Ein Browser für Alice

14. Mai 2003

Bernadette Blum
blum@ps.uni-sb.de

Marvin Schiller
schiller@ps.uni-sb.de

Betreuer:

Thorsten Brunklaus
brunklaus@ps.uni-sb.de

Andreas Rossberg
rossberg@ps.uni-sb.de

Leitung:

Prof. Dr. Gert Smolka
smolka@ps.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes
Im Stadtwald
66123 Saarbrücken

Zusammenfassung

Dieser Bericht zeigt Design und Implementierung eines interaktiven Browser-Tools für die funktionale Programmiersprache Alice. Da Alice-Werte nicht selbstbeschreibend sind, muss der Browser jeweils explizite Typinformationen zu denjenigen Werten erhalten, die dargestellt werden sollen. Um Werte abstrakter Typen in eine entsprechende Darstellung transformieren zu können, wird weiterhin deren Registrierung beim Browser erforderlich. Der Browser ist in Alice selbst implementiert und verwendet die Gtk-Bibliothek zur Erzeugung der graphischen Benutzerschnittstelle. Unser Design knüpft weitgehend an Thorsten Brunklaus' "Oz Inspektor" [1] an.

1 Einführung

Die graphische Aufbereitung von Programmen und Datenstrukturen ermöglicht Programmierern das Debuggen komplexer Programme. Dazu werden Algorithmen entwickelt, die man als "Pretty Printer" bezeichnet. Ziel bei der Entwicklung von Pretty Printern ist es, die Lesbarkeit eines Dokuments (meist Programme und mathematische Ausdrücke) durch Einrückungen, gezielte Zeilenumbrüche und visuelle Attribute zu verbessern. Dieser Bericht stellt Design und Implementierung eines solchen "Pretty Printers" für die Programmiersprache Alice vor.

Die funktionale Programmiersprache Alice ermöglicht nebenläufige Programmierung und die Erzeugung zyklischer Datenstrukturen. Dies erlaubt es dem Programmierer, Berechnungsstränge (sog. Threads) willkürlich zu starten oder zu beenden, Werte erst bei Bedarf auswerten zu lassen (sog. Laziness), oder Platzhalter für noch nicht bekannte Berechnungsergebnisse einzusetzen (sog. Futures). Die daraus resultierende Komplexität erfordert eine Darstellung, die diesen Konzepten in Übersichtlichkeit und Flexibilität gerecht wird.

Anders als beispielsweise in der Programmiersprache Oz sind Typen in Alice nicht selbstbeschreibend. Dadurch muss zur graphischen Darstellung eines Alice-Wertes durch den Browser nicht nur der Wert selbst, sondern auch strukturelle Information zu diesem Wert übergeben werden. Diese lässt sich durch sogenannte Typreflektion gewinnen. Dieser Bericht zeigt, wie im Browser die strukturelle Information bei der Darstellung der dazugehörigen Werte verwendet wird und welche Mechanismen nötig sind, um Werte beliebiger abstrakter Typen darstellen zu können.

