstrukturen jedes Skripts sind an einem Story_Record Wurzelelement aufgehängt (siehe Abbildung 6.1).

Abbildung 6.1 Skript-Datenstruktur



Jedes vom Parser erkannte Statement wird zu einem Statement_Record, die als verkettete Liste gespeichert werden.

Von diesem Record gibt es jeweils Ausprägungen für Zuweisungen („`Assign_Stmt`“), Abbruch („`Break_Stmt`“), Aufrufe („`Call_Stmt`“), Deklarationen („`Declare_Stmt`“), Graph Iteratoren („`Match_Stmt`“), If („`If_Stmt`“) sowie Loop- („`Loop_Stmt`“) und While-Schleifen („`While_Stmt`“).

Terme, die in Zuweisungen, Funktionsargumenten und bedingten Konstrukten be-

nutzt werden können, werden als eigenständige Term_Records repräsentiert.

6.3.2. Interpreter-Datenstrukturen

Die zentrale Speicherstruktur des Interpreters ist das Dictionary, in dem unter verschiedenen Namen Werte gespeichert und gelesen werden können. Es ist gleichzeitig ein Stapel eingebaut, so dass beim Eintritt in einen neuen Definitionsbereich neue Variablen angelegt werden können, ohne die alten Variablen zu überschreiben. Beim Verlassen des Definitionsbereichs können dann die alten Variablen einfach wiederhergestellt werden.

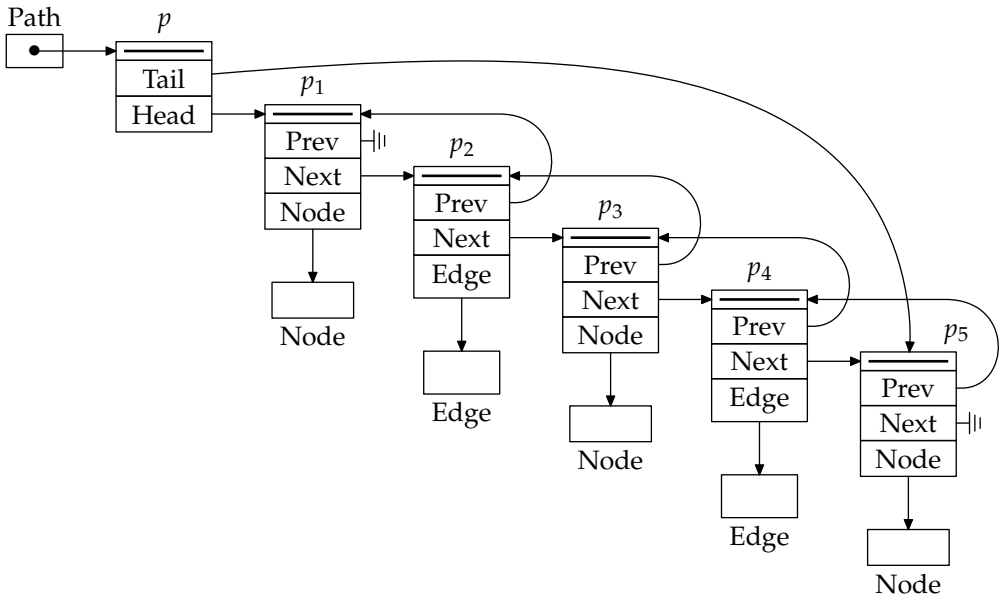
Variablen, und damit die vom Skript gehaltenen Daten, werden in Variable_Records gespeichert und in das Dictionary eingehängt.

6.3.3. Datenstrukturen der Werte

Im Skript benutzbare Typen werden in verschiedenen tagged records gespeichert, die eine gleichförmige Schnittstelle zur Übergabe bereitstellen. Sie erben vom abstrakten record `Value` die Typabfrage `Get_Type_Id` sowie die Ausgabe als String `To_String`.

Für RFGScript wurden neue Implementierungen von Pfaden und Relationen realisiert. Um diese auch getrennt einsetzen zu können, wurden sie nicht direkt in Form eines Erben von `Value` implementiert, sondern als abstrakter Datentyp (ADT).

Abbildung 6.2 Datenstruktur eines Pfads



Pfade

Die Datenstruktur eines Pfads ist in Abbildung 6.2 dargestellt.

Der Path-ADT speichert aus dem RFG stammende Kanten- und Knotenreferenzen und ist daher nicht vom RFG getrennt einsetzbar.

Die einzelnen Elemente eines Pfads sind in Form einer doppelt verketteten Liste realisiert, so dass auf einfache Weise auf beide Enden des Pfads zugegriffen werden kann.

Folgende Operationen sind definiert:

`function Create return Path`

erzeugt einen leeren Pfad

`procedure Destroy (The_Path : in out Path)`

zerstört den Pfad und gibt den Speicher frei. Im Pfad vorhandene Kanten und Knoten werden nicht verändert.

`function Head (The_Path : in Path) return RFGs.Node`

entfernt den Kopf des Pfads und liefert ihn zurück. Wenn der Kopf kein Knoten ist, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad leer ist ist, wird eine `No_More_Error`-Exception geworfen.

`function Head (The_Path : in Path) return RFGs.Edge`

entfernt den Kopf des Pfads und liefert ihn zurück. Wenn der Kopf keine Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad leer ist ist, wird eine `No_More_Error`-Exception geworfen.

