Verträge für die funktionale Programmierung
Design und Implementierung

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

17. März 2006

(Die Folien sind online verfügbar: .../~ralf/talks.html#T48.)

Motivation

Ein wesentliches Qualitätsmerkmal von Software ist Zuverlässigkeit:

- Korrektheit: die Software tut, was sie soll.
- Robustheit: die Software kommt mit unerwarteten Situationen zurecht.

Es gibt verschiedene Ansätze, um die Zuverlässigkeit von Software zu erhöhen:

- formaler Korrektheitsbeweis,
- Typsysteme (statisch, dynamisch),
- systematisches Testen,
- „design by contract“.

☞ Die Ansätze sind nicht konkurrierend; sie ergänzen sich eher.

Überblick

„Design by contract“ ist eine in der objekt-orientierten Welt weit verbreitete Entwurfsmethode.

In diesem Vortrag: Übertragung der Idee auf die funktionale Programmierung unter besonderer Berücksichtigung von

- Funktionen höherer Ordnung,
- algebraischen Datentypen,
- parametrisiertem Typpolymorphismus.

Besonderheit: der Ansatz erfordert keine Spracherweiterung, sondern wird im wesentlichen mit den Bordmitteln funktionaler Sprachen, hier Haskell, als Bibliothek realisiert.

Gliederung

- ✖ Syntax von Verträgen
- ✖ Beispiele
- ✖ Semantik von Verträgen
- ✖ Zusammenfassung

Syntax: Basisverträge

Ein Vertrag spezifiziert eine gewünschte Eigenschaft. Zum Beispiel:

$$nat \ : \ Contract \ Int$$
$$nat = \{ \, i \mid i \geqslant 0 \, \}$$

☞ Ist x eine Variable vom Typ σ und e ein Boole'scher Ausdruck, dann ist $\{ \, x \mid e \, \}$ ein Vertrag vom Typ $Contract \ \sigma$, ein sogenannter Basisvertrag.

$$true \ : \ Contract \ \alpha$$
$$true = \{ \, x \mid True \, \}$$

$$nonempty \ : \ Contract \ [\alpha]$$
$$nonempty = \{ \, x \mid not \ (null \ x) \, \}$$

☞ Verträge sind Bürger erster Klasse: sie können an Namen gebunden werden, Argument oder Ergebnis von Funktionen sein etc.

Syntax — Verträge schließen

Mit Hilfe der Funktion

$$assert : Contract\ \alpha \rightarrow (\alpha \rightarrow \alpha)$$

kann eine Eigenschaft zugesichert werden (ein Vertrag geschlossen werden).

$$\begin{aligned} &mersenne\ :\ Int \\ &mersenne = assert\ prime\ (2^{30402457} - 1) \\ \\ &prime\ :\ Contract\ Int \\ &prime = \{\ n \mid factors\ n == [1, n]\ \} \end{aligned}$$

Syntax — Funktionsverträge

Eigenschaften von Funktionen können mit Hilfe des Kombinators '$\mapsto$' spezifiziert werden:

$$nonempty \mapsto nat$$

Hier spezifiziert $nonempty$ den Definitionsbereich und nat den Wertebereich; $nonempty$ ist die Vorbedingung und nat die Nachbedingung.

Die Nachbedingung darf zusätzlich vom Funktionsargument abhängen:

$$(n : nat) \mapsto \{\, r \mid n < r \,\}$$

☞ Ist x eine Variable vom Typ σ_1 und sind e_1 und e_2 Verträge vom Typ $Contract\ \sigma_1$ bzw. $Contract\ \sigma_2$, dann ist $(x : e_1) \mapsto e_2$ ein Vertrag vom Typ $Contract\ (\sigma_1 \to \sigma_2)$, ein sogenannter Funktionsvertrag.

Verträge — Rechte, Pflichten, Vertragsbruch

Die Zusicherung $e_1 \mapsto e_2$ hat den Charakter eines juristischen Vertrages.

☞ Ein Vertrag regelt Rechte und Pflichten.

Partei	Pflichten	Rechte
Kunde	erfülle Vorbedingung e_1	verwende Nachbedingung e_2
Lieferant	erfülle Nachbedingung e_2	verwende Vorbedingung e_1

Die Rechte des einen sind die Pflichten des anderen.

☞ Wird eine Zusicherung zur Laufzeit verletzt, ist die Software fehlerhaft.

Wird die Vorbedingung verletzt, liegt der Fehler beim Kunden.
Wird die Nachbedingung verletzt, liegt der Fehler beim Lieferanten.