2 Zentrale Ideen

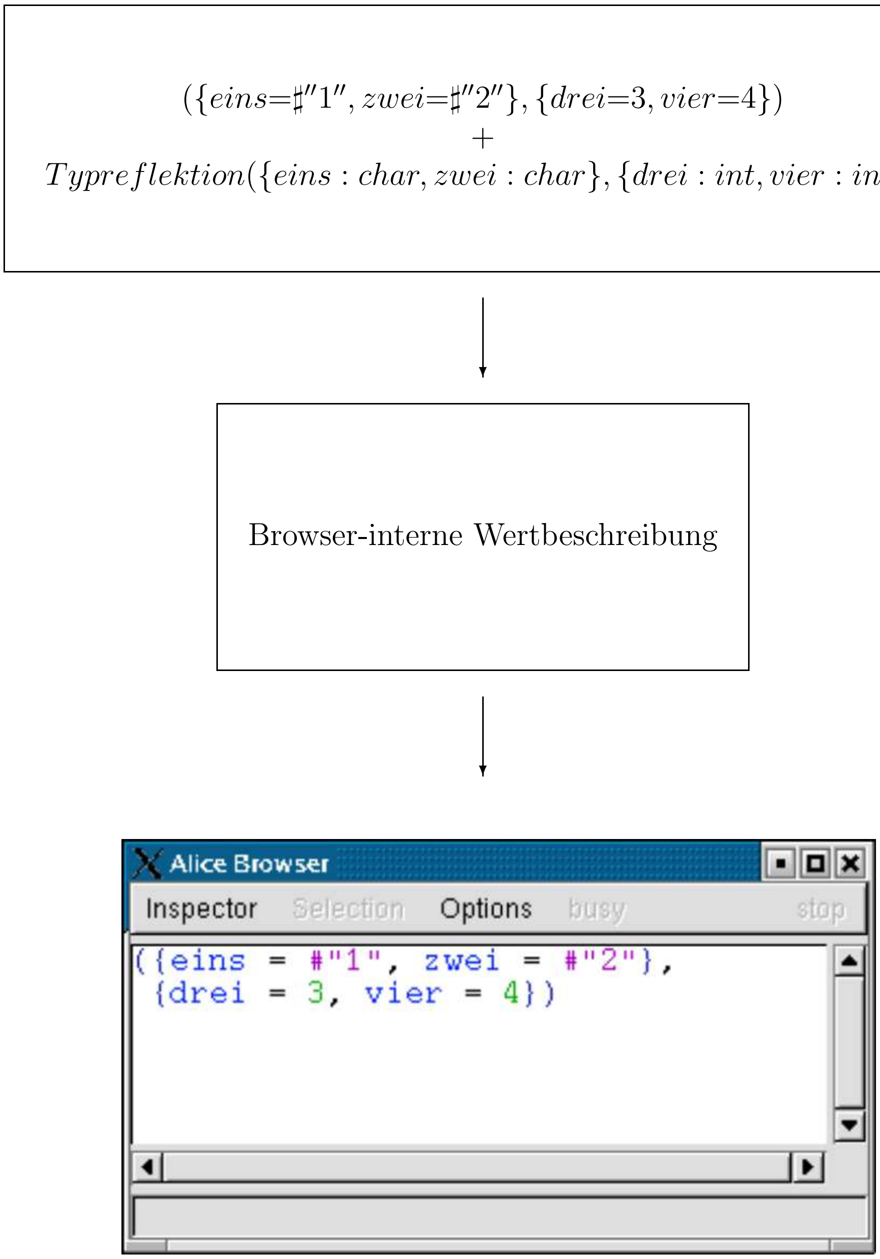

Abbildung 2.1: Vom Wert zur Browserdarstellung

Der Browser soll in der Lage sein, Alice-Werte übersichtlich auf einer graphischen Benutzeroberfläche darzustellen. Dies wird erreicht, indem die Werte in eine interne Beschreibung transformiert werden, aus der die graphische Repräsentation erzeugt werden kann. Abbildung 2.1 verdeutlicht diesen Vorgang. Die Transformation in die interne Beschreibung wird durch die Typinformation der zu inspizierenden Werte gesteuert.

2.1 Typregistrierung

Die Typen von Alice-Werten können strukturelle oder abstrakte Typen sein. Um die Werte in eine entsprechende Darstellung transformieren zu können, benötigt der Browser explizite strukturelle Information zu diesen Werten.

Gleichzeitig benötigt der Browser Anweisungen, wie ein Wert eines bestimmten Typs in die interne Repräsentation und letztendlich die graphische Darstellung umgewandelt werden soll.

Während strukturelle Typen die benötigte Strukturinformation zu dieser Transformation liefern, müssen abstrakte Typen zusammen mit passenden Darstellungsanweisungen beim Browser registriert werden. Diese Darstellungsvorschriften werden in Form einer Prozedur übermittelt, welche die Umwandlung des Wertes in eine Wertbeschreibung spezifiziert.

2.2 Darstellungskriterien

Der Entwurf eines graphischen Tools erfordert die Überlegung, welche Kriterien bei der Wahl von Anordnungsvorschriften für die Darstellung der Datenstrukturen im Vordergrund stehen sollen. Es gibt Pretty Printer, die, wie auch der Algorithmus von Oppen [2], die Einhaltung einer vorgegebenen darstellbaren Seitenbreite sicherstellen, und durch gezielte Umbrüche und Einrückungen die Lesbarkeit wiederherstellen. Im Gegensatz dazu steht die Vorgehensweise, die Anordnung nicht an einer vorgegebenen Seitenbreite zu orientieren, sondern vielmehr die Größe des Darstellungsbrereichs an eine möglichst übersichtliche Darstellung anzupassen. Im Fall des Alice-Browsers wurde letztere Vorgehensweise gewählt, um trotz der Komplexität der Alice-Datenstrukturen eine klare Darstellung zu ermöglichen.

Zusammengesetzte Alice-Werte lassen sich in Teilwerte untergliedern, so dass sich ein solcher Wert in eine baumartige Darstellung zerlegen lässt. Hier deutet sich schon an, dass sich auch die interne Behandlung der Datenstrukturen an Baumstrukturen orientieren muss. Gleichzeitig soll der Browser bei der graphischen Darstellung auf die für Alice übliche Notation für Werte mit Klammern und Trennzeichen zurück greifen. Die Darstellung von baumartigen Werten durch den Browser orientiert sich dabei an einer Regel, wie sie u.a. Kennedy [3] für das Baumzeichnen aufstellt: Gleiche Teilbäume werden gleich dargestellt.

3 Entwurf

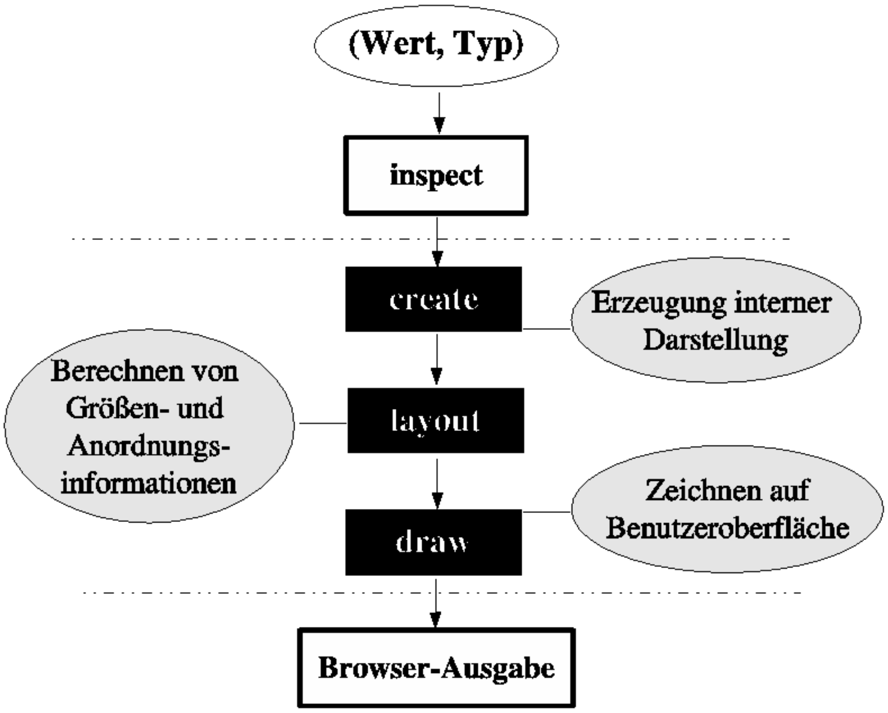

Abbildung 3.1: dreistufige Darstellungserzeugung

Abbildung 3.1 zeigt die dreistufige Erzeugung der graphischen Wertdarstellung.Vom Benutzer wird die inspect-Funktion des Browsers aufgerufen. Da Alice-Werte nicht selbstbeschreibend sind, benötigt die inspect-Funktion explizite Typinformation des zu inspizierenden Wertes. Mithilfe eines Typreflektors wird diese Information generiert und zusammen mit dem Wert an die inspect-Funktion übergeben. Um von dem Wert-Typ-Paar zur graphischen Darstellung des Wertes auf der Benutzeroberfläche zu gelangen, müssen drei Phasen durchlaufen werden.

