Visualisierung Termgraph-orientierter Berechnungen

Sascha Ecks

Bachelorarbeit
03/2022

Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
Prof. Michael Hanus und Kai Prott, M.Sc.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Für Lehr- und Lernzwecke sowie zum Finden von Programmierfehlern ist das schrittweise Durchlaufen eines Programmablaufs bei imperativen Sprachen ein hilfreiches Mittel, welches jedoch in deklarativen Sprachen weniger leicht anwendbar ist.

Um Berechnungen in der logisch-funktionalen Programmiersprache Curry, in der Berechnungszustände durch Termgraphen dargestellt werden, leichter nachvollziehen zu können, ist Gegenstand dieser Arbeit die Visualisierung von Termgraphen aus den Berechnungszuständen eines Curry-Laufzeitsystems. Zusätzlich zur Implementierung des Termgraph-Visualisierers wurde für eine einfache Bedienung des selben eine Webanwendung entwickelt, in welcher Programm und auszuwertender Ausdruck eingegeben werden und danach visualisierte Termgraphen in einer Diashow betrachtet werden können.

Inhaltsverzeichnis

Kapitel 1

Motivation und Ausgangslage

In vielen Situation ist es praktisch, den Ablauf eines Programms Schritt für Schritt nachvollziehen zu können: Ob in der Lehre, zur Vermittlung von Programmierkonzepten oder bei der Softwareentwicklung, um nach Programmierfehlern zu suchen oder die Funktionsweise existierender Programme nachzuvollziehen. Anders als in imperativen Programmiersprachen werden in deklarativen Programmiersprachen nun allerdings keine Anweisungen sequenziell ausgeführt, sondern die Werte von Ausdrücken durch Termersetzung berechnet. Auch hier ist es grundsätzlich möglich den Ablauf der Auswertung einer Anfrage Schritt für Schritt mit jeder angewendeten Ersetzungsregel nachzuvollziehen. Durch Sharing haben Terme jedoch keine reine Baumstruktur, sondern ähneln in Ihrer Struktur einem allgemeinen gerichteten Graphen. Dadurch ist eine reine Darstellung der Berechnungszustände in textueller Form entweder sehr unübersichtlich, oder sogar nicht ausreichend, da unter Umständen die Graphstruktur des aktuellen Terms nicht hinreichend dargestellt wird. Um also die Graphstruktur von Termen mit Sharing übersichtlich darzustellen, ist eine Visualisierung der Termgraphen notwendig.

Curry ist eine logisch-funktionale Programmiersprache, bei der während der Auswertung von Anfragen Terme mit Sharing zum Einsatz kommen und für welche eine grundlegende Visualisierung von Termgraphen während der Auswertung bereits in das ICurry-Laufzeitsystem integriert wurde.[1] Diese Visualisierung basiert allerdings auf der Nutzung von GraphViz, welches zwar ein leistungsstarkes Werkzeug zur Visualisierung von allgemeinen Graphen ist, aber wenige Möglichkeiten bietet, die ansatzweise baumähnliche Struktur von Termgraphen übersichtlich darzustellen. Hierzu gehört zB. die Anordnung der Kinder eines Funktionsknotens in der richtigen Argumentreihenfolge. Durch eine auf die Darstellung von Termgraphen zugeschnittene Implementierung eines Graph-Visualisierers lassen sich diese Probleme lösen. Ein weiteres Defizit der bestehenden Implementierung liegt in der etwas umständlichen Bedienung, bei der das Laufzeitsystem lediglich PDFs mit den dargestellten Graphen ausgibt, die dann manuell mit einem geeigneten Betrachter geöffnet werden müssen, welcher die Anzeige der Graphen in einer Diashow oder die Anzeige einzelner Schritte nebeneinander im Allgemeinen nicht unterstützt. Gegenstand dieser Arbeit ist darum die Entwicklung einer Anwendung, welche die Eingabe von Programm und Anfrage ermöglicht, dieses dann auswertet, die Berechnungszustände visualisiert und die so generierten Graphen in einer Diashow darstellt, in der die Berechnungsschritte nebeneinander dargestellt werden können.

[1] Mehr Informationen zu ICurry unter `https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/icurry`

Kapitel 2

Grundlagen

2.1. Curry

Curry ist eine funktional-logische Programmiersprache. Als solche vereinigt sie Konzepte der funktionalen und der logischen Programmierung. Funktionale Programmierung baut auf das Konzept der mathematischen Funktion auf, entsprechend bestehen funktionale Programme aus einer Menge von Funktionen, die auf Datenstrukturen mittels Fallunterscheidung und Rekursion arbeiten. Logische Programmierung hingegen baut auf der Prädikatenlogik auf und ein logisches Programm besteht aus einer Menge an Prädikaten, die beispielsweise durch Implikation definiert sind. Logische Programmierung bietet insbesondere das Konzept des Nichtdeterminismus bei der Auswertung von Ausdrücken an. Curry baut hierbei auf der Syntax von Haskell auf und erweitert es um Nichtdeterminismus und das Konzept von unbekannten Werten/freien Variablen, für die bei der Auswertung eines Ausdrucks nichtdeterministisch Werte eingesetzt werden können. Zudem verwendet es eine eigene Auswertungsstrategie, welche ähnlich der von Haskell bedarfsgesteuert und optimal in der Anzahl der durchgeführten Berechnungsschritte ist. Ausdrücke werden in Curry also nur ausgewertet, wenn ihr Wert auch tatsächlich benötigt wird.

Bei der Auswertung eines Ausdrucks werden, anders als in Haskell, die angegebenen Regeln nicht von oben nach unten ausprobiert, wobei dann die erste passende Regel als einzige angewendet wird. Stattdessen wird in Fällen mehrerer passender Ersetzungsregeln eine nichtdeterministische Berechnung durchgeführt, bei der alle passenden Regeln zur Auswertung des gegebenen Ausdrucks angewendet werden. Ein gutes Beispiel hierfür stellt die Basisoperation zur Einführung von Nichtdeterminismus in einer Berechnung in Curry dar. Diese existiert in Form des **?**-Operators, der folgendermaßen definiert ist:

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

Wie man sehen kann, sind beide Argumente der **?**-Operation ein möglicher Wert des Gesamtausdrucks, mit denen dann nichtdeterministisch weitergerechnet werden kann.

Durch die Einführung freier Variablen wird weiterhin die Berechnung von Ausdrücken wie `zs++[2] =:= [1,2]` ermöglicht (das Zeichen `=:=` wird zur Unterscheidung zwischen zu lösenden Gleichungen und bool'schen Ausdrücken verwendet). Dies geschieht, indem durch das nichtdeterministische Ausprobieren der existierenden Listenkonstruktoren (`[]` für die leere Liste und `(x:xs)` für eine Liste mit mindestens einem Element) der Wert `[1]` für `zs` gefunden wird, der die Gleichung erfüllt.

Eine weitere Möglichkeit, die sich durch den Mechanismus des nichtdeterministischen Findens von Belegungen für freie Variablen eröffnet, sind *funktionale Muster* in den linken Seiten von Regeln:

```
last :: [a] -> a
last (zs++[e]) = e
```

Mit dieser Regel kann das letzte Element einer Liste berechnet werden, indem für ein übergebenes Argument `x` zuerst die Gleichung `x =:= (zs++[e])` gelöst wird und dann `e` als das letzte Element von `x` zurückgegeben wird. Dementsprechend ist diese Regel auch nur auf Ausdrücke `x` anwendbar, für die die genannte Gleichung lösbar ist. In Haskell wäre eine solche Regel nicht erlaubt, da die Argumentmuster der linken Regelseite nicht konstruktorbasiert sind, es wird offensichtlich eine Funktion angewendet. Es lassen sich mittels eines funktionalen Musters jedoch unendlich viele konstruktorbasierte Muster repräsentieren, in diesem Fall nämlich die aller Listen, die auf einen bestimmten Wert enden (`(x:[])`, `(_:(x:[]))`, `(_:(_:(x:[])))`, usw.) [AH10].

2.1.1. Auswertungsstrategie

Die Auswertungsstrategie ist ein essentieller Teil logisch-funktionaler Programmiersprachen, denn sie bestimmt, welcher Teil eines Ausdrucks in einem Auswertungsschritt ausgewertet werden soll. Außerdem stellt sie fest, welche Regel im nächsten Schritt angewendet werden soll und wie ggf. freie Variablen instanziiert werden. Gerade die Auswahl des nächsten auszuwertenden Ausdrucks ist hierbei im Fall einer nicht-strikten, bedarfsgesteuerten Auswertungsstrategie nicht trivial.

Currys Auswertungsstrategie wird *needed narrowing* [AEH00] genannt und baut auf der grundsätzlichen Idee des *Narrowings* auf, bei dem definierte Regeln immer von links nach rechts auf einen Term angewendet werden (wie es bei der funktionalen Programmierung ebenfalls der Fall ist). Anders als bei funktionalen Programmiersprachen wird ein gegebener Term jedoch mit einer passenden linken Regelseite *unifiziert*, wie man es aus logischen Programmiersprachen kennt. Needed narrowing führt als Erweiterung die Idee ein, nur solche Unterausdrücke auszuwerten, die für die Auswertung eines Gesamtausdruckes notwendig sind, sodass es als bedarfsgesteuerte Auswertungsstrategie bezeichnet werden kann.

Um dies zu erreichen, werden bei der Auswertung eines Terms mit einer Funktionsanwendung die linken Regelseiten der entsprechenden Funktion analysiert und es wird ermittelt, ob eines der Argumente in allen linken Regelseiten als Kunstruktorterm angegeben ist. Da in diesem Fall im tatsächlichen Argument auch einer der entsprechenden Konstruktoren an der Wurzel stehen muss, wird das Argument entweder soweit ausgewertet, bis ein Kunstruktor an der Wurzel steht, oder im Fall einer freien Variable nichtdeterministisch mit den passenden Konstruktoren instanziiert. Um eine entsprechende Funktionalität zu erreichen, verwendet die Auswertungsstrategie *definierende Bäume*, die die Termersetzungsregeln einer Operation eines Curry-Programmes hierarchisch strukturiert [Han13].

Betrachten wir z.B. folgende (nicht minimal definierte) Operation zur Berechnung des logischen Oders:

```
(||) :: Bool -> Bool -> Bool
_     || True  = True
True  || False = True
False || False = False
```

In dieser Operation wird zuerst eine Fallunterscheidung im zweiten Argument für die Fälle `True` und `False` vorgenommen. Im ersten Fall muss das erste Argument zur Ermittlung des Ergebnisses gar nicht mehr betrachtet werden, im zweiten Fall wird eine weitere Fallunterscheidung über dem ersten Argument vorgenommen um das Ergebnis zu bestimmen. Dieser Prozess wird über definierende Bäume gesteuert, welche die Fallunterscheidungen und die Reihenfolge in der sie unternommen werden, codieren. Mit dieser Auswertungsstrategie werden die geforderten Argumente nicht, wie z.B. in Haskell, von links nach rechts ausgewertet. Dadurch würde beispielsweise die Berechnung des Ausdrucks `loop || True` mit der oben angegebenen Definition in Curry immer das Ergebnis `True` liefern, auch wenn die Berechnung von `loop` nicht terminiert. In Haskell würde die Berechnung im Fall eines nicht-terminierenden `loops` hingegen nicht terminieren.

2.1.2. Realisierung nichtdeterministischer Berechnungen

An dieser Stelle ist es wichtig, etwas genauer zu betrachten, wie nichtdeterministische Berechnungen in den betrachteten Termgraph-orientierten Berechnungen realisiert werden.