`function Tail (The_Path : in Path) return RFGs.Node`

entfernt das letzte Element des Pfads und liefert es zurück. Wenn es kein Knoten ist, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad leer ist ist, wird eine `No_More_Error`-Exception geworfen.

`function Tail (The_Path : in Path) return RFGs.Edge`

entfernt das letzte Element des Pfads und liefert es zurück. Wenn es keine Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad leer ist ist, wird eine `No_More_Error`-Exception geworfen.

`function At_Pos (The_Path : in Path; Pos : in Natural) return RFGs.Node`

liefert das an der angegebenen Position im Pfad befindliche Element zurück. Wenn es kein Knoten ist, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad kürzer als die angegebene Position ist, wird eine `No_More_Error`-Exception geworfen.

`function At_Pos (The_Path : in Path; Pos : in Natural) return RFGs.Edge`

liefert das an der angegebenen Position im Pfad befindliche Element zurück. Wenn es keine Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad kürzer als die angegebene Position ist, wird eine `No_More_Error`-Exception geworfen.

`function Head_Is_Node (The_Path : in Path) return Boolean`

liefert `True` zurück, wenn das erste Element des Pfads ein Knoten ist. Ansonsten wird `False` zurückgeliefert. Wenn der Pfad leer ist ist, wird eine `No_More_Error`-Exception geworfen.

`function Tail_Is_Node (The_Path : in Path) return Boolean`

liefert `True` zurück, wenn das letzte Element des Pfads ein Knoten ist. Ansonsten wird `False` zurückgeliefert. Wenn der Pfad leer ist ist, wird eine `No_More_Error`-Exception geworfen.

`function Node_At_Pos (The_Path : in Path; Pos : in Natural)`
`return Boolean`

liefert `True` zurück, wenn das Element an der angegebenen Position im Pfad ein Knoten ist. Ansonsten wird `False` zurückgeliefert. Wenn der Pfad kürzer als die angegebene Position ist, wird eine `No_More_Error`-Exception geworfen.

`function Size (The_Path : in Path) return Natural`

liefert die Länge des Pfads zurück.

`procedure Prepend (The_Path : in Path; The_Node : in Node)`

hängt den angegebenen Knoten an den Anfang des Pfad an. Wenn der Pfad schon zuvor mit einem Knoten anfing, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad zuvor mit einer Kante anfing, und der Knoten nicht der Startknoten dieser Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen.

`procedure Prepend (The_Path : in Path; The_Edge : in Edge)`

hängt die angegebene Kante an den Anfang des Pfad an. Wenn der Pfad schon zuvor mit einer Kante anfing, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad zuvor mit einem Knoten anfing, und der Knoten nicht der Endknoten dieser Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen.

`procedure Append (The_Path : in Path; The_Node : in Node)`

hängt den angegebenen Knoten an das Ende des Pfad an. Wenn der Pfad schon zuvor mit einem Knoten endete, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad zuvor mit einer Kante endete, und der Knoten nicht der Endknoten dieser Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen.

`procedure Append (The_Path : in Path; The_Edge : in Edge)`

hängt die angegebene Kante an das Ende des Pfad an. Wenn der Pfad schon zuvor mit einer Kante endete, wird eine `Out_Of_Order_Error`-Exception geworfen. Wenn der Pfad zuvor mit einem Knoten endete, und der Knoten nicht der Startknoten dieser Kante ist, wird eine `Out_Of_Order_Error`-Exception geworfen.

Relationen

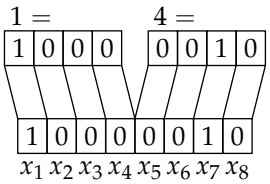
Relationen haben normalerweise keine Entsprechung im RFG und müssen daher separat gespeichert werden. Dies führt bei Benutztung der Ausdrucksmittel der relationalen Algebra zu großen Speicheranforderungen. Die Darstellung von Relationen als Kantenlisten oder Knotenmatrizen ist für größere Graphen kaum mehr möglich.

Beyer et al. [2] beschreiben die Darstellung von Relationen als Binäre Entscheidungsdiagramme (BDD), die eine kleinere Darstellung ermöglichen. Die Anwendung von BBDs in CrocoPat wird im Abschnitt 3.4 beschrieben.

Um mit einem BDD eine Relation darstellen zu können, müssen zuerst die Knoten des RFGs in eine geeignete binäre Darstellung konvertiert werden. Zu diesem Zweck wird die Indexnummer des Knotens verwendet, die momentan in einem RFG eindeutig ist. Alternativ könnte auch die Adresse des im RFG genutzten Knoten-Records zur generierung einer eindeutigen Nummer genutzt werden. Diese Indexnummer wird in eine Binärdarstellung konvertiert und mit '0' auf die maximale Größe aufgefüllt. Entgegen der üblichen Schreibweise werden hier die Bits aufsteigend von links notiert.

Abbildung 6.3 zeigt dies für die Knotenindizes „1“ und „4“). Die maximale Größe orientiert sich an der Definition der Indexnummer, momentan ist sie als 32-Bit Integer definiert, somit ist die maximale Größe 32. Da die Relation Knotenpaare speichert, werden die Binärdarstellungen zweier Knoten aneinandergehängt und bilden eine 64-Bit-Zahl. Eine Relation lässt sich auf diese Weise als eine Menge von 64-Bit-Zahlen darstellen.