Zuerst eird eine interne Beschreibung des Wertes erzeugt, die wichtige Informationen über dessen Struktur enthält. Diese Phase wird Create-Phase genannt.

In der nun folgenden Layout-Phase werden die Maße der Textrepräsentation des darzustellenden Wertes sowie all seiner Teilwerte berechnet. Dies dient zur Anordnung der Textrepräsentationen auf der Benutzeroberfläche.

Im letzten Schritt wird die graphische Repräsentation unter Nutzung der durch das Layout berechneten Anordnungsinformation auf die graphische Benutzeroberfläche gezeichnet.

3.1 Interne Darstellung

Für die interne Darstellung von Werten als Bäume benötigen wir einen Typen. Dieser wird in Abbildung 3.2 dargestellt.

$$
\begin{aligned}
doc := \ & simple \\
| \ & doc_1 \ \hat{} \ \ ... \ \ \hat{} \ doc_n \\
| \ & \#[doc_1, ..., doc_n]
\end{aligned}
$$

Abbildung 3.2: Definition von doc

Dabei repräsentieren “simples” nur atomare Werte, die als Strings dargestellt werden können. Folglich muss es sich in der Baumdarstellung bei “simples” stets um Blätter eines Baumes handeln.

Zusammengesetzte Werte bestehen im Gegensatz zu atomaren Werten aus einer Ineinanderschachtelung von Teilwerten. Zur Darstellung solcher nicht atomaren Werte stehen die Konkatenation ˆ und die Akkumulation # [ ] zur Verfügung. Während die Konkatenation stets horizontal anordnet (vergleichbar mit der sogenannten “hbox” bei manchen Pretty Printern), erfolgt bei der Akkumulation in der Aufbauphase noch keine Entscheidung, ob die Kindknoten eines Knotens horizontal oder vertikal dargestellt werden. Diese Entscheidung wird erst beim Layoutvorgang getroffen (nähere Erläuterung dazu siehe 4.4).

Der Wert (1, 2) wird in die folgende interne baumartige Darstellung transformiert:

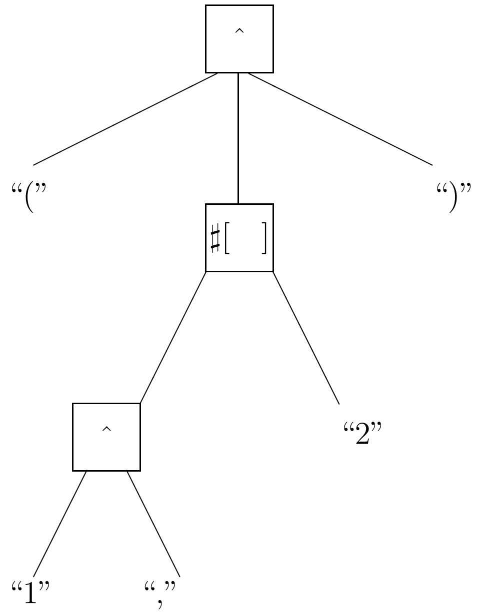

Abbildung 3.3: Beispiel für interne Darstellung

3.2 Typregistrierung

In der Create-Phase wandelt der Browser die zu inspizierenden Werte in eine Wertbeschreibung um. Dabei werden die Werte gegebenenfalls in Teilwerte zerlegt, die dann jeweils weiter in ihre Beschreibungen transformiert werden. Diese Zerlegung und Umwandlung wird durch die Typen der Werte gesteuert.

Im Gegensatz zu stukturellen Typen enthalten abstrakte Typen keine strukturelle Information zu den jeweiligen Werten. Deshalb werden für abstrakte Typen Anweisungen benötigt, die den Transformationsvorgang in die Wertbeschreibung genauer spezifizieren und damit letztendlich auch die graphische Darstellung auf der Benutzeroberfläche bestimmen.

Diese Anweisungen speichert der Browser für jeden abstrakten Typen in Form einer Prozedur. Jeder abstrakte Typ muss zusammen mit der entsprechenden Prozedur beim Browser registriert werden, damit Werte dieses Typs inspiziert werden können [siehe Abbildung 3.5].

Diese Prozeduren müssen folgende Anforderungen erfüllen: Sie sollen einen Wert mithilfe dessen Typinformation in eine Wertbeschreibung umwandeln, bei der die möglichen Teilwerte dieses Wertes unbearbeitet bleiben dürfen. Diese Teilwerte werden als Wert-Typ-Paare in die Wertbeschreibung eingebettet.

$$
\begin{array}{ll}
doc' := simple & \\
| & doc'_1 \;\hat{}\; \ldots \;\hat{}\; doc'_n \\
| & \#[doc'_1, \ldots, doc'_n] \\
| & value \quad * \quad type
\end{array}
$$

Abbildung 3.4: Erweiterung von doc zu doc'

Diese Wertbeschreibung muss also eine Variante beinhalten, die ein noch nicht abgearbeitetes Wert-Typ-Paar darstellt. Abbildung 3.4 zeigt die Erweiterung des Datentyps doc, die diese Anforderunge n erfüllt. Dieser Datentyp wird doc' genannt.