Wird eine nichtdeterministische Berechnung gestartet, so werden die nichtdeterministischen Verzweigungen der Berechnung in den zugehörigen Termgraphen über sog. *Choice-Knoten* codiert. Betrachten wir z.B. die Berechnung einer einfachen Anfrage mit Nichtdeterminismus wie `True ? False`, wobei die **?**-Operation wie in Abschnitt 2.1 definiert sei. Der entsprechende Termgraph ist in Abbildung 2.1a dargestellt. Man kann hier erkennen, dass der dargestellte Choice-Knoten noch über eine eigene ID, die *Choice-ID* verfügt. Diese wird verwendet, damit Choice-Knoten identifizierbar bleiben, auch wenn die Knoten-ID eines Choice-Knotens sich durch Umformungen des Termgraphen verändert.

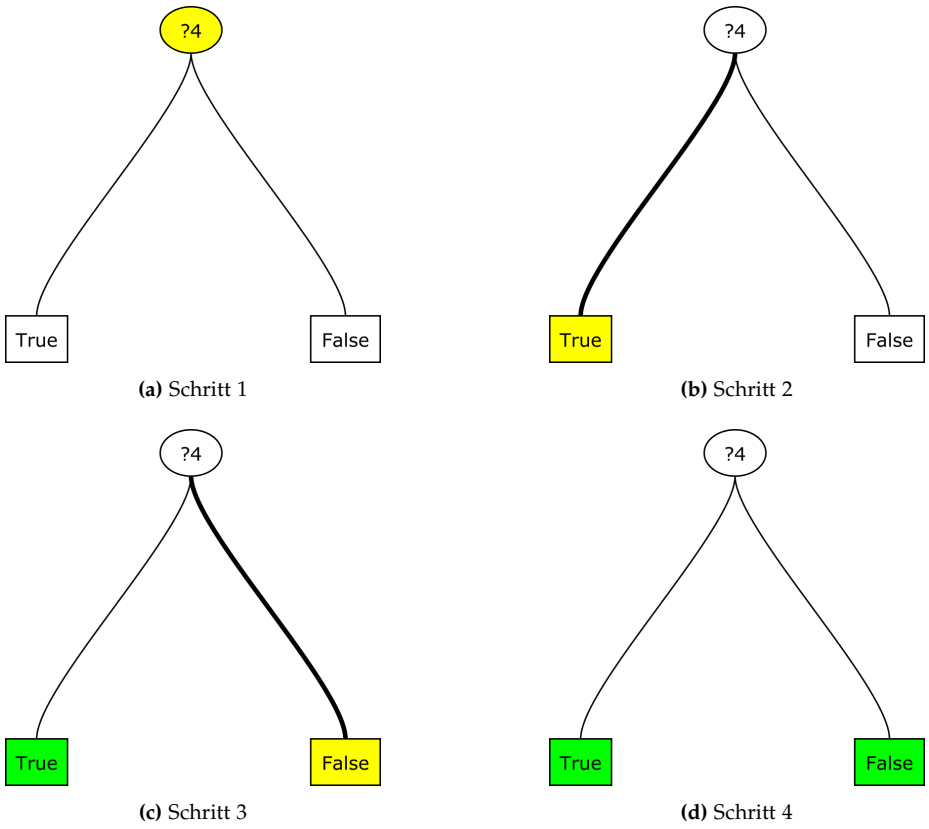
Um nun die Ergebnisse einer nichtdeterministischen Berechnung sequenziell zu ermitteln, werden die jeweiligen Kind-Terme (bzw. *Choice-Alternativen*) eines Choice-Knotens nacheinander berechnet. Auskunft darüber, welche Choice-Alternative eines Choice-Knotens in einem gegebenen Berechnungsschritt ausgewertet wird, gibt eine partielle Abbildung von Choice-IDs auf Indizes von Alternativen. Diese Abbildungen von Choice-Knoten auf Choice-Alternativen ist in den Abbildungen 2.1b und 2.1c durch dick gezeichnete Verbindungslinien dargestellt.

2.2. Sharing und Termgraphen

Da Curry eine bedarfsgesteuerte Auswertungsstrategie verwendet, kann es vorkommen, dass in einem Gesamtterm einzelne Subterme mehrfach vorkommen. Betrachten wir z.B. die folgende Operation:

```
andself :: Bool -> Bool
andself x = x && x
```

Wird nun z.B. eine komplexere Anfrage, wie `andself (True || False)`, ausgewertet, könnte der Eindruck entstehen, dass der innere Term `True || False` mehrfach ausgerechnet wird, da die Anfrage im ersten Schritt aufgrund der bedarfsgesteuerten Auswertung zu `(True || False) && (True || False)` ausgewertet wird. Um derartige unnötige doppelte Berechnungen zu vermeiden und gleichzeitig bei einer bedarfsgesteuerten Auswertungsstrategie zu bleiben, wird in Curry *Sharing* verwendet. Sharing beschreibt die Praxis, bei mehrfachem Vorkommen des selben Subterms in einem Gesamtterm eine Referenz auf den entsprechenden Subterm zu verwenden, anstatt den Term selbst zu kopieren. Abbildung 2.2 zeigt den Unterschied am Beispiel `(True || False) && (True || False)` graphisch, dabei sind ausgehende Kanten immer an der Unterseite und eingehende an der Oberseite eines Knotens.



(a) Schritt 1 **(b)** Schritt 2 **(c)** Schritt 3 **(d)** Schritt 4

Abbildung 2.1. Berechnungsschritte des Ausdrucks `True ? False`.
Im nächsten Schritt auszuwertende Knoten sind gelb, Ergebnisse sind grün markiert.

Terme in einer Berechnung in Curry haben somit also keine Baumstruktur mehr, sondern eine Struktur, die der eines *baumähnlichen* gerichteten Graphen nahe kommt, aber durch die Möglichkeit mehrerer gleichgerichteter Kanten zwischen zwei Knoten auch nicht mehr unter die herkömmliche Definition eines gerichteten Graphen fällt:

1. Definition (Termgraph)**.** Ein *Termgraph* $TG = (V, E, r)$ besteht aus einer Knotenmenge V, einem ausgezeichneten Wurzelknoten $r \in V$ und einer eindeutigen Kantenabbildung $E : V \to lists(V)$, wobei die Menge $lists(V)$ alle Listen über Elementen aus V beschreibt und wie folgt induktiv definiert ist:

$$() \in lists(V)$$
$$v : vs \in lists(V), \text{ falls } v \in V \wedge vs \in lists(V)$$

Hier lässt sich schon das zentrale Problem erahnen, das es bei der Visualisierung von Termgraphen mit Sharing zu bewältigen gilt: Wie wird die Darstellung von Kindknoten realisiert, wenn aufgrund von Sharing die Kinder eines Funktions- oder Konstruktorknotens nicht in der Reihenfolge der Argumente des Funktions- bzw. Konstruktorknotens von links nach rechts darstellen lassen oder wenn sogar Kreise im Graphen vorkommen? Eine genauere Betrachtung des Problems erfolgt in Kapitel 3.

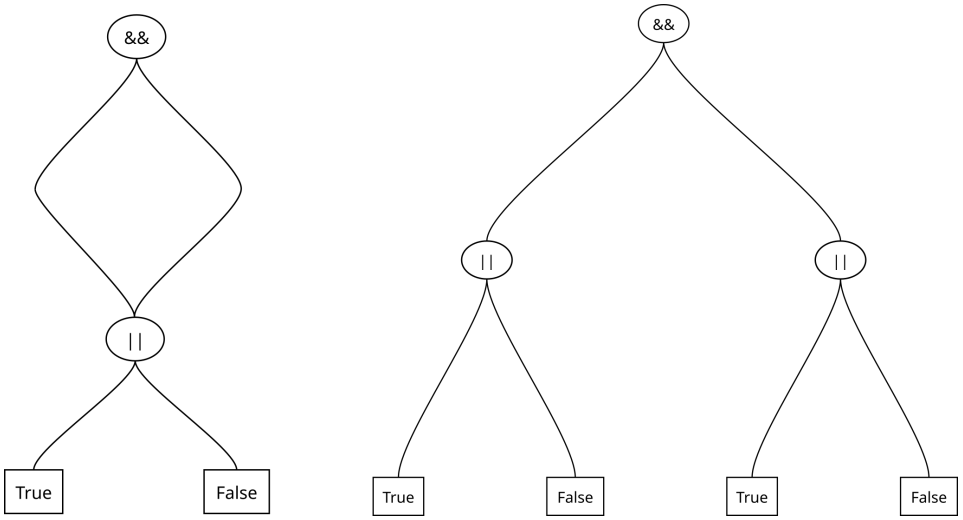


Abbildung 2.2. Darstellung des ersten Ableitungsschrittes von `andself (True || False)` mit (links) bzw. ohne Sharing (rechts)

2.3. ICurry

Im Rahmen dieser Arbeit soll der entwickelte Graphvisualisierer auch in ein bestehendes Curry-Laufzeitsystem integriert werden. Komplexe und funktionsreiche Laufzeitsysteme wie PAKCS[1] oder KiCS2[2] übersetzen jedoch Curry in die Sprachen Prolog bzw. Haskell und führen intern keine direkten Berechnungen mit Termgraphen aus. Deshalb eignen sich Laufzeitsysteme wie das ICurry-Laufzeitsystem hierfür fürs Erste besser, da es mit einem eigenen einfachen Interpreter ausgestattet ist. Ebenso ist es, wie in der Einführung bereits erwähnt, schon mit einem einfachen Graphvisualisierer ausgestattet.

2.3.1. Pull-Tab Schritte

Zur Verwaltung nichtdeterministischer Berechnungen müssen Berechnungen in mehrere Zweige kopiert werden. Um dies in den benötigten Argumenten von Funktions- oder Konstruktoranwendungen zu realisieren, verwendet der ICurry-Interpreter *pull-tabbing*. Grundsätzlich wird durch pull-tabbing eine nichtdeterministische Verzweigung in einem benötigten Argument einer Operation aus dieser herausbewegt [Ant+20].

Da die logische Negationsfunktion `not` den Wert ihres Argumentes benötigt, ist das Folgende ein Beispiel für einen pull-tab Schritt:

```
not (True ? False) → (not True) ? (not False)
```

2.4. XML und SVG

Der im Rahmen dieser Arbeit entwickelte Graphvisualisierer arbeitet mit einer Zwischendarstellung von Termgraphen im XML-Format und gibt seinerseits visualisierte Graphen im SVG-Format (kurz

[1] `https://www.informatik.uni-kiel.de/~pakcs/`

[2] `https://www-ps.informatik.uni-kiel.de/kics2/`

für *Scalable Vector Graphics*) aus, welches selbst wiederum auf XML basiert. In diesem Sinn widmet sich der nächste Abschnitt einer kurzen Einführung in diese beiden Formate.

XML (kurz für *Extensible Markup Language*) wurde entwickelt, um den Austausch strukturierter Daten über das Internet zu standardisieren und zu vereinfachen. Designziele waren dabei unter anderem eine einfache Verwendung von XML bei der Softwareentwicklung, breite Anwendbarkeit, eine präzise Syntax, bei der aber Menschenlesbarkeit beibehalten wird.

