Zwischenbericht: Rationale Zahlen für IML

Christian Güdel, Peter Rudolf von Rohr

Compilerbau, HS 2015, Team GR

Abstract

Unsere Erweiterung für IML führt rationale Zahlen ein. Rationale Zahlen werden als Bruch von zwei Integern z (Zähler) und n (Nenner) dargestellt. Für den Nenner *n* ist die Zahl 0 nicht zulässig.

Ein Literal einer rationalen Zahl wird im Programmcode dargestellt als z/n. Zur Laufzeit werden die Brüche jeweils gekürzt, so dass z und n keine gemeinsamen Teiler ausser 1 besitzen.

Lexikalische Syntax

Die rationalen Zahlen werden als neuer Datentyp *Ratio* in IML eingebaut. Dazu wird die Aufzählung *Type* um den Wert *RATIO* erweitert.

Ein neues Literal für rationale Zahlen wird eingeführt:

Pattern	Beispiel Lexeme	Beispiel Token
`[0-9]+('*[0-9]+)*/[1-9]+('*[0-9]+)*`	1'337/42	LITERAL, RatioVal 1337 42

Durch die zusätzliche Einschränkung dass im Nenner keine führende 0 vorkommen kann, wird das Problem der Division durch 0 elegant gelöst. Als Kompromiss können Zahlen wie 0001 nicht im Nenner verwendet werden.

Als Konsequenz wird ein neues Attribut *RatioVal* eingeführt, welches zwei ganzzahlige Werte z und n akzeptiert.

Zusätzlich fügen wir folgende Schlüsselwörter ein:

Pattern	Token	Funktion
ratio	(TYPE, RATIO)	Deklaration einer Variable für eine rationale Zahl
num	(RATIOOPR, NUMERATOR)	Zähler einer rationalen Zahl als *int* auslesen
denom	(RATIOOPR, DENOMINATOR)	Nenner einer rationalen Zahl als *int* auslesen
floor	(RATIOOPR, FLOOR)	Abrunden auf die grösste ganze Zahl, die kleiner oder gleich gross wie die rationale Zahl ist, als *int*
ceil	(RATIOOPR, CEIL)	Aufrunden auf die kleinste ganze Zahl, die grösser oder gleich gross wie die rationale Zahl ist, als *int*
round	(RATIOOPR, ROUND)	Runden auf die am nächsten liegende ganze Zahl, 0.5 wird auf 1 gerundet, als *int*.

Grammatikalische Syntax

Wir erweitern die Grammatik um die folgende Regel:

```
monadicOpr ::= RATIOOPR
```

Die Operatoren der Gruppe RATIOOPR haben die gleiche Präzedenz wie diejenigen der Gruppe der monadischen Operatoren (*monadicOpr*). Sie sind nicht assoziativ. Dies ergibt keinen Sinn, da die Operatoren einen anderen Rückgabe- als Eingabetyp aufweisen. Somit sieht die Operatorentabelle neu wie folgt aus:

	Präzedenz	Assoziativität
MONADICOPR (NOT ADDOPR)	4 (=höchste)	Keine
RATIOOPR **(num denom floor ceil round)**	**4**	**Keine**
MULTOPR (* divE modE)	3	Links
ADDOPR (+ -)	2	Links
RELOPR (< <= > >= = /=)	1	Keine
BOOLOPR (&? |?)	0 (= niedrigste)	Rechts

Deklaration:

```
const r1 : ratio;
const r2 : ratio;
const r3 : ratio;
```

Initialisierung:

```
r1 init := 4/3;
r2 init := 0/1;
r3 init := 24/24;
r4 init := 12/3;
```

Zugriff auf den Zähler:

```
debugout num r1; // Ausgabe: 4
debugout num r2; // Ausgabe: 0
debugout num r3; // Ausgabe: 1
debugout num r4; // Ausgabe: 4
```

Zugriff auf den Nenner:

```
debugout denom r1; // Ausgabe : 3
debugout denom r2; // Ausgabe : 1
debugout denom r3; // Ausgabe : 1
debugout denom r4; // Ausgabe : 1
```

Abrunden auf die nächstkleinere oder gleich grosse ganze Zahl:

```
debugout floor r1; // Ausgabe: 1
debugout floor r2; // Ausgabe: 0
debugout floor r3; // Ausgabe: 1
debugout floor r4; // Ausgabe: 4
```