Definiert man sich also einen neuen Typen
$type\,'a\,doc\_creator = {}'a * Type.t \rightarrow doc'$,
so sind die oben beschriebenen Prozeduren zur Zerlegung abstrakter Typen vom Typ 'a doc_creator (siehe Abbildung 3.5).

$$
\begin{aligned}
createAbsType1 &: {'a}\, doc\_creator \\
createAbsType2 &: {'a}\, doc\_creator \\
usw.
\end{aligned}
$$

Abbildung 3.5: Prozeduren zur Zerlegung abstrakter Typen

Hierbei ist die durch Typreflektion gewonnene Typinformation zu einem Alice-Wert jeweils vom Typ Type.t.

Da diese in die Beschreibung der Datenstrukturen eingebetteten Wert- und Typ-Paare nur während der Erzeugung auftreten, kann doc' intuitiv als Zwischenstufe bei der Konstruktion der Beschreibung aufgefasst werden. Am Ende der Create-Phase ist der Wert also jeweils in ein Konstrukt des Typs doc zerlegt worden.

3.3 Transientenverwaltung

Transienten, die in Alice als Byneeds (bei Laziness), Promises oder Futures auftreten können und als Platzhalter für Werte dienen, werden in der Browserdarstellung als solche gekennzeichnet. Sobald sie an einen Wert gebunden werden, muss die Darstellung des Browsers aktualisiert werden. Deshalb wird für jeden Transienten ein Wächter benötigt, der diesen überwacht. Nimmt ein Transient, der in mehreren bereits dargestellten Datenstrukturen bzw. mehrmals in der gleichen Datenstruktur auftritt, einen konkreten Wert an, so löst sein Wächter einen Aktualisierungsmechanismus aus, der alle seine Auftreten durch diesen Wert ersetzt. Folglich muss jeder Wächter den Transienten kennen, den er überwacht, und zusätzlich alle Knoten, die diesen Transienten repräsentieren. Für jeden Transienten wird genau ein Wächter gestartet. Diesen Sachverhalt stellen die Abbildungen 3.6 und 3.7 dar.

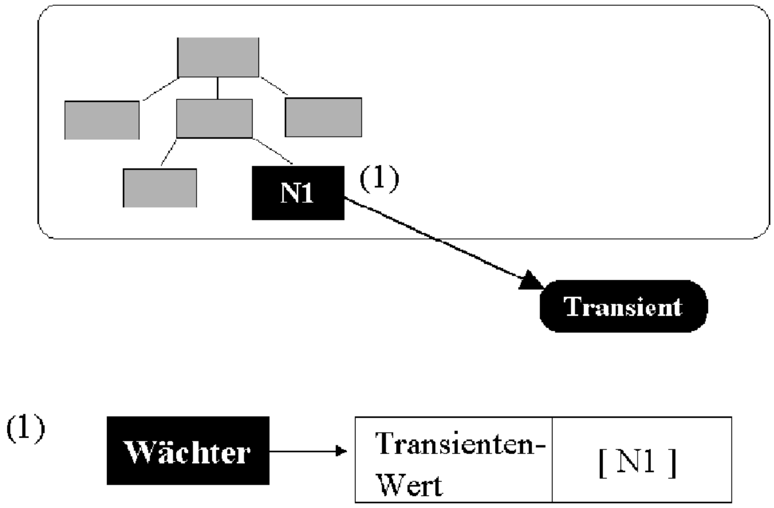

Abbildung 3.6: Transient mit Wächter

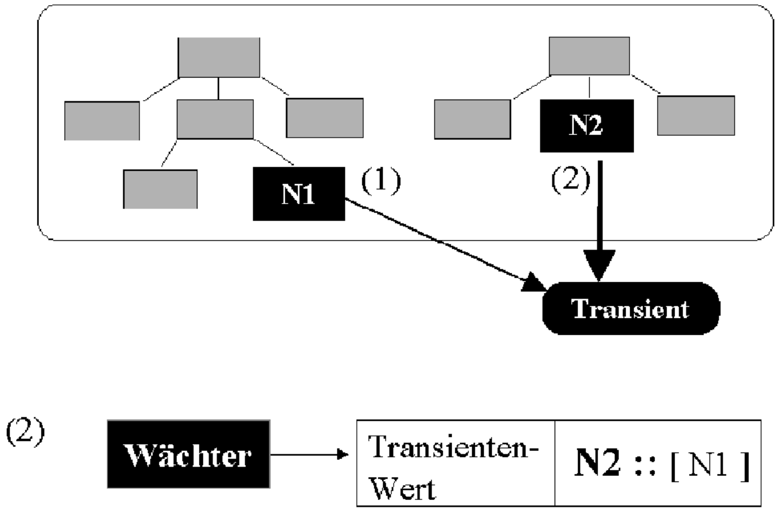

Abbildung 3.7: Mehrfachauftreten des Transienten

Wurde der Transient, wie in Abbildung 3.8 dargestellt, schließlich an einen Wert gebunden, hat der Wächter nach der Ingangsetzung des Aktualisierungsmechanismus' seine Funktion erfüllt und wird beendet.

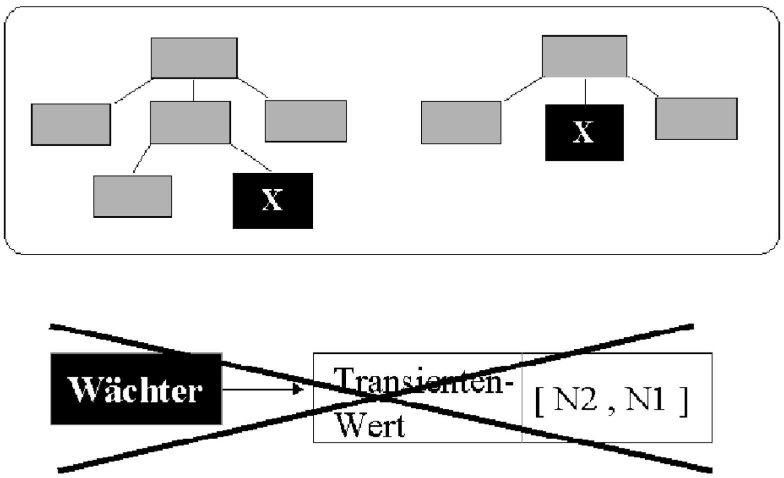

Abbildung 3.8: Bindung des Transienten

Die oben beschriebene Aktualisierung der Darstellung erfolgt - wie alle Darstellungsaktualisierungen - inkrementell. Dies bedeutet, dass - ausgehend von der bestehenden Darstellung - nur möglichst wenig Aufwand betrieben wird, um zu der neuen Darstellung zu gelangen. Nur diejenigen Teilbäume der internen Beschreibung werden neu konstruiert, gelayoutet und gezeichnet, für die dies unbedingt erforderlich ist.

