Meta-Operatoren in MOSTflexiPL

Christian Heinlein

Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@hs-aalen.de

Abstract. Meta-Operatoren sind spezielle Operatoren in der Programmiersprache MOSTflexiPL, deren Anwendungen über gewöhnliche Operatoranwendungen „gestülpt“ werden können, um diese in sehr flexibler Weise zu parametrisieren. Abhängig von Laufzeitbedingungen, können so aus einem einzigen Ausdruck im Quelltext zur Laufzeit unterschiedliche konkrete Ausdrücke entstehen. Dies ist insbesondere zur Verarbeitung und Weitergabe optionaler, alternativer und wiederholter Syntaxteile eines Operators nützlich.

1 Einleitung

MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language) [1] ist eine Programmiersprache, deren Syntax vom Anwender nahezu beliebig erweitert und angepasst werden kann. Aufbauend auf einer kleinen Menge vordefinierter Grundoperatoren (z. B. für Arithmetik, Logik und Ablaufsteuerung), können nach Belieben weitere Operatoren für unterschiedlichste Zwecke definiert werden. Da Operatoren beliebig viele Namen besitzen und auf beliebig viele Operanden angewandt werden können, decken sie neben den üblichen Präfix-, Infix- und Postfix-Operatoren (z. B. `-•` für Vorzeichenwechsel, `•+•` für Addition oder `•!` für Fakultät; `•` bezeichnet jeweils einen Operanden) auch Mixfix-Operatoren (z. B. `•[•]` für den Zugriff auf Elemente eines Felds), Steueranweisungen (z. B. `if•then•else•end`), Typkonstruktoren (z. B. `List•`) und Deklarationsformen (z. B. `•:•`) ab.

2 Beispiele

Beispielsweise definiert

```
(x:int) "²" -> (int = x * x)
```

einen neuen Operator `•²`, der das Quadrat seines Operanden berechnet und anschließend in Ausdrücken wie z. B. `10²` oder `(2+3)²` verwendet werden kann.

Die Signatur (links vom Pfeil) eines solchen Operators ist allgemein eine Folge von Namen und Parameterdeklarationen. Ein Name ist entweder eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt (z. B. `x` oder `abc123`) oder eine Folge beliebiger Zeichen innerhalb von Anführungszeichen (die nur bei der Definition des

Operators, aber nicht bei seiner Verwendung angegeben werden). Eine Parameterdeklaration ist von der Gestalt `(`*`name`*`:`*`type`*`)` mit einem Namen *name* wie gerade definiert und einem Typ *type*. Rechts vom Pfeil steht etwas der Gestalt `(`*`type`*`=`*`impl`*`)` mit einem Typ *type* (dem Resultattyp des Operators) und einem Ausdruck *impl*, der die Implementierung des Operators (wie in funktionalen Sprachen) darstellt.

Noch allgemeiner kann die Signatur eines Operators unter Verwendung der bekannten EBNF-Symbole auch optionale, alternative und wiederholte Teile enthalten, zum Beispiel:

```
sum of (x:int) { and (y:int) } -> (int = ......);

calc [minus] (x:int) { (plus|minus) (y:int) } -> (int = ......);

dnf [not] (a:bool) { and [not] (b:bool) }
{ or [not] (c:bool) { and [not] (d:bool) } }
-> (bool = ......);

print { [(T:type)] (x:T) } -> (bool = ......)
```

Mögliche Verwendungen der so definierten Operatoren wären dann zum Beispiel:

```
sum of 1;
sum of 1 and 2;
sum of 1 and 2 and 3;

calc 1 plus 2;
calc minus 1 plus 2 minus 3;

dnf not a or b and not c;
dnf a and b or not c or d and e and not f;

print 1 2.5 "hallo"
```

Weitere Beispiele für Operatoren mit optionalen, alternativen und wiederholten Teilen sind Steueranweisungen wie z. B. `if•then•{elseif•then•}[else•]end` und `try•{catch•then•}[finally•]end`.

