Pseudo: eine Programmiersprache auf der Basis von Pseudocode zur Unterstützung der akademischen Lehre

Marcus Frenkel **Karsten Weicker**

HTWK Leipzig, Fakultät IMN
04277 Leipzig
Web: portal.imn.htwk-leipzig.de/fakultaet/weicker
E-Mail: {mfrenkel,weicker}@imn.htwk-leipzig.de

Zusammenfassung: Pseudo ist eine auf Pseudocode basierende Programmiersprache, welche in der akademischen Lehre zum Einsatz kommen und hier die Vermittlung und Untersuchung von Algorithmen und Datenstrukturen unterstützen soll. Dieser Beitrag geht auf die Besonderheiten der Sprache sowie mögliche didaktische Szenarien ein.

Einleitung

Ein wichtiger Bestandteil in der Ausbildung von Informatikern ist die umfassende und verständliche Vermittlung von Algorithmen und Datenstrukturen in unterschiedlichen Veranstaltungen auf Bachelor- wie auch Masterniveau. Neben den klassischen asymptotischen Betrachtungen gewinnen auch empirische Untersuchungen immer stärker an Bedeutung – als Grundlage für effektive Designentscheidungen im Informatikeralltag. Das Ziel dieser Arbeit ist die Verknüpfung einer möglichst einfach lesbaren Notation mit einer direkten Ausführbarkeit der Algorithmen. Hiervon versprechen wir uns eine intensivere studentische Auseinandersetzung mit den Inhalten.
In der Lehre reicht die Notation von Algorithmen von rein textueller Beschreibung über oft individuell ausgeprägten Pseudocode hin zu echtem Programmcode – jede mit spezifischen Vor- und Nachteilen.
Textuelle Beschreibungen bieten den höchsten Grad an deskriptiver Freiheit, da sie keinem Formalismus unterworfen sind. Dies birgt den Nachteil einer oft abstrakten und mehrdeutigen Beschreibung. Weitere Nachteile sind hoher Platzbedarf und fehlende Möglichkeiten, die Korrektheit zu prüfen.
Programmcode hingegen bietet neben Eindeutigkeit und Präzision auch Möglichkeiten zur Überprüfung und weitergehenden Untersuchung, da der Code ausgeführt werden kann. In den meisten Programmiersprachen bedingen die strenge Syntax und die umfassenden programmiersprachlichen Konzepte (z.B. Objektorientierung, Ausnahmebehandlung etc.) allerdings eine stark aufgeblähte Beschreibung und verstellen den Blick auf den eigentlichen Algorithmus.
Pseudocode steht zwischen diesen beiden Extremen und geht einen Mittelweg zwischen Freiheit und Exaktheit. Durch mathematische Ausdrücke ergibt sich oft eine hohe Ausdrucksstärke und Kompaktheit, die in den beiden Alternativen nicht möglich ist. Aller-

dings übernimmt Pseudocode die fehlende Ausführbarkeit von textuellen Beschreibungen und auch Fehlinterpretationen sind dank der unterschiedlichen Handhabung von Referenzen bzw. formalen Parametern oder durch fehlende Typinformation, etwa bei der Anweisung $mitte \leftarrow (a+b)/2$, keine Seltenheit.
In der aktuellen Informatiklehre fehlt dementsprechend eine Möglichkeit zur geeigneten Notation von Algorithmen und Datenstrukturen, die sowohl einen genügend großen Detaillierungsgrad bei hoher Abstraktion bietet als auch eine automatische Verarbeitung durch einen Parser und Compiler erlaubt, um die syntaktische und semantische Korrektheit sowie Eindeutigkeit zu gewährleisten. Neben der reinen Ausführung des Codes sollen auch empirische Untersuchungen hinsichtlich Zeit- und Speicherverbrauch ermöglicht werden.