3.4 Graphdarstellung

Der Browser verfügt über den Graphmodus, in dem er token-Gleichheit - physikalische Gleichheit von Objekten - innerhalb jeder dargestellten Datenstruktur erkennt. Diesen Modus kann der Benutzer beliebig ein- und ausschalten. Im Graphmodus ist die interne Darstellung von Werten nicht mehr baumartig [3.1], sondern graphartig aufgebaut. Die Graphdarstellung eignet sich besonders zur Verbesserung der Übersichtlichkeit bei der Darstellung zyklischer Datenstrukturen.

Betrachten wir den folgenden Datentypen:
datatype tree = T of tree * tree.
Die Deklaration
val rec t = T(t,t)
erzeugt einen unendlichen binären Baum. Die Baumdarstellung dieses Wertes enthält für jeden Knoten eine eigene Repräsentation des jeweilgen Wertes, ohne die zyklische Struktur des Baumes zu beachten. Dagegen wird bei der Erzeugung

der internen graphartigen Darstellung der Zyklus erkannt, so dass ein endlicher zyklischer Graph, wie in Abbildung 3.9 gezeigt, entsteht.

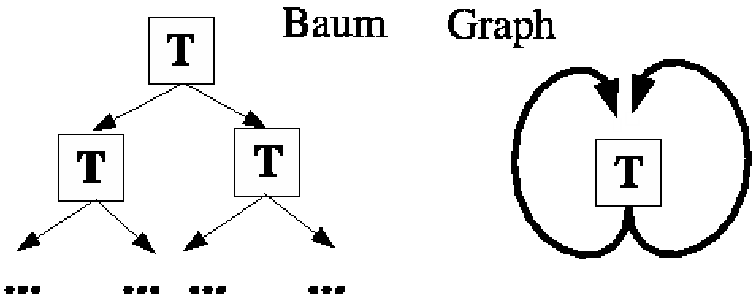

Abbildung 3.9: Baumdarstellung vs. Graphdarstellung

3.5 Filter

Der Browser muss in der Lage sein, den Umfang der Darstellung zu begrenzen. Dies erlaubt dem Benutzer, die Darstellung auf das Wesentliche einzuschränken. Weiterhin bietet ein Filter die Möglichkeit, die Darstellung zu großer oder zyklischer Datenstrukturen gemäß einer vom Benutzer gesetzten Grenze zu beschränken. Die Definition von doc' (also auch von doc) wird dazu um die Variante "limit" erweitert. erweitert (siehe Abbildung 3.10).

$$
\begin{aligned}
doc := \; & simple \\
\mid \; & doc_1 \;\hat{}\; ... \;\hat{}\; doc_n \\
\mid \; & \#[doc_1, ..., doc_n] \\
\mid \; & limit
\end{aligned}
$$

Abbildung 3.10: Erweiterung von doc' um "limit"

Bei der internen Beschreibung, die der Browser für die Datenstrukturen erzeugt, kann die Tiefe der Bäume beschränkt werden (sogenanntes Tiefenlimit), sowie auch die jeweilige Anzahl konkatenierter oder akkumulierter Kinder, die ein Knoten haben darf (sogenanntes Breitenlimit). Wird bei der Erzeugung der internen Darstellung das Tiefen- oder Breitenlimit überschritten, so wird an dieser Stelle die Konstruktion abgebrochen, und es wird ein "limit" angehängt. Folglich handelt es sich bei "limits" in der Baumdarstellung stets um Blätter. Den Abbruch des Aufbaus der internen Darstellung und das Setzen eines "limits" an die entsprechende Stelle bezeichnet man als Filtern.

In der Browserdarstellung wird das Überschreiten des Tiefen- oder Breitenlimits an der entsprechenden Stelle durch einen Pfeil gekennzeichnet.

4 Implementierung

4.1 Knotentypen

Für die Betrachtung des Designs genügt es, die aus den Werten erzeugte Beschreibung als doc darzustellen. In der Praxis möchte man diese doc-Knoten aber um zusätzliche Attribute erweitern, wie beispielsweise die von der Anordnungsberechnung erzeugte Information und die Gtk-Gruppierung, die vom Zeichenmechanismus benötigt werden.

Daher gibt es in der Implementierung zwei Datentypen, die die interne Beschreibung von Alice-Werten darstellen. Der erste - doc' - stellt die Benutzerschnittstelle für die Registrierung abstrakter Typen dar und ist daher auf das Wesentliche beschränkt. Jedes Konstrukt diese Datentyps speichert im Wesentlichen nur den Wert, den es repräsentiert, und seinen reflektierten Typen.