3 Probleme

Da solche Operatoren auf (zum Teil unendlich viele) unterschiedliche Arten angewandt werden können, besteht für ihre Implementierung das Problem herauszufinden, wie genau der jeweilige Operator jeweils angewandt wurde, d. h. ob ein optionaler Teil bei einer Anwendung vorhanden ist oder nicht, welche von mehreren Alternativen verwendet wurde und wie oft und auf welche Arten ein wiederholter Teil angegeben wurde. Je nach konkreter Anwendung eines Operators, kann ein Parameter innerhalb eckiger und runder Klammern einen Wert besitzen oder nicht, während Parameter innerhalb geschweifter Klammern sogar beliebig viele Werte besitzen können, die – wie das Bei-

spiel `print` zeigt – sogar unterschiedliche Typen besitzen können, sodass eine Zusammenfassung aller Wert in einem Feld o. ä. im allgemeinen nicht möglich ist.

Ein weiteres Problem besteht darin, solche Teile einer Operatoranwendung entweder unverändert oder geeignet angepasst an andere Operatoren weiterzugeben. Beispielsweise ist es wünschenswert, dass ein Operator `avg of•{and•}` zur Berechnung des Durchschnitts beliebig vieler Werte diese Werte an den Operator `sum of•{and•}` weitergeben kann, um auf einfache Art und Weise ihre Summe zu berechnen. Ebenso könnte ein Operator `cnf[not]•{or[not]•}{and[not]•{or[not]•}}` (conjunctive normal form) unter Ausnutzung der Regeln von De Morgan auf einfache Art und Weise unter Verwendung des Operators `dnf[not]•{and[not]•}{or[not]•{and[not]•}}` (disjunctive normal form) implementiert werden, wenn er die Werte seiner Parameter sowie das Vorhanden- oder Nichtvorhandensein der Namen `not` geeignet an diesen Operator weitergeben könnte.

4 Lösung durch Meta-Operatoren

Eine mögliche Lösung für die zuvor beschriebenen Probleme sind Meta-Operatoren, deren Bedeutung und Verwendung am besten anhand von Beispielen erläutert werden kann:

```
sum of (x:int) { and (y:int) } -> (int = x ^{ + y })
```

Hier wird ausgenutzt, dass es zu jeder Wiederholungsklammer `{...}` in der Signatur eines Operators einen korrespondierenden Meta-Operator **`^{∘}`** gibt, dessen Anwendungen über gewöhnliche Operatoranwendungen „gestülpt“ werden können. (Um gewöhnliche und Meta-Operatoren besser unterscheiden zu können, werden die Namen von Meta-Operatoren jeweils fett gedruckt.) Das Symbol ∘ kennzeichnet im Gegensatz zu • keinen gewöhnlichen Operanden, sondern einen Meta-Operanden, der aus beliebig vielen Namen und (gewöhnlichen) Operanden der „darunterliegenden“ Operatoranwendungen (sowie weiterer Meta-Operator-Anwendungen, für deren Meta-Operanden dasselbe gilt) bestehen kann. Zur Laufzeit (!) wird so eine Meta-Operator-Anwendung dann durch beliebig viele Aneinanderreihungen ihres Meta-Operanden ersetzt, sodass aus dem Ausdruck `x ^{ + y }` dann entweder `x` oder $x + y_1$ oder $x + y_1 + y_2$ usw. werden kann. Die Indizes bei `y` sollen hierbei anzeigen, dass jedes `y` einen anderen Wert besitzen kann, ebenso wie die Konstante `y` in C++ [2] in jedem Durchlauf der Schleife

```
for (const int y : { 1, 2, 3 }) ......
```

einen anderen Wert besitzt. Die konkreten Werte für y_1, y_2, ... stammen hierbei aus der konkreten Operatoranwendung, sodass aus den Anwendungen `sum of 1` bzw. `sum of 1 and 2` bzw. `sum of 1 and 2 and 3` zum Beispiel die Ausdrücke `1` bzw. `1 + 2` bzw. `1 + 2 + 3` in der Implementierung des Operators `sum` werden.