Abbildung 6.3 Konvertierung der Knotennummern „1“ und „4“



Die nun folgenden Betrachtungen finden auf den BDD Datenstrukturen statt und haben keine Verbindung mit dem RFG.

Ein BDD ist ein gerichteter Graph ohne Zyklen mit einem Wurzelknoten. Er besteht aus Entscheidungsknoten, die jeweils einen *Low*- und *High*-Nachfolger haben und mit

einer Booleschen Variable markiert sind. Nachfolger sind entweder weitere Entscheidungsknoten, oder die Terminalknoten $\boxed{0}$ und $\boxed{1}$.

In Abbildung 6.4 ist der Aufbau eines BDD dargestellt. Kanten zu *Low*-Nachfolgern sind gestrichelt gezeichnet, Kanten zu *High*-Nachfolgern sind durchgezogen gezeichnet. Kanten, die zum $\boxed{0}$-Terminalknoten führen, sind zur Vereinfachung der Darstellung nicht eingezeichnet. Die Markierung der Booleschen Variablen erfolgt mit $x_1 \ldots x_8$.

Abbildung 6.4 BDD (1)

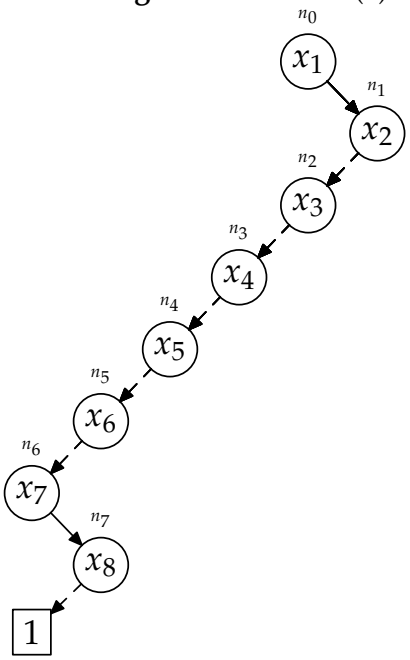
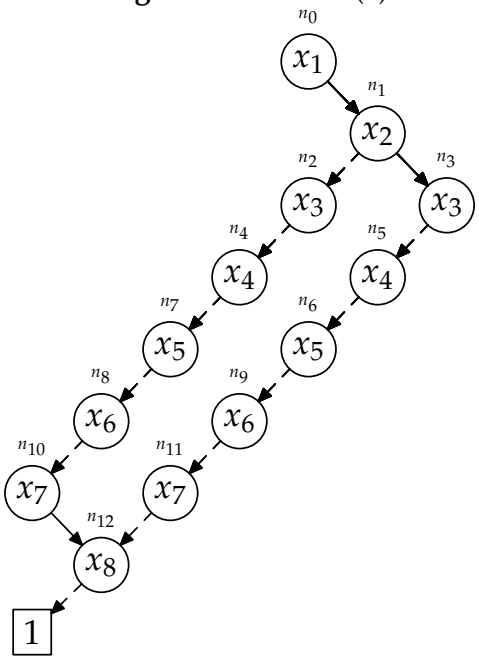


Abbildung 6.5 BDD (2)



Das Diagramm in Abbildung 6.4 repräsentiert die Relation $\{(1,4)\}$, die sich binär als $\{10000010\}$ darstellen lässt. Durch hinzufügen des Tupels $(3,0)$ entsteht das BDD in Abbildung 6.5. Dieses repräsentiert eine Relation, die sich binär als $\{10000010, 11000000\}$ darstellen lässt.

Um festzustellen, ob sich ein Tupel in der Relation befindet, wird das Tupel wie oben in eine Binärzahl konvertiert. Nun wird von links her für jedes Bit dieser Zahl einer Kante im Diagramm gefolgt. Ist das Bit 0, geht es in Richtung des *Low*-Nachfolgers, ansonsten in Richtung des *High*-Nachfolgers. Ist der Nachfolger ein $\boxed{1}$-Terminalknoten, ist das Tupel in der Relation enthalten. Ist der Nachfolger ein $\boxed{0}$-Terminalknoten, ist das Tupel nicht in der Relation enthalten. Ansonsten wird die Suche mit dem Nachfolgeknoten und dem nächsten Bit der Binärzahl fortgesetzt.

Dies soll mit folgendem Beispiel verdeutlicht werden: es soll festgestellt werden, ob das Tupel $(1,5)$ in der durch das BDD in Abbildung 6.4 dargestellten Relation enthalten ist. Die Binärdarstellung des Tupels ist 10001010. Das erste Bit der Binärzahl ist

1, daher wird im Diagramm vom Startknoten n_0 über die *High*-Kante zum Knoten n_1 gefolgt. Das zweite Bit der Binärzahl ist 0, es wird vom Knoten n_1 über die *Low*-Kante zum Knoten n_2 gefolgt. Das dritte Bit der Binärzahl ist 0, es wird vom Knoten n_2 über die *Low*-Kante zum Knoten n_3 gefolgt. Das vierte Bit der Binärzahl ist 0, es wird vom Knoten n_3 über die *Low*-Kante zum Knoten n_4 gefolgt. Das fünfte Bit der Binärzahl ist 1, es wird vom Knoten n_4 über die (nicht eingezeichnete) *High*-Kante zum Terminalknoten $\boxed{0}$ gefolgt. Damit ist klar, dass das Tupel $(1, 5)$ nicht in der Relation enthalten ist.