XML-Daten folgen grundsätzlich einer Baumstruktur. Ein wohlgeformtes XML-Dokument besteht deshalb aus einem optionalen Prolog, der eine XML- und/oder eine Deklaration des Dokumenttypen enthalten kann, und der Wurzel des Baumes, in dem die Daten des Dokuments enthalten sind. Diese Wurzel ist hierbei ein *XML-Element*. XML-Elemente repräsentieren die Knoten in der Baumstruktur eines XML-Dokuments, wobei jedes XML-Element eine beliebig lange Liste an Attribut-Wert-Paaren, sowie beliebig viele weitere XML-Elemente oder Text beinhalten kann. Definiert wird ein XML-Element entweder durch einen Start- und Endtag, zwischen denen der erwähnte Inhalt des Elements steht, oder durch einen Leertag, der ein XML-Element ohne Inhalt beschreibt. Nachfolgend ist ein Beispiel für ein wohlgeformtes XML-Dokument:

```xml
<?xml version="1.0" standalone="yes"?>
<!-- Dies ist ein Kommentar -->
<personen>
  <!-- Ein Element mit drei Attributen -->
  <person first_name="Boris" last_name="Robinsky" id="2">
    <freunde>
      <!-- Ein Element mit Text als Inhalt -->
      <id>3</id>
    </freunde>
  </person>
  <person first_name="Lara" last_name="Becker" id="3">
    <freunde>
      <id>2</id>
      <id>6</id>
    </freunde>
  </person>
  <person first_name="Wladimir" last_name="Plötzbogen" id="1">
    <!-- Ein Leertag -->
    <freunde />
  </person>
</personen>
```

Eine genaue Beschreibung des XML-Standards sowie eine Beschreibung wohlgeformter XML-Dokumente durch Produktionsregeln findet sich in [Bra+08].

Ein Anwendungsbeispiel für XML ist das SVG-Format für Vektorgrafiken. Das Wurzelelement des XML-Baums ist hierbei immer ein Element mit dem Namen `svg` und einigen Attributen, die Metainformationen über das dargestellte Bild, wie z.B. Höhe und Breite, enthalten. Der Inhalt des Wurzelelements kann dann aus anderen XML-Elementen bestehen, die z.B. die Darstellung von geraden Linien, Ellipsen, Rechtecken, Polygonen oder komplexerer Linien bewirken. Eine einfache Ellipse mit einer Beschriftung in der Mitte kann z.B. folgendermaßen dargestellt werden:

```
<ellipse cx="50.0" cy="50.0" rx="20.0" ry="15.0" fill="none" stroke="black" />
<text x="50.0" y="55.0" text-anchor="middle" font-size="12">Text</text>
```

2.4.1. Das SVG-path-Element

Ein *Path*-Element repräsentiert den Umriss einer Form, die gefüllt oder umrandet werden kann. Hierbei wird das Konzept eines aktuellen Punktes genutzt, der anfangs an einem Startpunkt ist und sich mit jedem weiteren Abschnitt des Pfades an den Endpunkt des letztes Abschnitts bewegt. Definiert wird die Form eines Path-Elements über das Attribut *b* des Elements, welches als Wert wiederum eine Reihe an Kommandos hat, die die Form des Pfades definieren. An dieser Stelle eine kleine Auswahl der möglichen Kommandos:

- M (absolut) oder m (relativ): Der aktuelle Punkt wird an die angegebenen Koordinaten verschoben, ohne etwas zu zeichnen. Ein Pfad muss mit diesem Kommando beginnen.
- Z oder z: Der Pfad wird beendet, indem eine gerade Linie vom aktuellen Punkt zum Startpunkt des Pfades gezeichnet wird. Durch Auslassen dieses Kommandos kann in Verbindung mit den Attributen `stroke="<farbe>" fill="none"` des Path-Elements eine einfache Linie gezeichnet werden.
- L (absolut) oder l (relativ): Eine gerade Linie wird vom aktuellen Punkt zu den angegeben Koordinaten gezeichnet. Sind mehrere Koordinatenpaare angegeben, wird ein Polygonzug gezeichnet.
- Q (absolut) oder q (relativ): Eine quadratische Bézierkurve[3] wird vom aktuellen Punkt mit dem ersten Koordinatenpaar als Kontrollpunkt zum zweiten Koordinatenpaar gezeichnet.
- C (absolut) oder c (relativ): Eine kubische Bézierkurve[3] wird vom aktuellen Punkt mit dem ersten und zweiten Koordinatenpaar als Kontrollpunkte zum dritten Koordinatenpaar gezeichnet.

Path-Elemente wurden für die Verbindungslinien der Graphen in Abbildung 2.2 verwendet, von denen eine wie folgt definiert wurde:

```
<path d="M 280.0,108 C 280.0,140 140.0,234 140.0,265.0" stroke="black" fill="none" />
```

Die genaue Spezifikation des SVG-Standards findet sich bei [McC+11].

Wir haben nun also die Sprache Curry, die Idee ihrer Auswertungsstrategie, das ICurry-Laufzeitsystem sowie die Grundlagen der XML- und SVG-Formate kennengelernt und damit die Grundlagen für den eigentlichen Gegenstand dieser Arbeit gelegt. Bevor wir uns aber der Implementierung zuwenden, betrachten wir im nächsten Kapitel zunächst die Anforderungen, die an den zu implementierenden Graphvisualisierer gestellt werden.

[3] Eine Kurzübersicht zu Bézierkurven findet sich bei Walser: Die Modellierung des schönen Scheins, S.10ff. (`http://www.mathematikinformation.info/pdf2/MI55Walser.pdf`, zuletzt abgerufen am 30.03.22, 12:43) oder mit anschaulichen Animationen unter: `https://de.wikipedia.org/wiki/B%C3%A9zierkurve` (zuletzt abgerufen am 30.03.22, 12:35)

Kapitel 3

Anforderungen

In den Anforderungen an den Graphvisualisierer liegt der Fokus auf einer besseren Übersichtlichkeit der Graphen im Vergleich zur bestehenden Implementierung mit GraphViz sowie auf einer höheren Benutzerfreundlichkeit als von einer Kommandozeilenapplikation erreicht werden kann. Letztlich ist aber auch eine höhere Modularisierung des neuen Graphvisualisierers erwünscht, damit die Visualisierung von Termgraphen auch in andere Curry-Laufzeitsysteme eingebaut werden kann ohne den Visualisierer für jedes Laufzeitsystem neu implementieren zu müssen.

3.1. Darstellung der Termgraphen

In Abbildung 3.1 ist die Darstellung der Baumstruktur des Terms eines beispielhaften bool'schen Ausdrucks zu sehen. Der Aufbau des Terms ist durch die klare Einteilung aller Knoten in Ebenen sowie durch die Darstellung der Kinder jedes Knotens in der Reihenfolge der Argumente der entsprechenden Funktion klar erkennbar. Die Darstellung von Knoten in eindeutigen Ebenen sowie in einer eindeutigen Argumentreihenfolge ist bei den zu Termen mit Sharing zugehörigen Termgraphen allerdings nicht oder nur bedingt möglich, insbesondere da Kreise im Termgraphen enthalten sein können. Dementsprechend ist für die bestmögliche Übersichtlichkeit in der Darstellung von Termgraphen eine möglichst gute Annäherungen an eine ideale Baumdarstellung nötig.

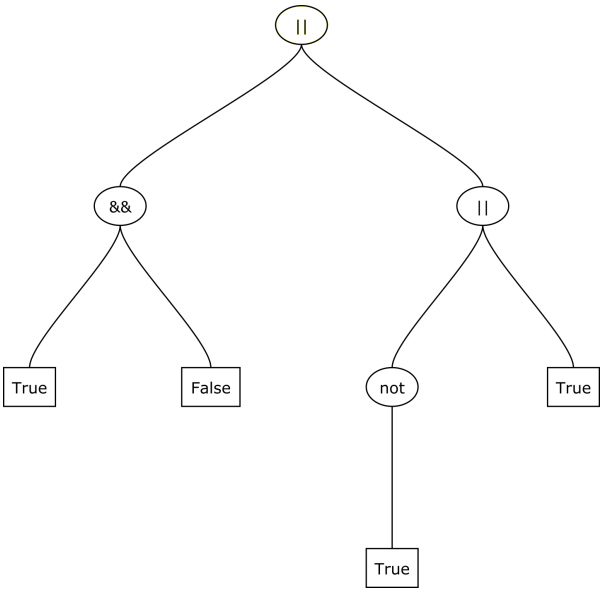


Abbildung 3.1. Termgraph eines beispielhaften bool'schen Ausdrucks

Bezüglich der Darstellung von Knoten in eindeutigen Ebenen ist die vermutlich beste Approximation für Knoten, die durch Sharing Nachfolger mehrerer Knoten sind und außerhalb eines Kreises liegen, die Darstellung auf der nächstniedrigeren Ebene des tiefstgelegenen Vorgängerknotens (vgl. Knoten „True“ und „False“ in Abbildung 3.2). Für Knoten innerhalb eines Kreises gilt weiterhin, dass der tiefstliegende Elternknoten nicht über einen Kreis erreicht werden darf, da sich sonst eine unendlich tiefe Ebene für den entsprechenden Knoten ergeben würde.

Die bestehende Implementierung wird einer anschaulichen Darstellung der Knoten auf Ebenen bereits gerecht (vgl. Abbildungen 3.2 und 3.3). Allerdings besteht ein wichtiges Defizit in der Übersichtlichkeit in der Darstellung der Argumente von Funktions- bzw. Konstruktoranwendungen, sobald

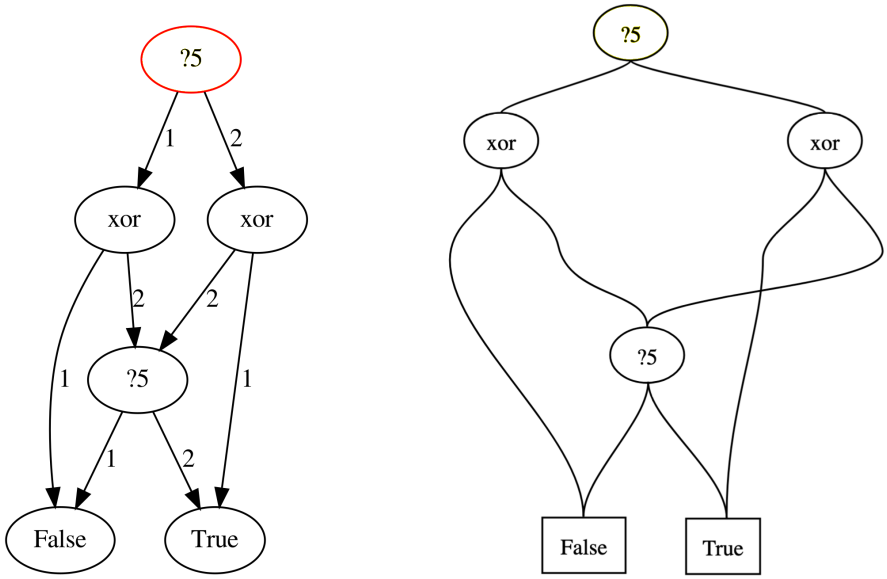


Abbildung 3.2. Vergleich der Graphdarstellung der bestehenden Implementierung (links) und einer Idee für eine übersichtlichere Darstellung (rechts) für einen Berechnungsschritt des xorSelf-Ausdrucks.