Da MOSTflexiPL eine statisch typisierte Sprache ist, muss der Compiler zur Übersetzungszeit überprüfen, ob jeder Ausdruck, der auf diese Weise zur Laufzeit entstehen

kann, sowohl syntaktisch als auch semantisch korrekt ist, was im obigen Beispiel offensichtlich ist, im allgemeinen aber eine nicht triviale Aufgabe darstellt.

Das folgende Beispiel zeigt die Verwendung weiterer Arten von Meta-Operatoren:

```
calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
    ^[ - | ] x ^{ ^( + | - ) y }
)
```

Zu jeder Verzweigungsklammer `(...|...)` mit zwei Alternativen in der Signatur eines Operators gibt es in der Implementierung des Operators einen korrespondierenden Meta-Operator **`^(`**`◦|◦`**`)`** mit zwei Meta-Operanden (und entsprechend für Klammern mit mehr als zwei Alternativen). Eine Anwendung dieses Meta-Operators wird zur Laufzeit durch einen der beiden Meta-Operanden ersetzt, je nachdem welche Alternative bei der vorliegenden Operatoranwendung verwendet wurde.

Zu jeder Optionsklammer `[...]` in der Signatur eines Operators gibt es in der Implementierung des Operators korrespondierende Meta-Operatoren **`^[`**`◦`**`]`** und **`^[`**`◦|◦`**`]`** mit einem bzw. zwei Meta-Operanden. Eine Anwendung des Meta-Operators **`^[`**`◦`**`]`** wird zur Laufzeit entweder durch den Meta-Operanden oder durch nichts ersetzt, je nachdem ob der optionale Teil bei der vorliegenden Operatoranwendung vorhanden ist oder nicht. Eine Anwendung des Meta-Operators **`^[`**`◦|◦`**`]`** wird zur Laufzeit entweder durch den ersten oder den zweiten Meta-Operanden ersetzt, je nachdem ob der optionale Teil bei der vorliegenden Operatoranwendung vorhanden ist oder nicht.

Aufgrund dieser Regeln entstehen aus den Operatoranwendungen `calc 1 plus 2` bzw. `calc minus 1 plus 2 minus 3` die Ausdrücke `1 + 2` bzw. `-1 + 2 - 3` in der Implementierung des Operators `calc`. Auch hier muss der Compiler wieder überprüfen, dass jeder Ausdruck, der zur Laufzeit entstehen kann, korrekt ist.

5 Ungewöhnliche Verwendungen von Meta-Operatoren

Durch konsequente Anwendung der bis jetzt beschriebenen Regeln für Meta-Operatoren sind auch ungewöhnliche Verwendungen wie zum Beispiel folgendes möglich:

```
strange (x:int) (a: plus | times) (y:int)
                        (b: minus | div) (z:int) -> (int =
    x ^(a: + | * ) y ^(b: - | / ) z
)
```

Um die zu den beiden Verzweigungsklammern gehörenden Meta-Operatoren unterscheiden zu können, sind die Klammern mit den Namen `a` bzw. `b` benannt, die dann auch bei der Verwendung der Meta-Operatoren angegeben werden können. Abhängig von der konkreten Anwendung des Operators `strange` können in seiner Implementierung vier verschiedene Ausdrücke entstehen, deren Operatorbaum aufgrund der üblichen Vorrangregeln für arithmetische Operatoren jeweils unterschiedlich aufgebaut ist:

```
strange x plus y minus z   →  (x + y) - z
strange x plus y div z     →  x + (y / z)
strange x times y minus z  →  (x * y) - z
strange x times y div z    →  (x * y) / z
```

Die Klammern dienen hier nur zur Verdeutlichung der unterschiedlichen Operatorbäume.