Pseudo

Um die in Kapitel 1 genannten Ziele einer geeigneten Notationsform für Algorithmen und Datenstrukturen zu erreichen, wurde eine neue Programmiersprache mit der Bezeichnung „Pseudo“ – in Anlehnung an den Pseudocode – entwickelt.
Pseudo orientiert sich an einer mathematischen Notation, da diese sowohl knapp, dabei aber auch präzise, ausdrucksstark und eindeutig ist. Da nicht davon auszugehen ist, dass häufig benutzte mathematische Symbole wie das Element-Zeichen $\in$, der Linkspfeil $\leftarrow$ oder das Zeichen für Ungleichheit $\neq$ über gängige Tastaturen eingegeben werden können, werden derartige Symbole über entsprechende Ersatzzeichen und Zeichenketten umschrieben. Das gesamte Erscheinungsbild der Sprache ist derart ausgelegt, dass durch einfaches Parsen und Konvertieren des Codes eine mathematische Notation zur Darstellung auf Folien und in Büchern und Artikeln erstellbar ist.
Als der Sprache zugrundeliegendes Programmierparadigma wurde eine Mischung aus imperativem und objektorientiertem Paradigma gewählt, da auf diese Art die Vorteile beider Paradigmen kombiniert und gleichzeitig ihre Nachteile vermieden werden können. Der objektorientierte Anteil ermöglicht die Umsetzung komplexer, benutzerdefinierter Typen und direkt damit verknüpfter Funktionen zur Erklärung von Datenstrukturen, wohingegen der imperative Anteil die Definition globaler Funktionen und damit die einfache Erklärung von Algorithmen erlaubt. Zusätzlich erlaubt Pseudo ähnlich wie C/C++ die Kontrolle über die Verwendung von Wert- beziehungsweise Referenzzuweisungen, wobei zwei verschiedene Zuweisungsarten – der Linkspfeil $\leftarrow$ als Wertzuweisung und der Rechtspfeil $\rightarrow$ als Referenzzuweisung, eingeführt wurden.
Ein weiterer wichtiger Aspekt von Pseudo ist die vorhandene strenge Typisierung. Dies erfordert zwar zusätzliche Syntaxelemente und erhöht den Platzaufwand geschriebenen Codes, trägt aber erheblich zum leichteren Verständnis dargestellter Algorithmen und Datenstrukturen bei, da die Typen verwendeter Variablen nicht mehr erraten oder abgeleitet werden müssen.
Im Rahmen der strengen Typisierung wurden für Pseudo verschiedene Standarddatentypen festgelegt, die zur Repräsentation von primitiven und zusammengesetzten Daten eingesetzt werden können. Neben üblichen skalaren Typen für natürliche, ganze und reelle Zahlen sowie Zeichenketten und Wahrheitswerte existieren als zusammengesetzte Typen Listen, Felder, Mengen und Tupel mit jeweils eigenen, an der Mathematik orientierten Konstruktoren.

Ein Beispiel für einen in die mathematische Darstellung konvertierten, in Pseudo geschriebenen Algorithmus bietet die folgende Abbildung an Hand von Quicksort:

```
1:    Quicksort⟨𝕋⟩ ∈ ( a ∈ 𝕋? × l,r ∈ ℕ ) → ( ) :
2:          if l < r :
3:                index ← Partition( a,l,r ) ∈ ℕ;
4:
5:                Quicksort( a,l,index − 1 );
6:                Quicksort( a,index + 1,r );
7:
8:    Partition⟨𝕋⟩ ∈ ( a ∈ 𝕋? × l,r ∈ ℕ ) → ℕ :
9:          i ← l,j ← r ∈ ℕ;
10:         p → a_l ∈ 𝕋;
11:
12:         while i < j :
13:               if a_j > p :          j−−;
14:               else if a_i < p :     i++;
15:               else :                ( a_i,a_j ) → ( a_j,a_i );
16:
17:         ( a_l,a_j ) → ( a_j,a_l );
18:         return j;
```