Der Vorteil der Darstellung von Relationen als BDDs soll durch die Abbildungen 6.6 und 6.7 verdeutlicht werden. Das Diagramm in Abbildung 6.6 repräsentiert die Relation $\{(0,0),(0,1),\ldots,(0,7)\}$. Obwohl in dieser Relation acht Tupel repräsentiert werden, ist das Diagramm nicht grösser als das der Relation $\{(1,4)\}$. Das Diagramm in Abbildung 6.7 repräsentiert die Relation $\{(0,0),(0,1),\ldots,(0,15),(1,0),\ldots,(6,0)\}$. Auch hier sieht man, dass die Anzahl der Entscheidungsknoten langsamer als die Anzahl der Tupel der Relation wächst.

Da beim Einsatz der relationalen Algebra oft grosse Relationen entstehen, ist diese Eigenschaft vorteilhaft.

Abbildung 6.6 BDD (3)

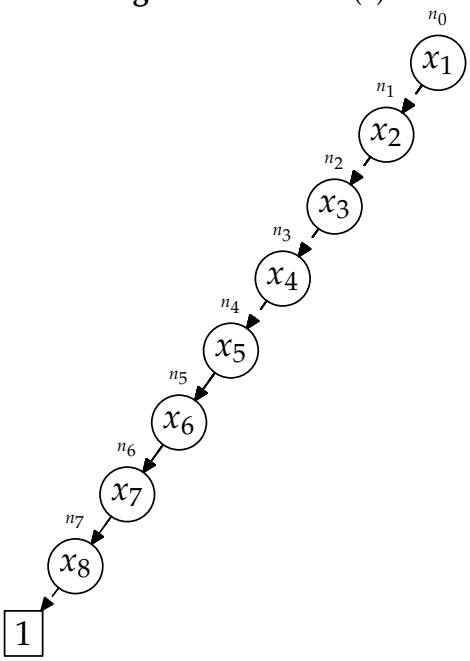
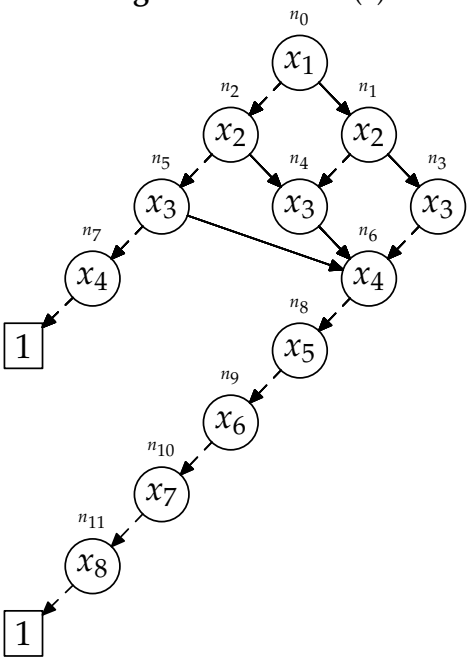


Abbildung 6.7 BDD (4)



Folgende Operationen sind auf einer Relation definiert:

`function Empty_Relation (The_RFG : RFGs.RFG) return Relation`

erzeugt eine leere Relation.

`procedure Destroy (The_Relation : in out Relation)`

gibt den von der Relation belegten Speicher frei. Die in der Relation referenzierten Knoten des RFGs werden dabei nicht verändert.

`function Clone (The_Relation : Relation) return Relation`

gibt eine tiefe Kopie der Relation zurück.

`function Union (A, B: Relation) return Relation`

gibt die Vereinigung $A \cup B$ der übergebenen Relationen zurück.

`function Difference (A, B: Relation) return Relation`

gibt die Differenz $A - B$ der übergebenen Relationen zurück.

`function Intersection (A, B: Relation) return Relation`

gibt die Schnittmenge $A \cap B$ der übergebenen Relationen zurück.

`function Composition (A, B: Relation) return Relation`

gibt die relationale Komposition der übergebenen Relationen zurück.

`function Inv (A : Relation) return Relation`

gibt die relationale Inversion der übergebenen Relation zurück.

`function Proj (N : Node_Set; R : Relation) return Node_Set`

projiziert die übergebene Knotenmenge durch die Relation.

`function Proj (R : Relation; N : Node_Set) return Node_Set`

projiziert die übergebene Knotenmenge rückwärts durch die Relation.

`function Closure (R : Relation) return Relation`

gibt den transitiven Abschluss der übergebenen Relation zurück.

`function Dom (R : Relation) return Node_Set`

gibt die Startknoten der übergebenen Relation zurück.

`function Ran (R : Relation) return Node_Set`

gibt die Endknoten der übergebenen Relation zurück.

`function Entities (R : Relation) return Node_Set`

gibt die Start- und Endknoten der übergebenen Relation zurück.

`function Identity (N : Node_Set; RFG : in RFGs.RFG) return Relation`

gibt die Identitätsrelation für die übergebene Knotenmenge zurück.

`function Cross (A, B : Node_Set; RFG : in RFGs.RFG) return Relation`

gibt das Kreuzprodukt für die übergebene Knotenmenge zurück.

`function To_Relation (E : Edge_Set; RFG : in RFGs.RFG) return Relation`

erzeugt aus der übergebenen Kantenmenge eine Relation.