Vertragsbruch — Funktionen erster Ordnung

$$
\begin{aligned}
&inc, dec : Int \to Int \\
&inc = assert\ (nat \mapsto nat)\ (\lambda n \to n + 1) \\
&dec = assert\ (nat \mapsto nat)\ (\lambda n \to n - 1)
\end{aligned}
$$

Interaktive Sitzung:

$Contracts\rangle\ \ inc\ 5$
6
$Contracts\rangle\ \ inc\ (-5)$
*** assertion failed: the argument -5 is to blame.
$Contracts\rangle\ \ dec\ 5$
4
$Contracts\rangle\ \ dec\ 0$
*** assertion failed: the function dec is to blame.

Vertragsbruch — Funktionen höherer Ordnung

$$
\begin{aligned}
&slope\ :\ (Double \to Double) \to (Double \to Double \to Double)\\
&slope = assert\ ((pos \mapsto pos) \mapsto (pos \mapsto pos \mapsto pos))\\
&\qquad\qquad (\lambda f\ a\ b \to (f\ b - f\ a)\ /\ (b - a))
\end{aligned}
$$

Interaktive Sitzung:

$Contracts\rangle\ \ slope\ (\lambda x \to x + 1)\ (-1.0)\ 1.0$
*** assertion failed: the argument -1.0 is to blame.
$Contracts\rangle\ \ slope\ (\lambda x \to x - 1)\ 2.0\ 3.0$
1.0
$Contracts\rangle\ \ slope\ (\lambda x \to x - 1)\ 0.5\ 1.0$
*** assertion failed: the argument $\lambda x \to x - 1$ is to blame.
$Contracts\rangle\ \ slope\ (\lambda x \to 1\ /\ x)\ 2.0\ 3.0$
*** assertion failed: the function $slope$ is to blame.

☞ Vertragsbrüche werden nur dann erkannt, wenn der jeweilige Wert außerhalb seiner Spezifikation verwendet wird.

Syntax: Listenverträge und Konjunktion

Verträge auf Listen:

$$[nat]$$
$$[nat \mapsto nat]$$

Ist e ein Vertrag vom Typ $Contract\ \alpha$, dann ist $[e]$ ein Vertrag vom Typ $Contract\ [\alpha]$.

Verträge können mit „und“ verknüpft werden.

$$nat\ \&\ \{\, n \mid n \leqslant 4711 \,\}$$

Gliederung

- ✔ Syntax von Verträgen
- ✖ Beispiele
- ✖ Semantik von Verträgen
- ✖ Zusammenfassung

Beispiele: Primfaktorenzerlegung

Im folgenden ist f' die 'gesicherte' Variante von f.

$$
\begin{array}{l}
\mathit{prime\text{-}factors}' \;:\; \mathit{Int} \to [\mathit{Int}] \\
\mathit{prime\text{-}factors}' = \mathit{assert}\ ((n : \mathit{pos}) \mapsto [\mathit{prime}]\ \&\ \{\, \mathit{fs} \mid \mathit{product}\ \mathit{fs} == n \,\})\ \mathit{prime\text{-}factors}
\end{array}
$$

☞ $\mathit{prime\text{-}factors}$ ist invers zu $\mathit{product}$. Dieses Idiom lässt sich mit Hilfe einer Funktion höherer Ordnung ausdrücken:

$$
\begin{array}{l}
\mathit{inverse} \quad : (\mathit{Eq}\ \beta) \Rightarrow (\alpha \to \beta) \to \mathit{Contract}\ (\beta \to \alpha) \\
\mathit{inverse}\ f = (x : \mathit{true}) \mapsto \{\, y \mid f\ y == x \,\} \\
\\
\mathit{prime\text{-}factors}' \;:\; \mathit{Int} \to [\mathit{Int}] \\
\mathit{prime\text{-}factors}' = \mathit{assert}\ ((\mathit{pos} \mapsto [\mathit{prime}])\ \&\ \mathit{inverse}\ \mathit{product})\ \mathit{prime\text{-}factors}
\end{array}
$$

Beispiele: Sortieren

$$fast\text{-}sort' \ : \ (Ord\ \alpha) \Rightarrow [\alpha] \rightarrow [\alpha]$$
$$fast\text{-}sort' = assert\ (true \mapsto ord)\ fast\text{-}sort$$

Der Vertrag ord schränkt Listen auf geordnete Listen ein.

☞ Wir haben nicht spezifiziert, dass die Ausgabeliste eine Permutation der Eingabeliste ist.