Sharing für eines der Argumente verwendet wird. Da die Kanten von GraphViz i.d.R. als gerade Verbindung zwischen zwei Knoten dargestellt werden, ist das Erkennen der Termgraphstruktur allein auf Basis des Layouts oftmals nicht möglich, sodass zusätzlich die Kanten mit Nummern beschriftet werden, damit das Erkennen der Reihenfolge der Kinder eines Knotens möglich ist (vgl 3.2 und 3.3 links). Diese Art der Darstellung enthält zwar alle notwendigen Informationen, macht die Struktur des Termgraphen allerdings nicht so anschaulich wie es erstrebenswert wäre.

Zur Verbesserung der Übersichtlichkeit sollen in der neuen Implementierung Verbindungen zu Nachfolgerknoten, die aufgrund von Sharing nicht an eine Position gezeichnet werden können, die der Darstellung der Argumentknoten eines Funktions- oder Konstruktorknotens von links nach rechts entsprechen würde, durch Stützpunkte gezeichnet werden. Diese Stützpunkte befinden sich an der idealen Position eines Argumentknotens, damit verdeutlicht wird, an welcher Stelle in der Argumentliste einer Funktions- bzw. Konstruktoranwendung sich der dem Teilgraphen entsprechende Teilterm befindet. In Abbildung 3.2 wird dies beispielhaft für einen kreisfreien Graphen gezeigt. Weiterhin sollen eingehende und ausgehende Kanten übersichtlicher dargestellt werden, indem eingehende Kanten immer mit der oberen Seite eines Knoten verbunden werden und ausgehende Kanten immer mit der unteren Seite eines Knotens. Dies trägt vorrangig zu einer verbesserten Übersichtlichkeit bei Graphen mit Kreisen bei (vgl. Abbildung 3.3).

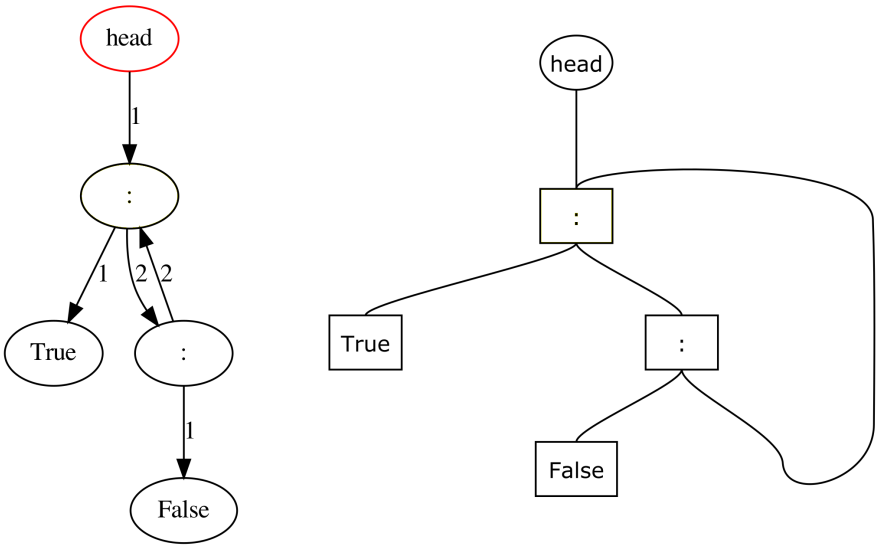


Abbildung 3.3. Vergleich der Graphdarstellung der bestehenden Implementierung (links) und einer Idee für eine übersichtlichere Darstellung (rechts) für einen Berechnungsschritt einer Anfrage mit rekursivem *let*.

3.1.1. Termgraphen als Bäume

Für Situationen, in denen die Termgraphen einer Berechnung zu unübersichtlich werden kann es ggf. wünschenswert sein, die Darstellung nicht in Form tatsächlicher Graphen, sondern von Bäumen zu unternehmen, in denen Knoten mit Sharing entsprechend mehrfach dargestellt werden. Um in Termgraphen, die auf diese Art dargestellt worden sind, mehrfach dargestellte Knoten als dieselben Knoten identifizieren zu können, sollen zusammengehörige Knoten farbig markiert werden. Weiterhin muss bei Graphen mit Kreisen in dieser Darstellungsform verhindert werden, dass unendliche Bäume gezeichnet werden. Dies soll durch eine einfache, nutzerwählbare Limitierung der Darstellungstiefe erfolgen.

3.2. Nutzerfreundliche Bedienung

Der Termgraphvisualisierer wurde im Rahmen dieser Arbeit unter anderem als Erweiterung für das ICurry-Laufzeitsystem entwickelt. In diesem Rahmen ist er allein über die Kommandozeile zu bedienen und die Ausgabe visualisierter Termgraphen erfolgt in Form von SVG-Dateien. Um jedoch eine, vor allem für die Anzeige der Termgraphen, benutzerfreundlichere Bedienungsoption des Termgraphvisualisierers zu Verfügung zu stellen, soll weiterhin eine Webanwendung entwickelt werden. Mittels dieser sollen ein Programm und eine Anfrage in eine grafische Nutzeroberfläche eingegeben werden können, woraufhin im Hintergrund die Visualisierung der Termgraphen der einzelnen Berechnungsschritte erfolgt, deren Ergebnisse dann in einer Diashow dargestellt werden. Durch die Diashow kann man sich dann einfach „durchklicken“, beliebig viele Einzelschritte nebeneinander darstellen und sich bei Bedarf den Quelltext des eingegebenen Programms anzeigen lassen.

3.3. Modularität

Der letzte Teil der Anforderungen beschäftigt sich nunmehr mit der Modularität des Visualisierers: Um den Visualisierer möglichst flexibel in jedem beliebigen Laufzeitsystem einsetzen zu können, soll eine textuelle Zwischendarstellung der Termgraphen entwickelt werden, auf deren Basis dann die tatsächliche Visualisierung der Termgraphen stattfindet. Zu einem Laufzeitsystem muss anschließend entsprechend nur die Funktionalität hinzugefügt werden, die Termgraphen von Berechnungsschritten in diesem Zwischenformat auszugeben, um die Visualisierung zu ermöglichen. Die bestehende Implementierung war direkt in den Interpreter des ICurry-Laufzeitsystems integriert und wird den gestellten Anforderungen damit nicht gerecht. Die textuelle Zwischendarstellung sollte entsprechend so geartet sein, dass die Umwandlung eines Termgraphen von einer laufzeitsystemspezifischen Darstellung in die textuelle Darstellung einfach zu implementieren ist und alle Informationen enthält, die zur Visualisierung eines beliebigen Termgraphen notwendig sind. Für die Visualisierung notwendige Informationen sind vor allem folgende:

- Die Knoten und Kanten des Graphen,
- die Information, welcher Knoten momentan aktiv ist bzw. ausgewertet wird,
- die Information, ob ein gegebener Knoten eine Lösung repräsentiert und
- Informationen über die aktuellen Choice-Alternativen.

Somit stehen nun die Anforderungen an den zu implementierenden Graphvisualisierer fest: Übersichtliche Darstellung, nutzerfreundliche Bedienbarkeit und Modularität. Mit diesem Wissen wenden wir uns in den nächsten beiden Kapiteln der Implementierung zu.

Kapitel 4

Implementierung der Graph-Visualisierung

Wie bereits erwähnt, erfolgt im Rahmen dieser Arbeit die Implementierung des Graphvisualisierers als eigenständiges Tool, welches Termgraphen aus einem im Folgenden spezifizierten Zwischenformat graphisch darstellt, aber ebenso eine Integration desselben in das ICurry-Laufzeitsystem.

4.1. Zwischenformat

Für das textuelle Zwischenformat wurde eine XML-Formatierung gewählt, da XML ein weit verbreitetes und einfach nutzbares Format für die textuelle Darstellung strukturierter Daten ist und für Curry bereits ein Modul zur einfachen Handhabung von XML-Daten existiert.
Hierbei soll nunmehr mit einem einzelnen XML-Dokument die vollständige Berechnung einer Anfrage für ein Curry-Programm repräsentiert werden, deshalb beinhaltet das Wurzelelement eine Liste von Berechnungsschritten/Termgraphen. Entsprechend ist der Name des Wurzelelements `graphlist`. Der Inhalt des Wurzelelements ist dementsprechend eine Sequenz aus `graph`-Elementen, die jeweils einen Berechnungsschritt bzw. Termgraphen der Anfrageauswertung repräsentieren.

Der Inhalt der `graph`-Elemente besteht dann wiederum aus einer Liste von `node`-Elementen zur Repräsentation der Knoten und Kanten des Graphen, einem `root`-Element, dessen Inhalt die ID der Wurzel des aktuellen Termgraphen ist sowie eine Liste von `ChoiceMapping`-Elementen, die jeweils über die Attribute `from` und `to` verfügen und damit eine gewählte Choice-Alternative (siehe Abschnitt 2.1.2) repräsentieren. Bei dieser ist `from` die ID des Choice-Knotens, zu dem die gewählte Choice-Alternative gehört, `to` hingegen über die möglichen Werte `1` und `2` anzeigt, ob die gewählte Choice-Alternative des Knotens gerade durch sein erstes oder zweites Kind repräsentiert wird. Wird gerade kein Teilterm unterhalb eines gegebenen Choice-Knotens berechnet wird für diesen Knoten entsprechend auch keine Choice-Alternative abgespeichert.

Die `node`-Elemente beinhalten nunmehr jeweils folgende Elemente:

- Ein `id`-Element, das die ID des repräsentierten Knotens als Inhalt hat,
- ein `type`-Element, das die Information über den Typen des repräsentierten Knotens enthält (dies können `funcNode` für Funktionsknoten, `ConsNode` für Konstruktorknoten, `ChoiceNode` für Choice-Knoten oder `FreeNode` für Knoten von freien Variablen sein),
- ein `label`-Element, welches das Label des Knotens, also den Namen der durch den Knoten repräsentierten Funktion oder Konstruktors bzw. Choice-ID des Choice-Knotens, als Inhalt hat,
- ein `isActive`-Element, das einen bool'schen Wert beinhaltet, der angibt, ob der Knoten im aktuellen Berechnungsschritt der aktive Knoten ist, d.h. ob der Knoten im aktuellen Berechnungsschritt ausgewertet werden soll,
- ein `isResult`-Element, das ebenfalls durch einen bool'schen Wert angibt, ob der Knoten ein bereits berechnetes Ergebnis repräsentiert und

▷ ein `children`-Element, das eine Auflistung an `nodeId`-Elementen beinhaltet, die wiederum jeweils die Id eines Kindknotens des aktuellen Knotens beinhaltet.

Damit enthält das Zwischenformat alle für die Visualisierung notwendigen Informationen, sodass wir uns im nächsten Abschnitt eben jener widmen werden.

4.2. Visualisierer

Wie in Abschnitt 3.1 erwähnt verfügt der Graphvisualisierer über zwei Darstellungsmodi, die größtenteils unabhängig voneinander implementiert wurden. Die Implementierungsdeteils werden entsprechend in den nächsten beiden Abschnitten betrachtet.