`function To_Edge_Set (R : Relation; Desc : String) return Edge_Set`

erzeugt für jedes Knotenpaar der übergebenen Relation eine Kante im RFG und gibt diese als Kantenmenge zurück. Die Kanten werden in der Base_View mit dem übergebenen Descriptor angelegt, sofern sie nicht schon existieren.

`procedure Append (R : in out Relation; A, B : Node)`

fügt das übergebene Knotenpaar in die Relation ein.

`procedure Remove (R : in out Relation; A, B : Node)`

entfernt das übergebene Knotenpaar aus der Relation.

`procedure Head (R : Relation; A, B : out Node)`

gibt ein in der Relation existierendes Knotenpaar zurück. Die Paare in der Relation haben keine definierte Ordnung, daher ist nicht definiert, welches Paar zurückgegeben wird.

`function Size (R : Relation) return Natural`

gibt die Anzahl der in der Relation existierenden Knotenpaare zurück.

`function Is_Empty (R : Relation) return Boolean`

gibt `True` zurück, wenn mindestens ein Knotenpaar in der Relation existiert, ansonsten `False`.

`function Equals (A, B : Relation) return Boolean`

gibt `True` zurück, wenn die übergebenen Relationen die gleichen Knotenpaare enthalten, ansonsten `False`.

Attribute

Knoten und Kanten im RFG können mit Name-Wert-Paaren annotiert werden. Diese werden zur Laufzeit registriert.

In Abbildung 6.8 sind die zur Zeit verfügbaren Attributtypen sowie die entsprechenden RFGScript Datentypen gegenübergestellt.

Abbildung 6.8 RFG Attributtypen vs. RFGScript Typen

Boolean	Boolean
Toggle	Boolean
Integer	Integer
Float	Integer
String	String

Diese Konvertierungen sind in den `Get_Attr-`, `Set_Attr-` sowie `Select`-Funktionen implementiert.

6.4. Funktionsweise des Interpreters

Die Ausführung eines Skripts geschieht in mehreren Phasen:

1. Laden des Skript-Quelltexts in den Speicher,
2. Parsen des Quelltexts, es wird eine zur Ausführung geeignete Datenstruktur im Speicher aufgebaut,
3. Verifikation des Skripts, insbesondere auf Übereinstimmung der Typen,
4. der Benutzer bekommt den Dialog zur Eingabe der Startparameter angeboten
5. Initialisierung der Interpreter-Datenstrukturen und Übergabe bzw. Berechnung der Startparameter,
6. das Skript wird ausgeführt,
7. die Rückgabeparameter werden dem Benutzer angezeigt.

Die Ausführung eines Skripts wird im folgenden genauer beschrieben.

6.5. Initialisierung und Ausführung

Der Programmzähler des Interpreters wird auf das erste Statement des Skripts gesetzt. Dieses ist im Story_Record vermerkt.

Die Eingabeparameter des Skripts können Terme sein, und werden dementsprechend interpretiert bzw. es werden die entsprechenden Werte oder Mengen aus dem RFG geladen und in die für den Interpreter passende Darstellung konvertiert. Diese werden dann wie normale Variablen ins Dictionary eingetragen.

6.5.1. Kontrollfluß

Das nächste auszuführene Statement wird im Programmzähler des Interpreters gespeichert, der in Form eines Pointers realisiert ist. Nach der Ausführung eines Statements muss entschieden werden, welches Statement als nächstes ausgeführt wird. Auf dieses wird dann der Programmzähler gerichtet.

Normalerweise wird das auf das momentane Statement nachfolgende Statement ausgeführt. Statement-Folgen sind als verkettete Liste realisiert, das nächste Statement ist im *Next*-Attribut vermerkt.

Für die Realisierung der Kontrollflußkonstrukte ist das nicht ausreichend. Es können folgende Fälle auftreten:

Aufspaltung des Kontrollflusses Diese treten bei fast allen Kontrollflußkonstrukten auf. Dabei wird zwischen dem innerhalb des Blocks liegenden Fluß (im Attribut *Children* vermerkt) und dem nachfolgenden Fluß (wie o.g. *Next*-Attribut) unterschieden.

Bevor die Verarbeitung des nachfolgenden Statements fortgeführt werden kann, muss erst der Block ausgeführt werden.

Diese Blöcke können in beliebiger Schachtelungstiefe auftreten und jede Blockausführung muss gespeichert werden, um die Ausführung des dem Block nachfolgenden Statements zu ermöglichen.

Dazu wird ein Stapel aller ausgeführten Blöcke geführt, der vor der Ausführung eines relevanten Statements aufgebaut, und am Ende des jeweiligen Blocks bzw. beim Abbruch durch ein Exit-Statement wieder abgebaut wird. Dabei muss evtl. auch eine vorher erzeugte Dictionary-Ebene aufgeräumt werden.

Loop-Schleife Diese wird ausgeführt, bis innerhalb des Blocks ein `break`-Statement durchlaufen wird. Wird es erkannt, wird die Ausführung des Blocks abgebrochen und sofort das dem Block folgende Statement ausgeführt.

Wird kein Break-Statement durchlaufen, wird am Ende des Blocks der Programmzähler wieder auf das erste Statement des Blocks gerichtet.

While-Schleife Am Anfang dieser Schleife wird dessen Argument evaluiert. Ist das Ergebnis „`True`“, wird der Block durchlaufen.

Am Ende des Blocks wird der Programmzähler wieder auf das erste Statement des Blocks gerichtet.