Aufrunden auf die nächstgrössere oder gleich grosse ganze Zahl:

```
debugout ceil r1; // Ausgabe: 2
debugout ceil r2; // Ausgabe: 0
debugout ceil r3; // Ausgabe: 1
debugout ceil r4; // Ausgabe: 4
```

Runden auf die am nächsten liegende ganze Zahl:

```
debugout round r1; // Ausgabe: 1
debugout round r2; // Ausgabe: 0
debugout round r3; // Ausgabe: 1
debugout round r4; // Ausgabe: 4
```

Ausgabe einer relationalen Zahl mittels debugout:

```
debugout 51/10; // Ausgabe: 5.1
debugout 2/3    // Ausgabe: 0.[6] oder 0.6666666667 oder 2/3
```

Für periodische Zahlen ist die Ausgabe noch nicht festgelegt. Die erste Möglichkeit wird im Moment favorisiert.

Kontext- und Typeinschränkungen

Die Operatoren der Gruppe *RATIOOPR* unterstützen als Argument nur solche vom Typ *RATIO*. Für die Operatoren der Gruppen *RELOPR*, *ADDOPR* und *MULTOPR* werden gemischte Argumenttypen aus *RATIO* und *INT* unterstützt. Dabei wird jeweils die *INT*-Seite in eine *RATIO*-Zahl konvertiert. Der Rückgabetyp von *ADDOPR* und *MULTOPR* bei gemischten Argumenttypen ist dann ebenfalls *RATIO*.

Die Zuweisung eines *RATIO*-Ausdrucks an eine *INT*-Variable führt zu einem Type-Error.

Vergleich mit anderen Programmiersprachen

Common Lisp unterstützt den Datentyp *RATIO* und bietet ebenfalls Funktionen zur Bestimmung des Zählers (*numerator*) bzw. des Nenners (*denominator*).

In Haskell gibt es einen *Rational* Datentyp welcher ähnlich funktioniert.

Java unterstützt rationale Zahlen nicht direkt. Es gibt aber Bibliotheken, mit denen die Funktionalität nachgebildet werden kann[1,2].

Warum wurde die Erweiterung so entworfen und nicht anders?

Beim Entwurf der Erweiterung wurde darauf geachtet, dass der Datentyp *RATIO* möglichst konsistent mit den anderen Datentypen verwendet werden kann. Wir haben uns bei der Syntax an bestehenden Implementationen in Lisp[3] sowie Python[4] orientiert.

Durch die automatische Konvertierung bei gemischten Ausdrücken wird der Umgang mit rationalen Zahlen weiter vereinfacht.

Typenkonvertierungen bei denen Präzision verloren gehen könnte, müssen explizit mit *floor*, *ceil* oder *round* durchgeführt werden. Damit wird vom Programmierer eine explizite Entscheidung gefordert und implizite Annahmen werden verhindert.

Das Literal hat eine gewisse Ähnlichkeit mit einem Bruchstrich, so können rationale Zahlen schnell als solche erkannt werden.

[1] http://jscience.org/api/org/jscience/mathematics/number/Rational.html
[2] http://commons.apache.org/proper/commons-math/userguide/fraction.html
[3] https://en.wikipedia.org/wiki/Rational_data_type
[4] https://docs.python.org/3.1/library/fractions.html

Anhang: IML Testprogramme

Akzeptiert einen rationalen Input und gibt eine ganze Zahl aus, wenn der Input ganzzahlig dargestellt werden kann.

```
program isInteger
global
   const m:ratio;
   const isInteger:bool;

   proc checkInteger(copy const m:ratio, ref var isInteger:bool)
   do
          isInteger := denom m = 1
   endproc

do
   debugin m init;
   call checkInteger(m, isInteger init);
   if isInteger = true then
          debugout num m
   endif
endprogram
```

Liest die erreichten Punkte und die maximale Punktzahl ein und gibt die erreichte Note berechnet nach dem linearen Notenmassstab gerundet auf Zehntel aus:

```
program notenberechnung
global
   const pkt:int
   const max:int
   const grade:ratio

   proc calculate(copy const pkt:int, copy const max:int, ref var g:ratio)
   do
          g := pkt * 5 / max + 1
          g := round (grade * 10) / 10
   endproc

do
   debugin pkt init;
   debugin max init;

   call calculate(pkt, max, grade init);

   debugout grade
endprogram
```