Da alle Werte letztendlich in Konstukte des Datentyps doc transformiert werden, besitzt doc in der Implementierung noch zusätzliche Felder, sogenannte "Zustände". Diese enthalten neben Anordnungsinformationen noch weitere Informationen, die für ds Zeichnen der Werte auf die Benuteroberfläche und für die Aktualiesierung bereits dargestellter Werte von Nutzen sind.

```
datatype doc' =

      SIMPLE of {desc : desc,
                 rep : string,
                 color : color_class }

    | CONCAT of {desc : desc,
                 kids : doc vector }

    | CONTAINER of {desc : desc,
                    kids : doc vector }

    | LIMIT of {desc : desc,
                sort : limit }

    | EMBEDDED of Reflect.value * Type.t
```

Abbildung 4.1: Datentyp doc'

Abbildung 4.1 zeigt die Implementierung des Datentyps doc'. Die "SIMPLES" stellen atomare Alice-Werte oder Trennzeichen dar. Diese können durch einen

String repräsentiert werden. Außerdem kann ihnen eine Farbe zugewiesen werden, in der sie auf die Benutzeroberfläche gezeichnet werden. Der abstrakte Typ desc dient dazu, Wert und Typ einzukapseln - das Speichern von Wert und Typ ist nötig, um bei der Änderung der Darstellung nochmal darauf zugreifen zu können. “CONCATS” und “CONTAINER” stellen Konkatenation und Akkumulation von mehreren doc'-Knoten dar. “LIMITS” markieren das Überschreiten der vom Benutzer einstellbaren Begrenzungen. Das Feld “sort” gibt dabei an, ob die maximal erlaubte Darstellungstiefe oder -breite überschritten wurde.

“EMBEDDEDS” stellen Wert-Typ-Paare dar, die von einer Prozedur nicht weiter bearbeitet werden konnten und deshalb in dieser Form an die nächste, dafür zuständige, Prozedur weitergereicht werden. Dabei wird allen Alice-Werten durch Reflektion der Typ Reflect.value gegeben. Das hat den Vorteil, dass der Datentyp doc' nicht polymorph als 'a doc' deklariert werden muss.

4.2 Typregistrierung und -zerlegung

Abstrakte Typen enthalten zwar keine strukturelle Information über Werte dieses Typs, jedoch existiert zu jedem abstrakten Typen ein sogenannter Pfad, der Informationen über de Struktur von Werten dieses Typs enthält. Beim Registrieren eines abstrakten Typs beim Browser wird der Pfad des Typs zusammen mit der entsprechenden create-Prozedur, die Werte dieses Typs auf genau einer Ebene zerlegen kann und somit ein doc'-Konstrukt erzeugt, in einer Map gespeichert. Dabei wird jeweils der Pfad als Schlüssel benutzt.

Der Browser enthält bereits die nötigen Mechanismen, um Werte struktureller Typen in entsprechende Beschreibungen zu transformieren. Diese Mechanismen arbeiten die zu inspizierenden Werte struktureller Typen solange ab, bis sie auf einen Teilwert stoßen, der einen abstrakten Typ hat. Zur weiteren Zerlegung extrahiert man nun den Pfad des abstrakten Typs und sucht in der oben beschriebenen Map nach der passenden create-Prozedur für Werte dieses Typs. Dieser wird dann der Wert zusammen mit seinem Typ zur weiteren Bearbeitung übergeben. Wird in der Map keine passende Prozedur gefunden, so wird ein Standardknoten erzeugt, der das Symbol _ als Stringrepräsentation enthält.

Die Implementierung des Browsers beinhaltet desweiteren schon create-Prozeduren für gebräuchliche abstrakte Typen wie beispielsweise Array, Vektor, Referenz, Int, String, Real, etc. Außerdem stehen dem create-Mechanismus die notwendigen Hilfsprozeduren zur Zerlegung struktureller Typen zur Verfügung. Der create-Mechanismus führt die Typzerlegung jeweils solange selbständig durch, bis er auf einen abstrakten Typen stößt. Für diesen muss dann in der Map nach der passenden create-Prozedur gesucht werden. Insgesamt wird ein Wert also solange in Teilwerte zerlegt und gegebenfalls zwischen verschiedenen Prozeduren hin- und hergereicht, bis er vollständig in die interne Beschreibung transformiert worden ist, d.h. bis aus dem Wert ein doc-Konstrukt entstanden ist, dass keine unbearbeiteten Wert-Typ-Paare mehr enthält.

4.3 Layout

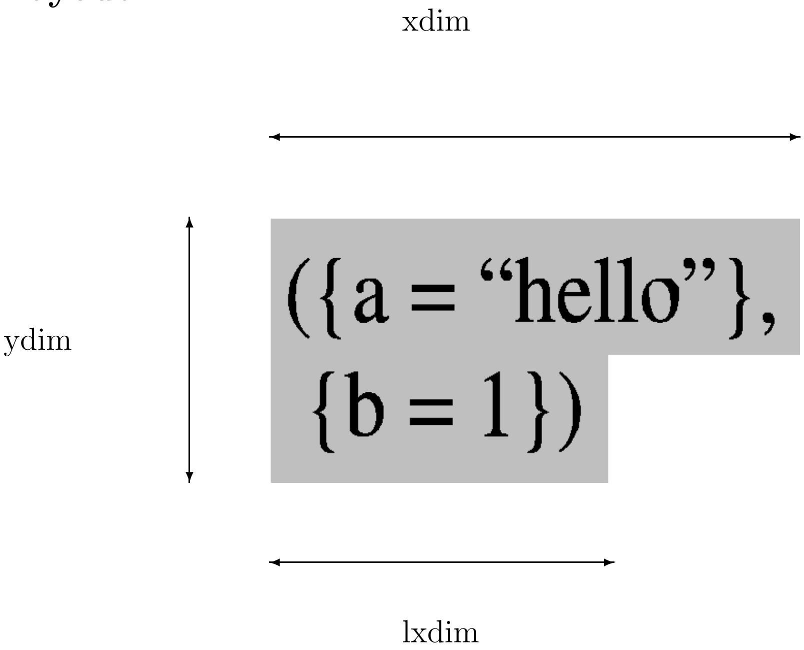

Abbildung 4.2: Bounding-Box einer Wertdarstellung

Die Anordnungsberechnung orientiert sich an den Varianten der Wertbeschreibung der darzustellenden Werte. Hierbei wird zwischen den Verknüpfungen Akkumulation und Konkatenation und atomaren Wertbeschreibungen unterschieden.