If-Statement Am Anfang dieses Statements wird dessen Argument evaluiert. Ist das Ergebnis „`True`“, wird der Block durchlaufen. Am Ende des Blocks wird der Programmzähler auf das erste Statement nach dem Block gerichtet.

Declare-Statement Am Anfang dieses Statements werden die im Statement deklarierten Variablen angelegt und der Block durchlaufen. Am Ende des Blocks wird der Programmzähler auf das erste Statement nach dem Block gerichtet.

Match-Statement Am Anfang dieses Statements werden die im Statement deklarierten Variablen angelegt und die Belegungsmengen erzeugt. Für jede passende Belegung wird der Block durchlaufen.

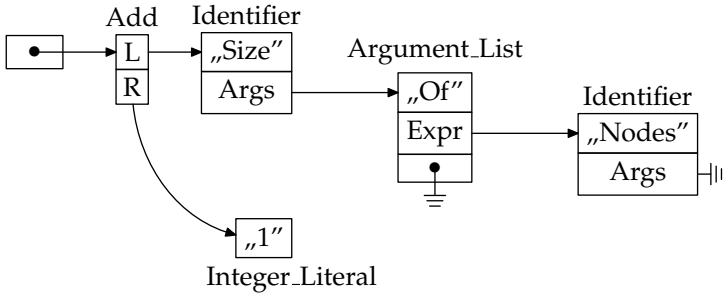
Am Ende des Blocks wird der Programmzähler wieder auf das erste Statement des Blocks gerichtet.

6.5.2. Berechnung von Ausdrücken

Beim Parsen eines Skripts oder eines einzelnen Ausdrucks wird ein Baum von Term-Objekten aufgebaut, siehe z.B. Abbildung 6.9. Die Argumente jedes Terms werden als Pointer auf den Term des jeweiligen Arguments gehalten.

Beim Aufruf einer Funktion über einen Identifier ist zum Zeitpunkt des Parsens noch keine Auflösung der Argumente möglich. Daher wird in diesem Fall eine Liste von Arguments angelegt, die die Argument-Terme mit dem Argumentnamen verknüpfen und zur Laufzeit ausgelesen werden können. Der Identifier-Term hält nur einen Pointer auf die Argumentliste.

Abbildung 6.9 Der Term „`Size (Of => Nodes) + 1`“



Jeder Term hat eine Eval-Methode, die ein Value mit dem Ergebnis des Terms zurückliefert. Der Baum wird vom Wurzelknoten her rekursiv abgearbeitet, so dass zuerst die

Eval-Methode in den Blättern aufgerufen wird. Somit können die der Wurzel näheren Objekte auf deren Ergebnisse zurückgreifen.

6.5.3. Speicherverwaltung

Während der Laufzeit des Interpreters wird zur Speicherung von Werten dynamisch Speicher angefordert. Dieser Speicher muss spätestens mit dem Ende der Skriptausführung wieder freigegeben werden.

Werte werden einerseits in Variablen gespeichert, wo sie dauerhaft zur Verfügung stehen. Allerdings werden auch während der Auswertung von Ausdrücken neue Werte erzeugt und evtl. Werte von Zwischenergebnissen überflüssig. Um den für diese Werte angeforderten Speicher wieder freigeben zu können, werden folgende Vorkehrungen getroffen:

- Jeder Wert besitzt einen Referenzzähler, der von jedem Objekt, der den Wert referenziert, erhöht und nach Verwendung wieder erniedrigt wird.
- Jeder Wert wird in eine Liste aller vorhandenen Werte eingefügt. Da pro Variable und Ausdruck-Term maximal ein neuer Wert erzeugt wird, wächst diese Liste nicht stark an.
- Nach der Ausführung jeder Anweisung wird die Liste nach nicht mehr verwendeten Werten durchsucht. Diese werden aus der Liste entfernt und der Speicher freigegeben.

Nachdem das Skript ausgeführt wurde, kann der Speicher aller in der Liste noch vorhandenen Werte freigegeben werden. Dabei ist zu beachten, dass mit `export` markierte Variablen Werte enthalten, die auch nach dem Ende des Skripts noch benötigt werden. Die Freigabe dieser Werte liegt in der Verantwortung der aufrufenden Anwendung, z.B. Gravis.

6.5.4. Typauflösung

Verschiedene Funktionen sind unter dem gleichen Namen für Eingabeparameter unterschiedlichen Typs definiert.

Die Validierung des Skripts vor der Ausführung stellt in vielen Fällen sicher, dass Funktionen nur mit Argumenten aufgerufen werden, für die sie definiert sind.

Dies ist jedoch nicht in allen Fällen möglich, z.B. bei Funktionen, die mit Pfaden arbeiten. In diesen Fällen wird in den Implementierungen der Funktionen zur Laufzeit anhand der übergebenen Werte die passende Implementierung aufgerufen.

6.5.5. Fehlerbehandlung

Tritt bei der Evaluierung eines Terms ein Fehler auf, wird der Fehlertext im Context hinterlegt und eine Exception geworfen. Diese führt normalerweise dazu, dass die Skriptausführung abgebrochen wird.

6.6. Umgebung

Der Interpreter wurde in Ada95 implementiert. Zusätzlich wurde die GtkAda-GUI-Bibliothek sowie die Pakete des Bauhaus-Systems verwendet.