4.2.1. Termgraphen als Graphen

Die Visualisierung von Termgraphen als Graphen erfolgt in drei Schritten:

Auslesen der Daten aus dem XML-Zwischenformat

Im ersten Schritt werden die Informationen über alle Schritte einer Berechnung aus dem XML-Zwischenformat ausgelesen und in einer Liste von Curry-Tripeln `(Graph, [ChoiceMapping], NodeID)` umgewandelt, wobei jedes Listenelement einen Berechnungsschritt repräsentiert. Hierbei ist `Graph` ein Typ-Alias für eine Liste aus `Node`-Objekten, welche wie folgt definiert sind:

```
--              Type     ID     Label  Children Active Result
data Node = Node NodeType NodeID String [NodeID] Bool   Bool
```

Der Typ `ChoiceMapping` ist wiederum ein Tupel aus zwei `Int`-Werten, sodass über eine Liste aus `ChoiceMapping`-Objekten die partielle Abbildung von Choice-Knoten auf Choice-Alternativen dargestellt werden kann und `NodeID` ist wiederum nur ein Typ-Alias für `Int` und der Wert im Tripel zeigt an, welcher Knoten die Wurzel des darzustellenden Termgraphen ist.
Man kann also erkennen, dass dieses erste Zwischenformat die gleichen Informationen wie die XML-Zwischendarstellung enthält.

Erzeugung des Formats zur Visualisierung

Im zweiten Schritt sollen die Daten nun für die Visualisierung vorbeireitet werden, indem Informationen, wie z.B. die Färbung der Knoten, vor dem tatsächlichen Zeichenprozess berechnet werden. Hierfür wird nun ein dem Graph-Typen ähnlicher Datentyp `DGGraph` verwendet. Dieser Typ ist wiederum nur ein Alias für eine Liste aus `DGNode`-Objekten, welche folgendermaßen definiert sind:

```
data DGNode = DGNode {
        nodeTypeDG     :: NodeType,
        nodeIDDG       :: NodeID,
        labelDG        :: String,
        childrenDG     :: [NodeID],
        strokeColourDG :: String,
        fillColourDG   :: String,
        depthDG        :: Int
}
```

Wie man sieht, sind im Vergleich zum `Node`-Typen die Informationen über Färbung der Umrandung und Füllung sowie Darstellungstiefe der Knoten hinzugekommen. Die Färbung der Umrandung ist im Fall der Visualisierung tatsächlicher Graphen immer schwarz, die Färbung der Füllung ist gelb, wenn ein Knoten gerade aktiv ist, bzw. grün, wenn er ein Ergebnis repräsentiert.

Der Wert `depthDG` speichert die Darstellungstiefe eines Knotens. Für eine möglichst übersichtliche Visualisierung der Termgraphen, in der von unten nach oben verlaufende Kanten außerhalb von Zyklen vermieden werden, sollen Knoten immer so tief wie möglich gezeichnet werden. Dementsprechend soll die Darstellungstiefe eines Knotens grundsätzlich der Länge des Längsten nicht-zyklischen Pfades von der Wurzel zum Knoten entsprechen.

Die Berechnung der Darstellungstiefe eines Knotens könnte nun realisiert werden, indem der Graph mit Tiefen- oder Breitensuche von der Wurzel aus nach dem darzustellenden Knoten durchsucht wird und das Maximum der gefundenen Pfadlängen ermittelt wird. Da von der Wurzel jedoch alle Knoten eines Termgraphen erreichbar sind, muss auf diese Weise zur Berechnung der Tiefe eines Knotens jedoch der gesamte Graph durchsucht werden. Hier schafft ein rückwärts-Durchsuchen des Graphen vom Zielknoten zur Wurzel Abhilfe, da auf diese Weise bei kreisfreien Graphen nur direkte Wege vom Zielknoten zur Wurzel durchsucht werden. Um ein rückwärtiges Durchsuchen der Termgraphen effizient zu ermöglichen muss noch eine Abbildung *preds* von Knoten auf die Liste ihrer Vorgänger berechnet werden, da zu jedem Knoten im bestehenden Graph-Datentypen nur die Nachfolgeknoten unmittelbar bekannt sind. Mit zusätzlicher Memoisation der Knotentiefen aller Vorgänger wird der Aufwand für die Berechnung aller Knotentiefen weiter optimiert.

Um die Kanten zwischen einem Knoten und seinen Vorgängern ordentlich darstellen zu können wenn der entsprechende Knoten mehrere direkte Vorgänger hat wird das Verfahren erweitert: Knoten, die mehr als einen direkten Vorgänger haben, werden eine zusätzliche Ebene tiefer dargestellt. Dadurch ergibt sich folgende rekursive Definition für die Darstellungstiefe eines Knotens, durch welche die kaskadierende Auswirkung dieser Ebenenverschiebung bei nicht-Blatt-Knoten berücksichtigt wird. Hierbei ist über einer Menge M die Funktion $len : lists(M) \to \mathbb{N}$ intuitiv als die Länge der Liste definiert.

$$depthDG(node) = \begin{cases} 0, \text{ falls } node = r \text{ (}node\text{ ist die Wurzel)}; \\ max(\{depthDG(x) \mid x \in preds(node)\}) + 2, \text{ falls } len(preds(x)) > 1 \wedge node \neq r; \\ max(\{depthDG(x) \mid x \in preds(node)\}) + 1, \text{ falls } len(preds(x)) = 1 \wedge node \neq r.^{1} \end{cases}$$

[1] Eine Verbesserung ist, Vorgängerknoten auf einer tieferen Ebene aus der Menge $preds(x)$ zu entfernen, um eine unnötig tiefe Darstellung von Knoten zu vermeiden.

Visualisierung der Graphen

Aus dem letzten Format werden dann letztlich visualisierte Graphen im SVG-Format erzeugt. Hierfür werden zuerst die Dimensionen für die Zeichenfläche berechnet, damit auch größere Termgraphen gut dargestellt werden können: Hierbei sei *maxDepth* die maximale Darstellungstiefe der Knoten eines Termgraphen, *deepestNodes* die Menge aller Knoten mit Darstellungstiefe *maxdepth* und *breadth* die *Visualisierungsbreite* der Wurzel des Termgraphen.

Die Visualisierungsbreite ist dabei grundsätzlich die Breite (in Knoten), die zur Darstellung eines, bei einem bestimmten Wurzelknoten beginnenden, Subgraphen auf der Zeichenfläche benötigt wird. Da in Termgraphen mit Sharing diese benötigte Breite davon abhängig ist, welche Knoten bereits gezeichnet wurden und welche Knoten innerhalb des Subgraphen durch Sharing über mehrere Pfade erreichbar sind, ergibt sich die folgende Definition:

2. Definition (Visualisierungsbreite)**.** Sei $G = (V, E, r)$ ein Termgraph, $v \in V$ ein Knoten, E wie in Definition 1 eine Abbildung von Knoten auf die Liste seiner Nachfolgerknoten und $drawn \subseteq V$ eine Menge, die alle in einem Visualisierungsprozess bereits gezeichneten Knoten beinhaltet. Dann ist die *Visualisierungsbreite* von v definiert durch $vbreadt(v, drawn)$, wobei die Funktion *vbreadth* wie folgt definiert ist:

Function vbreadth(v, vis)

```
1 if succs(v)\vis = ∅ then
2 |   return 1;
3 else
4 |   sum ← 0;
5 |   foreach x ∈ E(v) do
6 |   |   sum ← sum + vbreadth(x, vis);
7 |   |   Füge alle im rekursiven Afruf besuchten Konten zu vis hinzu;
8 |   return sum;
```

Zu Beginn des Visualisierungsprozesses wird dann die minimale Höhe h_{min} und Breite w_{min} der Zeichenfläche wie folgt berechnet:

$$w_{min} = breadth * 140$$

$$h_{min} = graphheight * 80, \text{ mit}$$

$$graphheight = \begin{cases} maxdepth + 1, \text{ falls } \bigcup_{v \in deeptestNodes} Set(E(v)) = \emptyset; \\ maxdepth + 2, \text{ sonst.} \end{cases}$$

Dabei ist für Listen über einer beliebigen Menge M die Funktion $Set : lists(M) \to \mathcal{P}(M)$ über folgende Äquivalenz definiert:

$$x \in l \Leftrightarrow x \in Set(l)$$

Es wird dann $max(w_{min}, h_{min})$ als Höhen- und Breitendimension für die Zeichenfläche verwendet, um eine ansprechendere Darstellung der Grafiken in der Diashow der Webanwendung (Kapitel 5) zu ermöglichen.

Im nächsten Schritt wird die Höhe der Zeichenfläche in *graphheight* Ebenen unterteilt. In Fällen, in denen Knoten mit Darstellungstiefe=*maxDepth* ausgehende Kanten haben, wird $maxdepth + 2$ verwendet, damit unter der untersten Knotenebene noch Platz dar Darstellung von Kanten ist, die

einen Bogen zu Knoten auf der gleichen oder höheren Ebene machen. In allen anderen Fällen wird $maxdepth + 1$ verwendet.

Das Zeichnen der Knoten beginnt an der Wurzel, die in der Mitte von Knotenebene 0 dargestellt wird. Die folgenden Ebenen werden dann in der Breite in Darstellungsabschnitte für die Nachfolgeknoten der Wurzel aufgeteilt, wobei die Aufteilung im Verhältnis zur Visualisierungsbreite der dort beginnenden Subgraphen erfolgt. Dieser Prozess wiederholt sich nun für jeden Knoten als Wurzel seines Subgraphen in der Mitte seines Darstellungsabschnitts auf der Ebene seiner vorher bestimmten Darstellungstiefe. Abbildung 4.1 verbildlicht die Unterteilung der Zeichenfläche für einen Beispiel-Termgraphen.

Die Darstellung der Verbindungslinien erfolgt mittels mehrerer *Path*-Elemente für jede Verbindung in folgenden Schritten:

1. Es wird eine Bézierkurve dritten Grades vom aktuellen Knoten zur Mitte des Darstellungsabschnittes des entsprechenden Nachfolgeknotens gezeichnet (unabhängig ob der Nachfolgeknoten tatsächlich an dieser Stelle darstellt wird).

2. Falls der Nachfolgeknoten sich auf einer höheren oder der gleichen Ebene befindet wie der aktuelle Knoten, wird ein Bogen gezeichnet, von dem aus die Verbindungslinie auf eine höhere Ebene geführt werden kann.

3. Liegt der Nachfolgeknoten nicht vom aktuellen Knoten aus auf der nächsttieferen Ebene, wird eine gerade Verbindungslinie zur Ebene des Nachfolgeknotens gezeichnet, falls dieser auf einer höheren oder der gleichen Ebene wie der aktuelle Knoten liegt. Liegt jener auf einer tieferen Ebene als der aktuelle Knoten, wird bis zur Ebene direkt über dem Nachfolgeknoten gezeichnet.

4. Liegt der Nachfolgeknoten mehr als eine Ebene tiefer als der aktuelle Knoten, so wird in diesem Schritt eine weitere Bézierkurve zur tatsächlichen Position des Nachfolgeknotens gezeichnet.

5. Falls der Nachfolgeknoten auf der gleichen oder einer höheren Ebene als der aktuelle Knoten liegt, wird ein weiterer Bogen zur tatsächlichen Position des Nachfolgeknotens gezeichnet.

In Abbildung 4.1 sind die einzelnen Schritte dieser Prozedur in den Verbindungslinien farblich markiert.

Damit ist die Visualisierung eines Termgraphen als tatsächlicher Graph abgeschlossen, sodass im nächsten Abschnitt die Visualisierung eines Termgraphen als Baum betrachtet werden kann.

4.2.2. Termgraphen als Bäume

Ebenso wie die Visualisierung als Graph erfolgt die Visualisierung als Baum in drei Schritten:

Auslesen der Daten aus dem XML-Zwischenformat

Das Vorgehen ist vollständig identisch mit dem in Abschnitt 4.2.1 beschriebenen Vorgehen zum Auslesen des Zwischenformats.