Die graphischen Repräsentationen von Wertbeschreibungen, die über Akkumulation miteinander verknüpft sind, werden abhängig davon, ob diese wiederum Verknüpfungen enthalten, vertikal oder horizontal nebeneinander auf der Zeichenfläche angeordnet. Nur in dem Fall, dass die Verknüpfung keine weiteren Verknüpfungen enthält, wird die horizontale Anordnung gewählt.

Die graphischen Repräsentationen von Wertbeschreibungen, die mit Konkatenation verknüpft sind, werden immer horizontal nebeneinander angeordnet.

Die Anordnungsberechnung dient insbesondere dazu, die Höhe und die Breite der Textdarstellung der darzustellenden Werte zu ermitteln sowie die Breite der jeweils letzten Zeile dieser Darstellung, wie in Abbildung 4.2 gezeigt. Dies entspricht der Berechnung der "bounding box" [1] einer Wertrepräsentation. Die Berechnung erfolgt bottom-up unter Berücksichtigung der Anordnungsvorschriften.

4.4 Zeichnen

Jedem Knoten der internen Beschreibung wird bereits bei seiner Erzeugung eine Gtk-Gruppe zugewiesen, die alle Unterbäume dieses Knotens beinhaltet. Die Gtk-Gruppen sind folglich analog zu der Baumstruktur der internen Beschreibung hierarchisch geordnet. Nach der Erzeugung der internen Beschreibung und der Layout-Berechnung wird die Repräsentation der Datenstrukturen mittels top-down left-to-right-Traverierung auf die Zeichenfläche der Benutzerschnittstelle gezeichnet. Die Textrepräsentationen eines Knotens werden dabei in die dazugehörige Gruppe gesetzt.

4.5 Transientenverwaltung

Der Browser erzeugt für jeden Transienten einen Wächter-Thread, der darauf wartet, dass dieser an einen Wert gebunden wird. Da der Browser mehrere Repräsentationen desselben Transienten erzeugen kann - z.B. durch wiederholtes Inspizieren - ist es sinnvoll, nur jeweils einen Wächter pro Transient dafür einzusetzen, der aber alle diesen Transienten darstellenden Knoten der internen Beschreibung kennen muss. Dazu gibt es ein sogenanntes Transienten-Dictionary, in dem jeweils Tripel, bestehend aus dem inspizierten Transienten-Wert, einem dazugehörigen Wächter-Thread, und einer Liste aller Knoten, die auf diesen Transienten verweisen, gespeichert sind. Abbildung 4.3 zeigt die Form eines Dictionary-Eintrages.

Transient	**Thread**	**Knotenliste**

Abbildung 4.3: Eintrag im Transienten-Dictionary

Der Wächter-Thread reagiert auf die Wertzuweisung an den von ihm überwachten Transienten und aktualisiert die Darstellung. Dabei müssen die Tranisenten-Knoten der Beschreibung durch die Repräsentationen der neuen Werte ersetzt werden. Dies geschieht inkrementell [4.6].

4.6 Inkrementalität

Die Darstellung der inspizierten Alice-Werte auf der graphischen Benutzeroberfläche muss schnell an Veränderungen der inspizierten Werte angepasst werden können. Dies wird unter anderem dann erforderlich, wenn eine Wertzuweisung an einen Transienten erfolgt, so dass die Darstellung des neuen Wertes die Darstellung des Transienten auf der Benutzeroberfläche ersetzen soll. Diese Veränderungen erfolgen inkrementell, d.h. ausgehend von der bereits bestehenden Darstellung wird möglichst wenig Berechnungsaufwand betrieben, um zur neuen Darstellung zu gelangen.

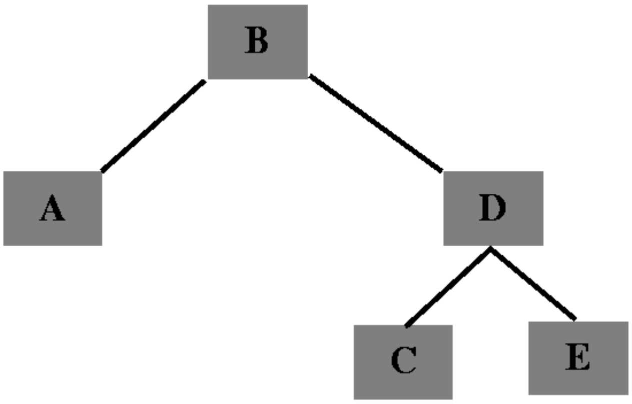

Abbildung 4.4: interne Beschreibung (stark vereinfacht)

Abbildung 4.4 stellt die interne (baumartige) Beschreibung einer Datenstruktur dar. Es wird angenommen, dass die Darstellung des durch Knoten A repräsentierten Wertes ersetzt werden muss.