Beispiele: Sortieren—Fortsetzung

Sei $bag : [\alpha] \to \wr\alpha\int$ eine Funktion, die eine Liste in eine Multimenge überführt.

$$
\begin{array}{l}
fast\text{-}sort' \ : \ (Ord\ \alpha) \Rightarrow [\alpha] \to [\alpha] \\
fast\text{-}sort' = assert\ ((x : true) \mapsto ord\ \&\ \{\ s \mid bag\ x == bag\ s\ \})\ fast\text{-}sort
\end{array}
$$

☞ *fast-sort* verändert nicht die Vielfachheit von Elementen. Auch dieses Idiom lässt sich mit Hilfe einer Funktion höherer Ordnung 'einfangen':

$$
\begin{array}{l}
preserve : (Eq\ \beta) \Rightarrow (\alpha \to \beta) \to Contract\ (\alpha \to \alpha) \\
preserve\ f = (x : true) \mapsto \{\ y \mid f\ x == f\ y\ \} \\
\\
fast\text{-}sort' = assert\ ((true \mapsto ord)\ \&\ preserve\ bag)\ fast\text{-}sort
\end{array}
$$

☞ Eine schwächere Zusicherung: *preserve length*.

Beispiele: Sortieren—Fortsetzung

Alternativ können wir *fast-sort* mit Hilfe einer vertrauenswürdigen Sortierfunktion spezifizieren.

$$
\begin{aligned}
&\mathit{fast\text{-}sort'} \;:\; (\mathit{Ord}\ \alpha) \Rightarrow [\alpha] \to [\alpha] \\
&\mathit{fast\text{-}sort'} = \mathit{assert}\ ((x : \mathit{true}) \mapsto \{\, s \mid s == \mathit{trusted\text{-}sort}\ x \,\}) \\
&\qquad\qquad\qquad\qquad \mathit{fast\text{-}sort}
\end{aligned}
$$

Ein weiteres Idiom:

$$
\begin{aligned}
&\mathit{is} : (\mathit{Eq}\ \beta) \Rightarrow (\alpha \to \beta) \to \mathit{Contract}\ (\alpha \to \beta) \\
&\mathit{is}\ f = (x : \mathit{true}) \mapsto \{\, y \mid y == f\ x \,\} \\
&\\
&\mathit{fast\text{-}sort'} = \mathit{assert}\ (\mathit{is}\ \mathit{trusted\text{-}sort})\ \mathit{fast\text{-}sort}
\end{aligned}
$$

Beispiele: while

Kontrollkonstrukte wie $while$ müssen nicht gesondert behandelt werden.

$$
\begin{aligned}
&while : (\alpha \to Bool) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \\
&while\ p\ f\ a = \textbf{if}\ p\ a\ \textbf{then}\ while\ p\ f\ (f\ a)\ \textbf{else}\ a \\
\\
&while' : Contract\ \alpha \to (\alpha \to Bool) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \\
&while'\ inv = assert\ (true \mapsto (inv \mapsto inv) \mapsto (inv \mapsto inv)) \\
&\qquad\qquad\qquad\qquad while
\end{aligned}
$$

☞ Die Invariante wird an $while'$ übergeben!

Gliederung

- ✔ Syntax von Verträgen
- ✔ Beispiele
- ✖ Semantik von Verträgen
- ✖ Zusammenfassung

Semantik

☞ Wir spezifizieren die Semantik von Verträgen, indem wir eine Implementierung in Haskell angeben (hier ohne Schuldzuweisungen).

Die konkrete Syntax für Verträge lässt sich einfach in Haskell Syntax überführen.

konkrete Syntax	Haskell Syntax
$\{ x \mid p\ x \}$	$Prop\ (\lambda x \rightarrow p\ x)$
$(x : c_1) \mapsto c_2\ x$	$Function\ c_1\ (\lambda x \rightarrow c_2\ x)$
$[c]$	$List\ c$
$c_1\ \&\ c_2$	$And\ c_1\ c_2$

☞ Variablenbindende Konstrukte werden auf λ-Ausdrücke abgebildet.

Semantik: *Contract*

Verträge lassen sich somit als Elemente des folgenden algebraischen Datentyps repräsentieren.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Contract} : * \to * \ \textbf{where} \\
\quad \mathit{Prop} \quad\ : (\alpha \to \mathit{Bool}) \to \mathit{Contract}\ \alpha \\
\quad \mathit{Function} : \mathit{Contract}\ \alpha \to (\alpha \to \mathit{Contract}\ \beta) \to \mathit{Contract}\ (\alpha \to \beta) \\
\quad \mathit{List} \quad\ : \mathit{Contract}\ \alpha \to \mathit{Contract}\ [\alpha] \\
\quad \mathit{And} \quad\ : \mathit{Contract}\ \alpha \to \mathit{Contract}\ \alpha \to \mathit{Contract}\ \alpha
\end{array}
$$

☞ *Contract* ist ein sogenannter verallgemeinerter algebraischer Datentyp, englisch GADT oder phantom type.