Erzeugung des Formats zur Visualisierung

Anders als in Abschnitt 4.2.1 kann bei der Visualisierung von Bäumen eine verlinkte Datenstruktur verwendet werden, da letztlich nur Bäume und keine Graphen visualisiert werden. In der verwendeten

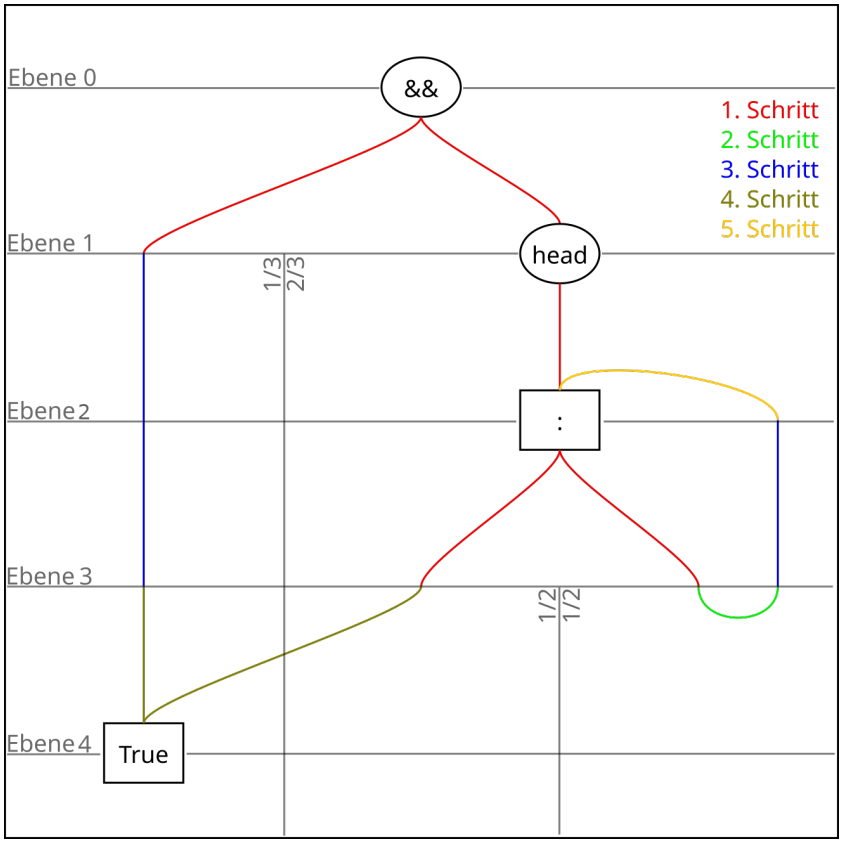


Abbildung 4.1. Verdeutlichung der Unterteilung der Zeichenfläche zur Termgraph-Visualisierung sowie der Schritte der Zeichenprozedur für die Verbindungslinien.

Datenstruktur wird ein Graph/Baum durch seinen Wurzelknoten repräsentiert, sodass sich folgende Datenstruktur ergibt:

```
data DNode = DNode {
        nodeType     :: NodeType,
        nodeID       :: NodeID,
        label        :: String,
        children     :: [DNode],
        strokeColour :: String,
        fillColour   :: String,
        width        :: Int
}
```

Anders als beim `Node`-Datentypen besteht die Liste der Kind-Knoten nun aus anderen `DNode`-Objekten, was der verlinkten Natur des Datentypen entspricht. Ebenso wie in Abschnitt 4.2.1 wird die Farbe der Füllung eines Knoten daraus abgeleitet, ob er aktiv ist bzw. ein Ergebnis repräsentiert. Die Färbung der Umrandung wird nun aber verwendet, um Sharing zu visualisieren. Da Knoten mit Sharing bei dieser Art der Darstellung mehrfach gezeichnet werden, ist es wichtig, dass weiterhin erkennbar ist, dass es sich bei solchen Knoten nicht um unterschiedliche, sondern um den selben

Knoten handelt. Aus diesem Grund werden alle visualisierten Instanzen eines Knotens mit der gleichen Farbe umrandet.

Um nun die Farbe der Umrandung für jeden Knoten bestimmen zu können, wird im Vorhinein eine partielle Abbildung *cmap* von den Knoten-IDs eines Graphen auf Farben berechnet. Dies geschieht, indem geprüft wird, ob eine Knoten-ID als Nachfolger mehrerer Knoten des bestehenden `Graph`-Objektes eingetragen ist. Wird damit festgestellt, dass ein Knoten mehrere Vorgängerknoten hat, wird seine ID zusammen mit einer Farbe aus einer vordefinierten Liste von elf gut unterscheidbaren Farben zu *cmap* hinzugefügt. Knoten die am Ende auf keine Farbe abbilden, also nicht mehr als einen Vorgänger haben, werden schwarz umrandet.
Existieren mehr Knoten mit Sharing in einem Termgraphen als Farben in der erwähnten Farbliste sind, so müssen Farben mehrfach für unterschiedliche Knoten verwendet werden. Da die Alternative einer zufälligen Generierung von Farben zu schlecht unterscheidbaren Farben führen kann und es nur selten vorkommt, dass in einem Termgraphen mehr als elf Knoten mit Sharing vorkommen, wird die erwähnte Limitation in Kauf genommen. Sollten regelmäßig sehr große Graphen mit vielen Sharing-Knoten visualisiert werden, kann die vordefinierte Farbliste außerdem leicht erweitert werden.

Da bei der Visualisierung von reinen Bäumen die zu visualisierende Breite eines Teilbaumes nicht von bereits gezeichneten bzw. besuchten anderen Knoten abhängig ist, kann die Breite eines Baumes direkt in den ihn repräsentierenden Wurzelknoten eingetragen werden. Sei nun $TG = (V, E, r)$ wieder ein Termgraph, *maxDepth* die zur Vermeidung unendlicher Bäume bei Termgraphen mit Kreisen notwendige maximale Darstellungstiefe und $succs : V \to \mathcal{P}(V)$ eine Abbildung, die ähnlich der Abbildung E Knoten auf ihre Nachfolger abbilde, mit der Ausnahme, dass Knoten auf Mengen abgebildet werden (Sharing muss an dieser Stelle nicht mehr berücksichtigt werden) sowie dass Knoten mit Tiefe=*maxdepth* auf die leere Menge abbilden. Dann lässt sich die Breite eines Teilbaumes durch folgende Funktion berechnen:

$$treeBreadth(v) = max(1, \sum_{x \in succs(v)} treeBreadth(x))$$

Visualisierung der Bäume

Die Visualisierung der Termgraphen als Bäume aus dem letzten Format verläuft nun in Teilen analog zum in Abschnitt 4.2.1 beschriebenen Verfahren: Sei r die Wurzel des im vorigen Schritt generierten Baumes und *treeheight* dessen Höhe, definiert als die Länge des längsten Pfades von einem beliebigen Blatt des Baumes zur Wurzel. Dann werden die minimale Höhe h_{min} und minimale Breite w_{min} der Zeichenfläche zur Darstellung des Baumes ähnlich wie in Abschnitt 4.2.1 berechnet:

$$w_{min} = treeBreadth(r) * 140$$

$$h_{min} = treeheight * 80$$

Genau wie bei der Visualisierung von Graphen wird $max(w_{min}, h_{min})$ als Höhen- und Breitendimension für die Zeichenfläche verwendet. Zum Zeichnen des Baumes wird nun die Höhe der Zeichenfläche in $h + 1$ Ebenen unterteilt. Eine eventuelle zusätzliche Ebene für Kanten, die einen Kreis formen, ist im Fall der Visualisierung von Bäumen offensichtlich nicht nötig.

Das Zeichnen der Knoten verläuft nun ebenfalls sehr Ähnlich wie in Abschnitt 4.2.1. Es unterscheidet sich dadurch, dass die Visualisierungsbreite für jeden zu zeichnenden Teilbaum bereits bekannt ist und Knoten immer auf der von ihrem Vorgänger ausgehend nächsttieferen Ebene gezeichnet werden, da kein Platz zur Darstellung von eingehenden Kanten von mehr als einem Vorgänger gelassen werden muss.

Die Darstellung der Kanten erfolgt nun ebenfalls nach einem ähnlichen, aber einfacheren Verfahren als in Abschnitt 4.2.1, denn es wird effektiv nur Schritt 1 aus dem dargestellten mehrschrittigen Verfahren ausgeführt.

4.3. Integration in das ICurry-Laufzeitsystem

Um den Termgraph-Visualisierer effektiv nutzen zu können, wurde er in das bestehende ICurry-Laufzeitsystem integriert. Der Interpreter des Laufzeitsystems nutzt bei Auswertung einer Anfrage einen `State`-Datentypen, der den Ausführungszustand nach jedem Auswertungsschritt repräsentiert:

```
-- module ICurry.Graph

type NodeID = Int

data State = State {
        program :: [IFunction],
        graph   :: Graph,
        tasks   :: [Task],
        results :: [NodeID],
        currResult :: Maybe NodeID
}
```

Naheliegenderweise ist für die Visualisierung die Information aus dem `graph`-Objekt relevant, jedoch befinden sich in der Liste `tasks` noch Informationen über den gerade aktiven Knoten sowie die aktuellen Choice-Alternativen, sowie in der Liste `results` Informationen über die Knoten, die berechnete Ergebnisse repräsentieren, liegen. Der Datentyp `Graph` ist im Interpreter dabei wie folgt definiert:

```
-- module ICurry.Graph

data Graph = Graph [(NodeID,Node)] NodeID

data Node = FuncNode   String   [NodeID]
| ConsNode   String   [NodeID]
| PartNode   String   PartCall [NodeID]
| ChoiceNode ChoiceID NodeID NodeID
| FreeNode
}
```

Relevant ist hier, dass der Datentyp in dieser Form keine Information über die aktuelle Wurzel des Termgraphen enthält, sodass er um ein Attribut vom Typ `NodeID` zur Speicherung der aktuellen Wurzel ergänzt wurde. Dieses ist initial immer 1 und wird bei einer Ersetzung des Wurzelknotens entsprechend angepasst.

Um nun eine Visualisierung der Termgraphen der einzelnen `States` zu ermöglichen, müssen die relevanten Informationen nun lediglich in das in Abschnitt 4.1 beschriebene Zwischenformat übertragen und an den Graphvisualisierer weitergegeben werden.
Während eine Option zur Generierung des Zwischenformates und der Ausgabe der generierten XML-Daten in einer Textdatei in das Laufzeitsystem implementiert wurde, läuft die Übergabe an den Visualisierer direkter, um etwas Rechenaufwand zur Datenkonvertierung einzusparen. Konkret

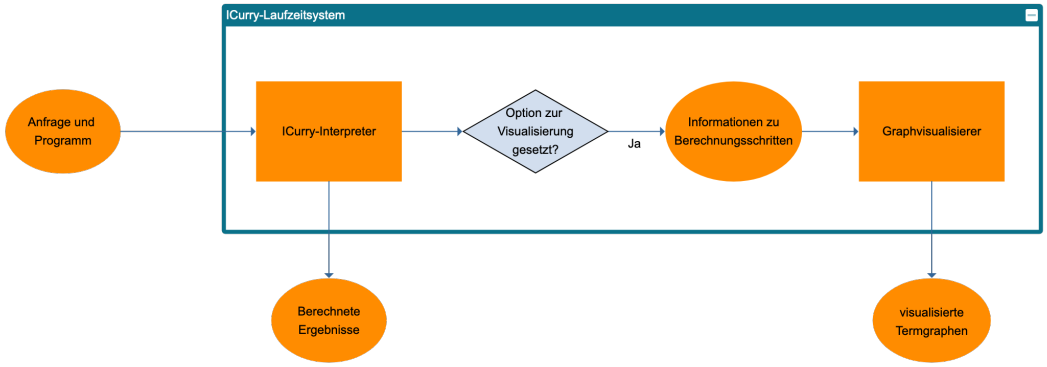


Abbildung 4.2. Vereinfachte Darstellung der Integration des Visualisierers in das ICurry-Laufzeitsystem.

werden an den Visualisierer reduzierte `State`-Objekte übergeben, die folgendermaßen definiert sind, wobei die bekannten Datentypen aus `ICurry.Graph` importiert wurden:

```
-- module Termgraph.XmlGraph

data State = State  {
        graph       :: Graph,
        activeNode  :: NodeID,
        results     :: [NodeID],
        fingerprint :: FingerPrint
}
```

Hier sind nun nur noch die Informationen vorhanden, die zur Erstellung in der in Abschnitt 4.2.1 eingeführten `graph`-Objekte notwendig sind.