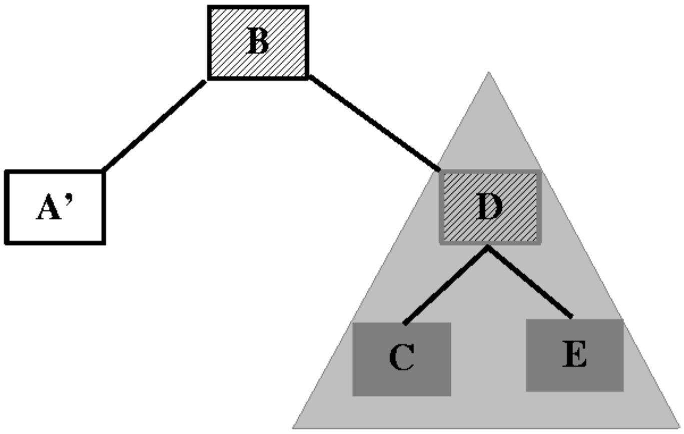

Abbildung 4.5: inkrementelle Veränderung

Abbildung 4.5 zeigt die nötigen Schritte, um zu einer neuen Beschreibung und zu einer neuen Darstellung auf der Benutzeroberfläche zu gelangen. Der betroffene Teilbaum selbst, hier der Knoten A', muss vollkommen neu konstruiert und angeordnet werden, und dessen graphische Repräsentation muss neu gezeichnet werden. Da dies die Anordnung der graohischen Repräsentationen der gesamten baumartigen Beschreibung stören kann, muss auf dem Pfad von diesem Teilbaum zum Wurzelknoten die Anordnung neu berechnet werden. In diesem Beispiel betrifft das den Knoten B. Dies kann zur Folge haben, dass die Textdarstellungen der von der Änderung nicht direkt betroffene Teilbäume an neue Positionen verschoben werden müssen. Von diesen Positionsänderungen abgesehen können diese Teilbäume unverändert beibehalten werden. In dem Beispiel müssen die textuellen Repräsentationen des Teilbaums C,D,E auf der graphischen Benutzeroberfläche verschoben werden. Das ist günstiger, als den gesamten Baum neu zu konstruieren, anzuordnen und zu zeichnen.

4.7 Module

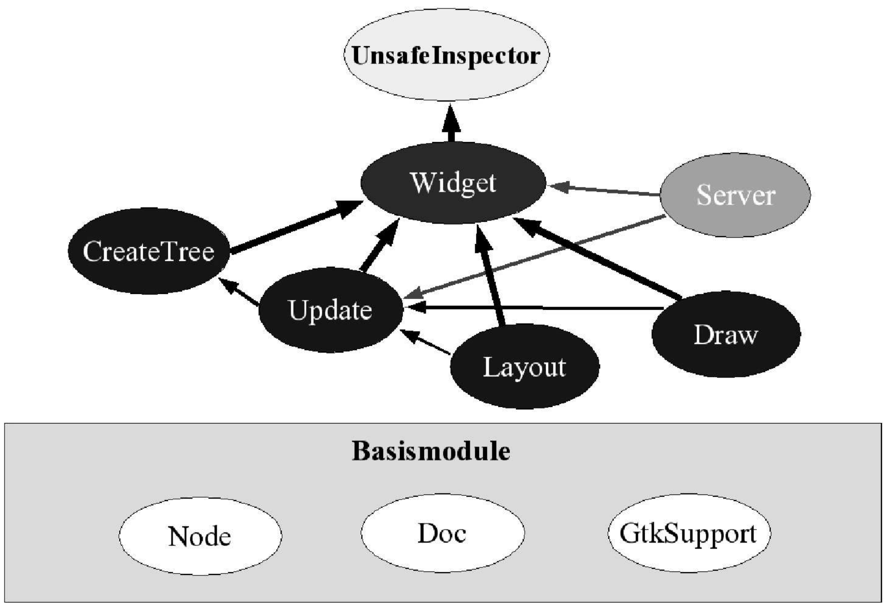

Abbildung 4.6: Module des Browsers

Abbildung 4.6 zeigt die Module des Browsers. Dieser verfügt über zwei Module, die die Datentypen der internen Beschreibung - doc' und doc - enthalten. Das Modul Doc enthält doc', und Node entält doc. Neben den Datentypen beinhalten die Module über entsprechende Accessoren. Da diese während der Erzeugnung, der Anordnungsberechnung, dem Zeichnen und der Darstellungsabänderung verwendet werden, werden diese Module in vielen anderen Modulen benutzt.
Das Modul GtkSupport kapselt die Gtk-Funktionalität gegenüber allen anderen Modulen des Browsers ab. Als einziges Modul hat es direkten Zugriff auf die

Gtk-Prozeduren. Die anderen Module des Browsers greifen dabei auf GtkSupport zurück. Die Module CreateTree, Layout, und Draw repräsentieren die drei Phasen des Inspektionsvorgangs, während das Modul Update die notwendigen Mechanismen repräsentiert, um die Darstellung an Veränderungen anzupassen. Das Modul Widget verfügt über die graphische Benutzeroberfläche und steuert die Darstellungserzeugung bzw. Veränderung. Letztendlich macht das Modul UnsafeInspector die für den Benutzer relevanten Prozeduren und Datentypen nach außen sichtbar.

5 Ausblick

Zur Zeit verwenden wir ein Gtk-Binding für Mozart [6] zur Erzeugung der graphischen Benutzeroberfläche. Allerdings ist es auch möglich, den Browser an eine von Robert Grabowski im Rahmen seines Fortgeschrittenenpraktikums neu implementierte Gtk-Schnittstelle für Alice [7] anzuschließen.

Desweiteren ist der Alice-Browser nicht typsicher, da der Benutzer zur Zeit ein Tupel aus Wert und mithilfe des Typreflektors reflektierten Typs an die inspect-Funktion übergibt. Eine automatische Typreflektion würde die Benutzung des Tools erleichtern und eine Fehlbenutzung ausschließen. In diesem Fall müsste der Benutzer nur noch den zu inspizierenden Wert übergeben.

Mithilfe eines übergeordneten Funktors ließe sich der Browser einkapseln.

Literatur

[1] Thorsten Brunklaus 2000. *Der Oz Inspector - Browsen: Interaktiver, einfacher, effizienter.* Diplomarbeit, Universität des Saarlandes, Fachbereich Informatik.

[2] Derek Oppen 1980. *Prettyprinting.* ACM Transactions on Programming Languages and Systems, Bd. 2, Nr. 4, 1980, pp 465-483.

[3] Andrew Kennedy 1996. *Drawing Trees.* Journal of Functional Programming, Bd. 6(3), pp. 527-534.

[4] Alice-Homepage. *www.ps.uni-sb.de/alice/*

[5] GTK+: The Gimp Toolkit.*www.gtk.org.*

[6] The Mozart Programming System. *www.mozart-oz.org/.*

[7] Robert Grabowski 2003. *Eine Gtk-Schnittstelle für Alice.* FoPra-Ausarbeitung, Lehrstuhl Prof. G. Smolka, Universtität des Saarlandes.