Der Interpreter wird hauptsächlich in Verbindung mit Gravis benutzt. Zu Testzwecken wurde jedoch auch eine eigenständige Variante realisiert, die ihre Ergebnisse als Text ausgibt.

Die Datenstrukturen des Interpreters sind auf folgendes Mengengerüst ausgelegt:

	typisch Anzahl	max.
Knoten und Kanten im RFG	100000	∞
Views im RFG	20	64
Skripts	100	∞
Knote, Kanten und Pfade in Match-Statements	30	∞

6.7. Test der Implementierung

6.7.1. Methode

Die ADTs Path und Relation wurden durch Unit-Tests getestet.

Der Interpreter wurde mit Hilfe verschiedener Skripts getestet, die jeweils eine Funktionalität benutzten.

Dazu wurde zu jedem Skript das Sollergebnis in einer Datei abgelegt. Mit der Kommandozeilenversion des Interpreters wurde dann eine Datei mit dem Istinhalt erzeugt.

Dadurch ist der Test automatisiert wiederholbar.

6.7.2. Testfälle

Für folgende Funktionalität existieren automatisierte Testfälle:

- Ausgabe von Boolean, Integer und String mit `Print`

- Setzen und Lesen von Boolean-, Integer- und Stringvariablen
- Sichtbarkeitsbereiche von Variablen
- Erzeugen und Lesen von Knoten- und Kantenmengen

6.7.3. Ergebnisse

Die Auswertung der Tests ergab keine Abweichung der Istergebnisse von den Sollergebnissen.

7. Effizienzvergleich

In diesem Kapitel wird die Vergleichsmethode, die untersuchten Daten sowie die Ergebnisse des Effizienzvergleichs dargestellt.

Untersucht werden dabei neben RFGScript die in Kapitel 3 vorgestellten Systeme Grok[1] sowie CrocoPat[2]. GReQL war leider für den Vergleich nicht verfügbar.

7.1. Methode

Es werden Algorithmen ausgesucht, die durch jede zu vergleichende Methode realisiert werden kann. Die Algorithmen sollen gebräuchliche Anforderungen abdecken.

Ein fairer Vergleich wird dadurch erschwert, dass die zu vergleichenden Systeme zwar alle Quellcode analysieren; jedoch wird keineswegs der gleiche Ursprungsgraph erstellt, sondern jeweils an das System angepasste Graphen, die völlig unterschiedlichen Aufbaus sein können. Daher wird zum Vergleich auf einen synthetisch erzeugten Graph, der für alle Systeme gleich aussieht, zurückgegriffen.

Als Ausgangsbasis dienen für das Bauhaus-Projekt erzeugte RFGs der Quelltexte von ACM (120 Knoten, 205 Kanten), GNU Chess (515 Knoten, 2571 Kanten) und Mosaic (1219 Knoten, 2152 Kanten). Diese RFGs werden dann nach RSF konvertiert, das von Grok und CrocoPat eingelesen werden kann.

Die Laufzeit wird jeweils 10 mal gemessen und gemittelt. Um die reine Bearbeitungszeit zu erhalten, wird die Anfrage zu Anfang mit einem leeren Graphen durchgeführt und die somit erhaltenen Startaufwände abgezogen. Als Hardware dient ein AMD Athlon mit 1,2 GHz.

[1] SWAGKit Version 3.01 von `http://www.swag.uwaterloo.ca/swagkit/`

[2] Version 2.1.1 von `http://www-sst.informatik.tu-cottbus.de/~db/CrocoPat/`

7.2. Untersuchte Probleme

Folgende Probleme wurden untersucht:

- es wird die relationale Komposition zweier Relationen berechnet
- es wird jeweils der transitive Abschluss einer Relation berechnet
- aus dem Graph werden Uses-Beziehungen extrahiert

7.3. Ergebnisse

Alle Laufzeiten sind in Sekunden angegeben.

	Grok	CrocoPat	RFGScript
relationale Komposition			
ACM	0,009	0,094	0,493
GNU Chess 5.0	0,046	0,512	27,263
Mosaic	0,111	1,126	42,970
Transitiver Abschluss			
ACM	0,006	0,094	5,167
GNU Chess 5.0	0,054	0,601	∞[3]
Mosaic	0,120	1,252	285,812
Uses-Beziehungen extrahieren			
ACM	0,005	0,101	0,822
GNU Chess 5.0	0,041	0,544	201,794
Mosaic	0,106	1,238	29,841

7.4. Auswertung

Auf den ersten Blick fällt die deutlich längere Laufzeit der Anfragen durch RFGScript auf. Da die relationalen Operationen auf den gleichen Strukturen wie bei CrocoPat aufbauen, deutet dies auf Optimierungsmöglichkeiten in RFGScript hin. Der im Vergleich zu den anderen Systemen geringere Anstieg der Laufzeit bei der Berechnung der relationalen Komposition lässt darauf schließen, dass die Laufzeit der RFGScript-Anfrage mit größer werdenden Graphen nicht in gleichem Maße zunimmt, wie bei den anderen Systemen.