Somit bestehen im ICurry-Laufzeitsystem nun neben der Option zur Ausgabe von Termgraphen im XML-Zwischenformat auch Optionen zur direkten Ausgaben von visualisierten Termgraphen in Form von tatsächlichen Graphen und in Form von Bäumen. Abbildung 4.2 stellt vereinfacht dar, wie der Visualisierer in das ICurry-System integriert wurde.

Für eine einfachere Generierung visualisierter Berechnungen wurde aber weiterhin eine Webanwendung entwickelt, deren Implementierung im nächsten Kapitel betrachtet wird.

Kapitel 5

Implementierung der Webanwendung

Für die einfache Implementierung von Webanwendungen stehen eine große Anzahl von Frameworks für die verschiedensten Programmiersprachen zur Verfügung. Mit Spicey [HK10] existiert sogar ein Curry-Framework für webbasierte Anwendungen mit Fokus auf Datenmanipulation in relationalen Datenbanksystemen. Da allerdings eine Datenbank für die zu implementierende Webanwendung nicht notwendig war, wurde das Python-Framework *Flask*[1] wegen seiner Minimalität und einfachen Benutzbarkeit gewählt.

5.1. Flask

Flask verwendet das in Python-Webanwendungen zum Standard gewordene WSGI (Web Server Gateway Interface) [Eby10] zu Kommunikation mit einem WSGI-Webserver, verfügt aber auch über einen integrierten Webserver für Entwicklungszwecke. Weiterhin nutzt es die Templating-Engine *Jinja*[2] zum Rendern von Webseiten-Templates.

Wie in den meisten Frameworks für webbasierte Applikationen werden in einer Flask-Applikation grundsätzlich Methoden zur Bearbeitung von Anfragen an unterschiedliche Pfade bzw. mit Unterschiedlichen HTTP-Methoden definiert, die am Ende eine Antwort auf die Anfrage zurückliefern. Diese Antworten können jeglicher Form sein, bei einer normalen GET-Anfrage für eine Webseite ist dies jedoch häufig ein gerendertes Template. Die wichtigste Funktionalität von mit Jinja gerenderten Templates ist hierbei die Möglichkeit der Integration von Ausdrücken und Statements für den Kontrollfluss in Seiten-Templates, wodurch z.B. mit dem folgendem Code alle Elemente einer an den Template-Renderer übergebenen Liste `list` auf einer Webseite angezeigt werden können:

```
<ul>
{% for item in list %}
        <li>{{item}}</li>
{% endfor %}
</ul>
```

[1] https://flask.palletsprojects.com/en/2.0.x/

[2] https://jinja.palletsprojects.com/en/3.0.x/

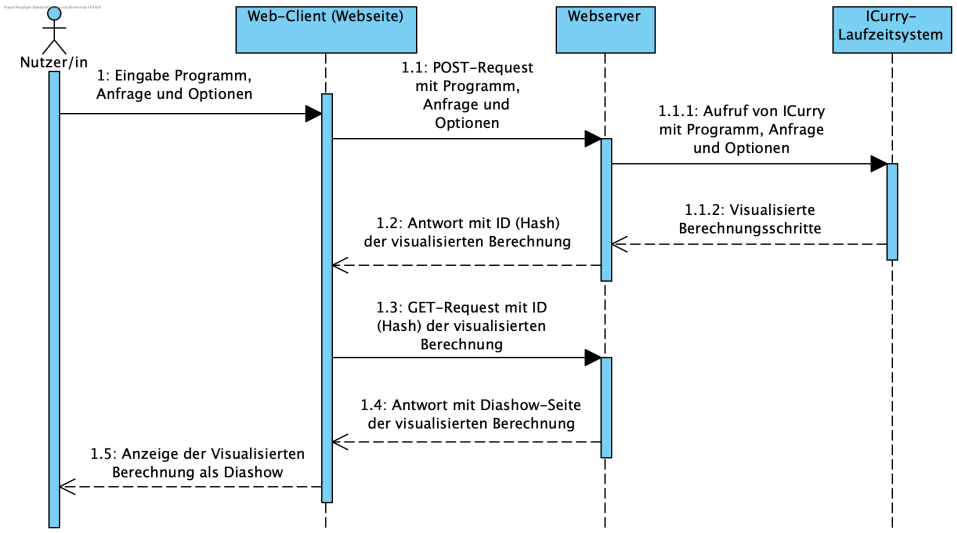


Abbildung 5.1. Ein Sequenzdiagramm, welches den Ablauf der Visualisierung einer Berechnung darstellt, wenn die entsprechende Berechnung nicht bereits im Cache vorliegt.

5.2. Funktionsweise der Webanwendung

Nach der kurzen Einführung in das Framework können wir uns in diesem Abschnitt der Webanwendung selbst zuwenden.

Der Einstiegspunkt in die Webanwendung besteht aus einem Formular, in welches das auszuführende Curry-Programm, der auszuwertende Hauptausdruck sowie Einstellungen zur Darstellung der Termgraphen als Graphen oder Bäume eingegeben werden können. Werden die Formulardaten abgeschickt, geschieht dies über ein XMLHttpRequest an den Webserver, sodass dann im Hintergrund die Berechnung der Termgraphen stattfinden kann, während auf der Seite des Formulars eine Information über die laufende Berechnung angezeigt wird.

Wird eine Anfrage zur Visualisierung der Termgraphen eines Programms P mit Anfrage A und Visualisierungseinstellungen S an den Webserver gestellt, so wird noch folgendem Verfahren IDs für visualisierte Termgraphen und das eingegebene Programm erstellt:

Programm:
$$"\mathsf{p}\_" ++ md5\_hash(P + A)$$

Grafiken für Graphen:
$$"\mathsf{p}\_" ++ md5\_hash(P + A)$$

Grafiken für Bäume mit maximaler Tiefe t:
$$"\mathsf{g}" ++ t ++ "\_" ++ md5\_hash(P + A)$$

Das übergebene Programm wird dann unter der generierten ID in einem Cache gespeichert und es werden die geforderten Termgraphen mit dem ICurry-Laufzeitsystem generiert und ebenfalls unter der berechneten ID im Cache gespeichert, sodass bei einer wiederholten Anfrage mit den selben Parametern keine neue (und zeitaufwändige) Visualisierung der Termgraphen stattfinden muss.

Im Anschluss wird die ID der generierten Graphiken an den Client zurückgeschickt, welcher dann auf eine Seite zur Anzeige der generierten Graphiken weiterleitet. Hierzu wird vom Server dann ein HTML-Dokument generiert, in dem die generierten Grafiken in einer Diashow dargestellt werden.

Die Aufteilung dieses Prozesses in zwei Anfrage-Antwort-Schritte geschieht, da die Generierung der Termgraphen u.U. mehr als zehn Sekunden in Anspruchen nehmen kann und ein Webbrowser bei einer solchen längeren Wartezeit auf die nächste anzuzeigende Seite unter Umständen mit einem Timeout reagieren könnte.

Auf der Diashow-Seite besteht dann neben der Betrachtung der Termgraphen in der Diashow die Möglichkeit, die Anzahl der nebeneinander dargestellten Termgraphen anzupassen, und sich den Quelltext der visualisierten Berechnung in einem Pop-up-Fenster anzeigen zu lassen. Dieser wird über seine vorher generierte ID aus dem Cache geladen. Weiterhin werden bei der Darstellung von Termgraphen in Baumform Knoten aller Instanzen eines Knotens mit Sharing markiert, sobald mit der Maus über eine der gezeichneten Instanzen gefahren wird. Ein Screenshot der Diashow-Seite mit Markierung der Bedienelemente findet sich im Anhang bei Abbildung 6.2. Abbildung 5.1 zeigt den Ablauf einer Visualisierungsanfrage an die Webanwendung für eine noch nicht im Cache vorhandene Anfrage in Form eines Sequenzdiagramms.

Außerdem verfügt die Webanwendung über die Möglichkeit, mit dem Einstiegsformular bestehende im SVG-Format visualisierte Termgraphen in der Diashow darzustellen. Dies ist in Fällen praktisch, in denen über den interaktiven Modus im ICurry-Laufzeitsystem eine Berechnung nur teilweise ausgeführt wurde (z.B. für nicht terminierende Berechnungen) und die Ergebnisse danach über die Diashow der Webanwendung angezeigt werden sollen.

Damit sind die Teile zur Implementierung dieser Arbeit abgeschlossen und wir haben gelernt, wie das Zwischenformat zwischen Laufzeitsystem und Graphvisualisierer aufgebaut ist, wie die Visualisierungsprozess für Termgraphen funktioniert und wie die Webanwendung strukturiert ist. Hiermit können wir uns im nächsten Abschnitt dann dem Fazit und dem Ausblick auf mögliche weitere Arbeit in diesem Bereich widmen.

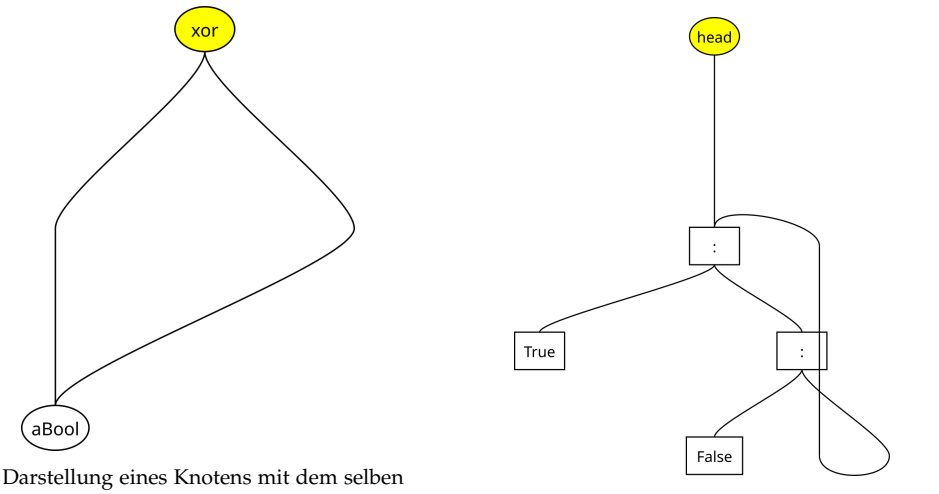
Kapitel 6

Fazit und Ausblick

6.1. Fazit

Mit dem entwickelten Visualisierer lassen sich die Berechnungsschritte von Berechungen in Curry übersichtlich darstellen und mittels der Webanwendung können komplette Curry-Berechnungen anwenderfreundlich über den Browser erstellt und betrachtet werden. Weiterhin ermöglicht die Ausgabe im SVG-Format eine einfache Bearbeitung der visualisierten Graphen mittels Zeichenprogrammen für Vektorgrafiken wie *Inkscape*[1]. Bereits im Rahmen dieser Arbeit konnte der Visualisierer für die direkte Erstellung vieler Abbildungen bzw. für die Erzeugung der Grundlage einiger Abbildungen, die im nächsten Schritt bearbeitet wurden, bereits erfolgreich eingesetzt werden.