Semantik: *assert*

Die Funktion *assert* ist induktiv definiert:

$$
\begin{array}{lll}
assert & : Contract\ \alpha \to (\alpha \to \alpha) & \\
assert\ (Prop\ p) & a & \\
\quad \mid p\ a & & = a \\
\quad \mid otherwise & & = error\ \texttt{"assertion failed"} \\
assert\ (Function\ c_1\ c_2)\ f & & = (\lambda x \to (assert\ (c_2\ x) \cdot f)\ x) \cdot assert\ c_1 \\
assert\ (List\ c) & as & = map\ (assert\ c)\ as \\
assert\ (And\ c_1\ c_2) & a & = (assert\ c_2 \cdot assert\ c_1)\ a
\end{array}
$$

☞ *assert* c ist die Identität, wenn c nicht verletzt wird.

Für Verträge der Form $c_1 \mapsto c_2$ vereinfacht sich *assert* zu

$$assert\ (c_1 \mapsto c_2)\ f = assert\ c_2 \cdot f \cdot assert\ c_1$$

Optimierung von Verträgen

Hat man die Korrektheit einer Funktion mit externen Mitteln nachgewiesen (mit einem Theorembeweiser), so lässt sich der zugehörige Vertrag 'optimieren'.

Beispiel: Die Funktion inc ist korrekt:

$$inc \ : \ Int \to Int$$
$$inc = assert \ (nat \mapsto nat) \ (\lambda n \to n + 1)$$

Ist das Argument eine natürliche Zahl, dann auch das Ergebnis. Somit lässt sich die Zusicherung wie folgt optimieren:

$$inc = assert \ (nat \mapsto true) \ (\lambda n \to n + 1)$$

☞ Die Vorbedingung kann nicht entfallen, da nicht garantiert werden kann, dass inc korrekt aufgerufen wird.

Vertragserfüllung

Eine Funktion f erfüllt den Vertrag c genau dann, wenn

$$assert\ c\ f = assert\ c^+\ f$$

wobei c^+ wie folgt definiert ist:

$$\begin{array}{ll} (Prop\ p)^+ & = true \\ (Function\ c_1\ c_2)^+ & = Function\ c_1^-\ (\lambda x \rightarrow (c_2\ x)^+) \\ \\ (Prop\ p)^- & = Prop\ p \\ (Function\ c_1\ c_2)^- & = Function\ c_1^+\ (\lambda x \rightarrow (c_2\ x)^-) \end{array}$$

In den restlichen Fällen werden $(\cdot)^+$ und $(\cdot)^-$ einfach propagiert.

☞ *inc* und *while* erfüllen ihren Vertrag, *dec* und *slope* hingegen nicht.

Gliederung

- ✔ Syntax von Verträgen
- ✔ Beispiele
- ✔ Semantik von Verträgen
- ✖ Zusammenfassung

Zusammenfassung

Wir haben eine EDSL (embedded domain-specific language) für Verträge kennengelernt:

- Verträge sind integraler Bestandteil der Programmiersprache (keine separate Sprache für Verträge wie zum Beispiel in Eiffel),
- Verträge sind Bürger erster Klasse,
- es lassen sich eigene Abstraktionen definieren,
- Funktionen höherer Ordnung werden in natürlicher Weise unterstützt.

Ausblick:

- Verträge für algebraische Datentypen,
- rekursive Verträge,
- Verträge für polymorphe Funktionen.

Literatur

Typed Contracts for Functional Programming, R. Hinze, J. Jeuring und A. Löh. Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), 24–26 April 2006, Fuji Susono, Japan.

Anhang: Bibliotheksfunktionen

$$
\begin{aligned}
&pos \;:\; (Num\ \alpha) \Rightarrow Contract\ \alpha \\
&pos = \{\, x \mid x > 0 \,\}
\end{aligned}
$$

$$
\begin{aligned}
&ord \;:\; (Ord\ \alpha) \Rightarrow Contract\ [\alpha] \\
&ord = \{\, x \mid ordered\ x \,\} \\
&\\
&ordered : (Ord\ \alpha) \Rightarrow [\alpha] \rightarrow Bool \\
&ordered\ [\,] \qquad\qquad\quad = True \\
&ordered\ [a] \qquad\qquad\quad = True \\
&ordered\ (a_1 :: a_2 :: as) = a_1 \leqslant a_2 \wedge ordered\ (a_2 :: as)
\end{aligned}
$$