[3] Programm wurde nach 30 Minuten abgebrochen

A. Grammatik

story	::=	“**story**” Identifier “**is**” (“**begin**” stmt_list “**end story**” “**;**” | varlist “**begin**” stmt_list “**end story**” “**;**”)
stmt_list	::=	stmt { stmt }
stmt	::=	(Identifier “**:=**” exp | “**if**” exp “**then**” stmt_list “**else**” stmt_list “**end if**” | “**if**” exp “**then**” stmt_list “**end if**” | “**while**” exp “**loop**” stmt_list “**end loop**” | “**loop**” stmt_list “**end loop**” | “**break**” | “**match**” matchvarlist “**where**” wherelist “**begin**” stmt_list “**end match**” | “**script**” stmt_list “**end script**” | “**declare**” varlist “**begin**” stmt_list “**end declare**” | exp) “**;**”
exp	::=	exp “**and**” exp
	|	exp “**or**” exp
	|	exp “**xor**” exp
	|	exp “=” exp
	|	exp “!=” exp
	|	exp “<” exp
	|	exp “<=” exp
	|	exp “>” exp
	|	exp “>=” exp
	|	exp “+” exp
	|	exp “−” exp
	|	“−” exp
	|	exp “∗” exp
	|	exp “/” exp
	|	“**not**” exp
	|	Identifier
	|	Identifier “**(**” arg_list “**)**”
	|	Numeric_Literal
	|	Boolean_Literal
	|	String_Literal
	|	“**(**” exp “**)**”
varlist	::=	{ varspec } varspec
varspec	::=	[“**import**” | “**export**”] Identifier “**:**” Type_Identifier “**;**”
matchvarlist	::=	{ matchvarspec } matchvarspec

matchvarspec	::=	Identifier “**:**” Type_Identifier “$<-$” exp “**;**”
wherelist	::=	{ wherespec } wherespec
wherespec	::=	(Identifier “$<-$” | Identifier “$--$” Identifier “$->$” Identifier | Identifier “$\sim\sim$” Identifier “$\sim>$” Identifier | Identifier “$<-$” Identifier “$--$” Identifier | Identifier “$<\sim$” Identifier “$\sim\sim$” Identifier | exp) “**;**”
arg_list	::=	{ arg “**,**” } arg
arg	::=	Identifier “$=>$” exp

B. Pakete

- Scanner und Parser
  - **Scanner, Parser**. Scanner und Parser für Skripte.
  - **Expression_S, Expression_P**. Scanner und Parser für Ausdrücke.
- Laufzeitumgebung
  - **RFGScript.Types**. Definition der in RFGScript vorhandenen Typen.
  - **RFGScript.Values**. ADTs für Werte.
  - **RFGScript.Terms**. In Ausdrücken verwendbare Terme.
  - **RFGScript.Functions**. Die in RFGScript fest eingebauten Funktionen.
  - **RFGScript.Statements**. Anweisungen.
  - **RFGScript.Stories**. Datenstrukturen für Skripte.
  - **RFGScript.Interpreters**. Der Interpreter sowie das Dictionary.
  - **RFGScript.Parsers**. Eine Kapsel für die generierten Parser.
  - **RFGScript.Dumpers**. Funktionen zum Generieren von Quelltext aus Skript-Datenstrukturen.
  - **RFGScript.Variables**. Variablen.
  - **RFGScript.Globals**. Globale Objekte.
- GtkAda-Erweiterungen
  - **Gtk.List_Store_Extra**. Erweiterung des normalen List_Store zur Speicherung von Benutzerdaten.
  - **Gtk.Text_Buffer_Marshallers**. Marshaller für Callbacks mit Benutzerdaten für Gtk_Text_Buffers.
  - **Gtk.Text_View_Marshallers**. Marshaller für Callbacks mit Benutzerdaten für Gtk_Text_Views.
- Gravis-Integration
  - **Gravis_Gui.RFGScript_Widgets**. Funktionen zur Anzeige von Knoten-, Kanten- und Pfadlisten und die Reaktion auf deren Signale.

- **Gravis_Gui.Query_Expression**. Dialogfenster für die Ausführung von Ausdrücken.
- **Gravis_Gui.Query_Script**. Dialogfenster für die Ausführung von Skripten.
- **RFGScript.Gui.Story_Editors**. Texteditor für Skripte.
- **RFGScript.Gui.Match_Widgets**. Darstellung von Match-Elementen.

Literaturverzeichnis

[1] Aho, A. V. ; Sethi, R. ; Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. 1. Reading, Mass. : Addison-Wesley, 1986

[2] Beyer, Dirk ; Noack, Andreas ; Lewerentz, Claus: Simple and Efficient Relational Querying of Software Structures. In: *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE 2003, Victoria)*, IEEE Computer Society Press, Los Alamitos (CA), 2003, S. 216–225

[3] Eisenbarth, Thomas. *GropiusSE. Eine Resource Flow Graph Bibliothek in Ada95 für das Speichern und Aufbereiten von Reengineeringinformationen*. 1998

[4] Festi, Florian. *Einbindung einer Skriptsprache für Gravis*. 2002

[5] Holt, Richard C.: Structural Manipulations of Software Architecture using Tarski Relational Algebra. In: *Working Conference on Reverse Engineering*, 1998, S. 210–219

[6] Rohrbach, Jürgen. *Erweiterung und Generierung einer Zwischensprache für C–Programme*. 1998

[7] Schwienbacher, Martin [u. a.]: *GIANT Spezifikation, Version 1.1*, 2003

[8] Tarski, Alfred: On the Calculus of Relations. In: *The Journal of Symbolic Logic* 6 (1941), Nr. 3, S. 73–89

[9] Wong, Kenny: *Rigi User's Manual*, 1996

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

_____

(Michael Stürmer)