6.2. Reflexion und Ausblick



(a) Darstellung eines Knotens mit dem selben Knoten als mehrfachem Nachfolger.

(b) Darstellung eines Graphen mit rekursivem *let*.

Abbildung 6.1. Beispiele für nicht ideal visualisierte Termgraphen.

Auch wenn der generelle Rahmen des entwickelten Visualisierers gut funktioniert, so ist doch die Visualisierung von Graphen generell ein sehr komplexes Thema. Aus diesem Grund produziert der entwickelte Visualisierer ein einigen Fällen Ergebnisse, die nicht ideal sind:

Einerseits werden Knoten, die mittels Sharing Nachfolger mehrerer Knoten sind, immer unter die entsprechende Argumentposition des ersten gezeichneten Elternknoten gezeichnet, was z.B. bei

[1] `https://inkscape.org/`

Knoten, die den selben Knoten mehrfach als Nachfolger haben, zu unintuitiven Darstellungen führen kann, wie in 6.1a zu sehen ist. Eine gesonderte Behandlung von Fällen, in welchen der betroffene Kindknoten in die Mitte der Argumentpositionen gezeichnet wird, wie es z.B. in Abbildung 2.2 (links) der Fall ist, wäre eine mögliche Abhilfe.

Andererseits ist bei Kanten, die aufgrund von Kreisen im Termgraphen zu einem höher liegenden Knoten verlaufen, nicht sichergestellt, dass diese nicht durch andere Knoten verlaufen, wie in Abbildung 6.1b erkennbar ist. Für diese Fälle wäre eine Erweiterung des Visualisierungsalgorithmus denkbar, mit der bei der Berechnung der x-Koordinate für den senkrecht nach oben verlaufenden Abschnitt der Kante die Positionen aller Knoten zwischen dem aktuellen und dem Nachfolgerknoten berücksichtigt und vermieden werden können.

Aber auch noch weitere Erweiterungen für das Visualisierungssystem sind denkbar, wie z.B. die Ausgabe der visualisierten Graphen in einem Format wie *GraphML* für Graph-Editoren wie *yEd*[2], damit visualisierte Graphen noch einfacher im Nachhinein bearbeitet werden können, oder die Nutzung von Animationen in der Webanwendung, mit welchen mittels sich bewegender Knoten o.Ä. die Übergänge von einem Termgraphen zum nächsten noch anschaulicher dargestellt werden können. Letzteres wäre jedoch sehr aufwändig bzw. nur durch bereits bestehender Software mit angemessenem Aufwand umsetzbar.

[2] https://www.yworks.com/products/yed

Literatur

[AEH00] Sergio Antoy, Rachid Echahed und Michael Hanus. „‚A needed narrowing strategy‘“. In: *Journal of the ACM (JACM)* 47.4 (2000), S. 776–822.

[AH10] S. Antoy und M. Hanus. „‚Functional Logic Programming‘“. In: *Communications of the ACM* 53.4 (2010), S. 74–85.

[Ant+20] S. Antoy u. a. „‚ICurry‘“. In: *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, 2020, S. 286–307. DOI: 10.1007/978-3-030-46714-2_18.

[Bra+08] Tim Bray u. a. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. https://www.w3.org/TR/2008/REC-xml-20081126/. W3C, Nov. 2008.

[Eby10] P.J. Eby. Python Web Server Gateway Interface v1.0.1. PEP 3333. 2010. URL: https://peps.python.org/pep-3333/.

[Han13] M. Hanus. „‚Functional Logic Programming: From Theory to Curry‘“. In: *Programming Logics: Essays in Memory of Harald Ganzinger*. Berlin, Heidelberg: Springer, 2013, S. 123–168. DOI: 10.1007/978-3-642-37651-1_6.

[HK10] M. Hanus und S. Koschnicke. „‚An ER-based Framework for Declarative Web Programming‘“. In: *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*. Springer LNCS 5937, 2010, S. 201–216.

[McC+11] Cameron McCormack u. a. Scalable Vector Graphics (SVG) 1.1 (Second Edition). W3C Recommendation. https://www.w3.org/TR/2011/REC-SVG11-20110816/. W3C, Aug. 2011.

Installation und Nutzung

Voraussetzung

Eine funktionierende Installation des PAKCS[3] mit dem Curry package manager *cypm* im Pfad muss vorhanden sein.

Zur Nutzung der Webanwendung ist außerdem Python 3.6 oder eine aktuellere Version erforderlich.

Installation

Die beschriebene Implementierung befindet sich in zwei Ordnern eines Git-Repositories:

- `icurry`: In diesem Verzeichnis befindet sich das ICurry-Laufzeitsystem, in die der Graphvisualisierer integriert wurde. Dabei liegt im Verzeichnis `icurry/src/Termgraph` die implementierung des Visualisierers an sich und im Verzeichnis `icurry/src/ICurry` das modifizierte ICurry-Laufzeitsystem.
- `webapp`: In diesem Verzeichnis befindet sich die Webanwendung.

Um nun das ICurry-Laufzeitsystem mit integriertem Graphvisualisierer zu kompilieren muss in das `icurry`-Verzeichnis des Repositories gewechselt werden. Dort kann das Laufzeitsystem nun durch das Kommando `cypm install` kompiliert werden. Die ausführbare Binärdatei befindet sich anschließend in `/.cpm/bin`.

Für die Webanwendung muss noch Flask über den python-packetmanager mit folgendem Kommando installiert werden: `pip3 install Flask`

Nutzung

Zur Nutzung des in das ICurry-Laufzeitsystem integrierten Visualisierers wurden zur icurry-executable folgende Kommandos hinzugefügt:

- `--textgraph[=<dateiname>]`: Generiere und speichere die Termgraphen für jeden Berechnungsschritt im textuellen Zwischenformat.
- `--graphsvg[=<dateiname>]`: Generiere und speichere die als Graph visualisierten Termgraphen für jeden Berechnungsschritt.
- `--treesvg[=<dateiname>]`: Generiere und speichere die als Baum visualisierten Termgraphen für jeden Berechnungsschritt
- `--shownodeids`: Zeige in den visualisierten Termgraphen die IDs der Knoten in ihrem Label.
- `--maxdepth=<n>`: Setze die maximale Tiefe für die Visualisierung von Termgraphen als Baum. Der Standardwert ist 10.

Die Graphvisualisierung lässt sich hierbei auch problemlos mit der Option `-i` bzw. `--interactive` kombinieren, um nur einen bestimmten Teil einer Berechnung zu visualisieren.

[3] Verfügbar unter: `https://www.informatik.uni-kiel.de/~pakcs/`

Zur Nutzung der Webanwendung muss sich die `icurry`-executable des modifizierten ICurry-Laugzeitsystems in Pfad befinden. Dann kann einfach das `icurry_web`-Skript im `webapp`-Verzeichnis ausgeführten: `python3 icurry_web.py`. Die Webanwendung ist dann unter `localhost:5000` erreichbar. Es sei an dieser Stelle jedoch erwähnt, dass der integrierte Flask-Webserver, der auf diese Weise ausgeführt wird, nur für Entwicklungs- und Testzwecke ausgelegt ist und nicht auf einem frei zugänglichen Server ausgeführt werden sollte. Hierfür sollte ein gängiger WSGI-Server wie *Gunicorn*[4] oder *mod_wsgi*[5] für den Apache-Webserver benutzt werden.

[4] `https://gunicorn.org/`

[5] `https://pypi.org/project/mod-wsgi/`

Screenshots

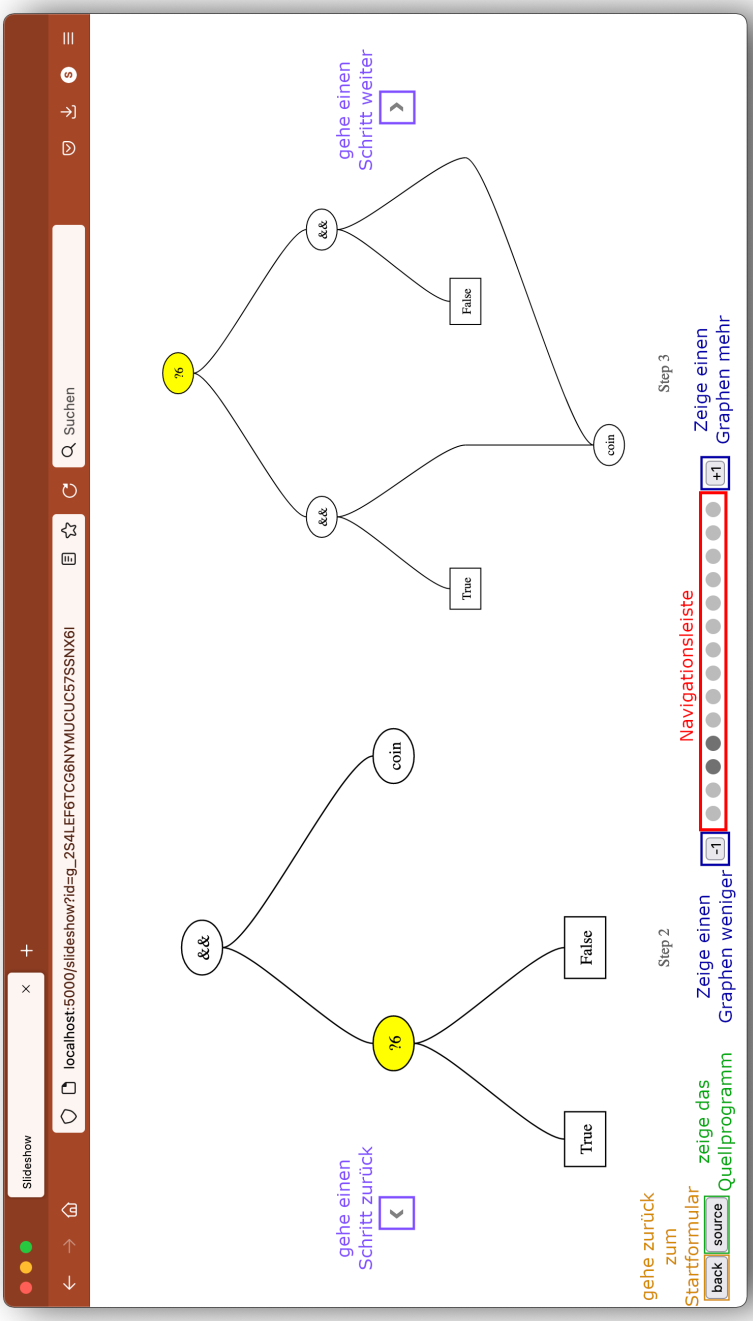


Abbildung 2. Screenshot der Diashow in der Webanwendung mit Beschreibung der Bedienelemente.