6 Vergleichbare Konzepte

Meta-Operatoren haben eine gewisse Ähnlichkeit mit den Sprachmitteln für bedingte Übersetzung in C und C++ [2]: Abhängig davon, ob die Makros `a` und `b` definiert sind oder nicht, entsteht aus dem Code

```
x
    #ifdef a
    +
    #else
    *
    #endif
y
    #ifdef b
    -
    #else
    /
    #endif
z
```

zur Übersetzungszeit ebenfalls einer der in §5 genannten Ausdrücke. Die wesentlichen Unterschiede zu Meta-Operatoren sind jedoch, dass letztere erst zur Laufzeit ersetzt werden und dass der C/C++-Präprozessor keine Möglichkeit für Wiederholungen bietet, sodass Meta-Operatoren wesentlich mehr Flexibilität bieten.

Meta-Operatoren haben außerdem eine gewisse Ähnlichkeit mit Makros in Lisp [3]: Anwendungen des im folgenden definierten Makros `strange` werden abhängig von den Werten der Parameter `a` und `b` zur Laufzeit ebenfalls durch einen der vier in §5 genannten Ausdrücke (in Lisp-Syntax) ersetzt:

```
(defmacro strange (x a y b z)
    (if a
        (if b
            `(- (+ ,x ,y) ,z)
            `(+ ,x (/ ,y ,z))
        )
```

```
            (if b
                `(- (* ,x ,y) ,z)
                `(/ (* ,x ,y) ,z)
            )
        )
    )
```

Ein wesentlicher Unterschied zu Meta-Operatoren ist hier, dass in Lisp keine Überprüfungen zur Übersetzungszeit stattfinden und deshalb prinzipiell Fehler zur Laufzeit auftreten können, wenn einer der entstehenden Ausdrücke nicht korrekt ist. Außerdem kann die Abhängigkeit der resultierenden Ausdrücke von `a` und `b` in Lisp nicht unabhängig voneinander ausgedrückt werden.

7 Benutzerdefinierte Meta-Operatoren

Da die Syntax von MOSTflexiPL beliebig erweiterbar und anpassbar ist, gibt es neben den bis jetzt beschriebenen vordefinierten Meta-Operatoren auch die Möglichkeit, eigene Meta-Operatoren zu definieren. Mit einem geeignet definierten Meta-Operator **`^{•=•..•:○}`** – der neben einem Meta-Operanden ○ auch drei gewöhnliche Operanden • besitzt – würde dann z. B. der Ausdruck

```
sum of 0 ^{ i = 1 .. 3 : and i }
```

zur Laufzeit durch `sum of 0 and 1 and 2 and 3` ersetzt werden.

8 Zusammenfassung und Ausblick

Meta-Operatoren sind Operatoren mit speziellen Meta-Operanden, deren Anwendungen über gewöhnliche Operatoranwendungen „gestülpt“ werden können, um diese in sehr flexibler Weise zu parametrisieren. Abhängig von Laufzeitbedingungen – z. B. wie oft ein wiederholter Syntaxteil bei einer Operatoranwendung vorhanden ist oder ob ein optionaler Teil vorhanden ist oder nicht – können so aus einem einzigen Ausdruck im Quelltext zur Laufzeit mehrere (eventuell sogar unendlich viele) verschiedene Ausdrücke mit ähnlicher oder auch sehr unterschiedlicher Bedeutung entstehen. Damit können optionale, alternative und wiederholte Syntaxteile eines Operators in seiner Implementierung sehr einfach verarbeitet oder an andere Operatoren weitergegeben werden.

Referenzen

[1] C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

[2] B. Stroustrup: *The C++ Programming Language* (Fourth Edition). Addison-Wesley, Upper Saddle River, NJ, 2013.

[3] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.