Als weitere Besonderheit von Pseudo wird durch spracheigene Mittel die einfache Parallelisierung sowie die Untersuchung von Algorithmen durch Profiling ermöglicht. Die Realisierung dieser beiden Konzepte erfolgt direkt im Programmcode durch Annotationen.
Die Annotationen erlauben dem Benutzer, den eigentlichen Programmcode um zusätzliche Informationen anzureichern; sie werden durch eine eigene Syntaxstruktur beschrieben, die durch den Austausch einzelner Wörter innerhalb der Struktur den gewünschten Beschreibungseffekt hervorruft. Die Annotationen in Pseudo orientieren sich in ihrer Notation an der Sprache C#, werden also durch in eckige Klammern eingeschlossene Schlüsselwörter definiert, wobei zusätzliche Argumente in runden Klammern angegeben werden. Im Unterschied zu C# erlaubt Pseudo allerdings auch Annotationen direkt im Code einer Funktion, wodurch sich erweiterte Kontrollmöglichkeiten über die Art der Codeausführung ergeben. Neben Möglichkeiten zur Zugriffskontrolle auf Typbestandteile, Implementierungsinformationen für bestimmte Datenstrukturen wie etwa Mengen, Angaben zum Speichermanagement oder der Blockbildung sind die beiden für die Lehre wichtigsten Funktionen von Annotationen das Profiling und die Beschreibung von Parallelität im Programmcode.
Das Profiling erfasst unterschiedlichste Ablaufparameter, wie etwa die Anzahl von Funktionsaufrufen, die Anzahl ausgeführter Instruktionen, die für einen Funktionsaufruf aufgewendete Zeit oder den Speicherverbrauch einer Funktion, aber auch weitere Metriken wie die durchschnittliche Anzahl von Durchläufen einer Schleife oder einer Verzweigung, die Anzahl von Zugriffen auf eine Variable oder die Anzahl von erstellten Instanzen eines Typs. All diese Informationen können durch Pseudo während der Codeausführung ermittelt werden, wobei entweder der gesamte Programmcode oder speziell mar-

kierte Codebestandteile analysiert werden. Die Markierung der Codebestandteile erfolgt über die Annotation `profile`, welche im Zusammenhang mit dem Profiling an Funktions-, Typ- und Variablendeklarationen sowie an Kontrollstrukturen mit untergeordneten Blöcken (wie etwa Schleifen oder Verzweigungen) benutzbar ist. Argumente in der Annotation bestimmen die jeweils benutzten Metriken. Im folgenden Beispiel werden einige Anwendungsbereiche für die Profile-Annotation dargestellt, wobei für den Typ `T` die Anzahl der Instanzen sowie die Anzahl der Zugriffe auf die einzelnen Bestandteile, für die Funktion `g` die Anzahl der beim Aufruf ausgeführten Instruktionen und für die Verzweigung sämtliche verfügbare Metriken aufgezeichnet werden:

```
profile(instances, calls)]
type T = ( n element N ) :
  f: () -> N :
          return n;
```

```
[profile(instructions)]
g: ( b element B ) -> T :
        t element T;
                [profile]
                if b = true :
                    t.n <- 1;
                else :
                    t.f();
                return t;
```

Neben der Untersuchung von Algorithmen ist die Beschreibung von Parallelität ein wichtiger Faktor in der Lehre. Gerade bei Algorithmen und Datenstrukturen kommt der Aspekt der Parallelisierung jedoch häufig zu kurz – trotz der steigenden Bedeutung durch die Mehrkernprozessoren. Die Lehre konzentriert sich dabei häufig auf abstrakte Konzepte oder komplexe Programmbibliotheken wie PVM oder MPI. Eine einfache, intuitiv verständliche und dabei dennoch umfangreiche Möglichkeit zur Beschreibung und vor allem auch Ausführung von Parallelität existiert bisher nicht. Daher erlaubt Pseudo die Deklaration von umfangreicher Parallelität mit Hilfe der Annotationen, ohne dabei auf die intuitive Verständlichkeit zu verzichten.
Mit Blick auf die gängigsten Formen von Parallelität werden in Pseudo synchrone und asynchrone Prozesse sowie Prozesse mit getrenntem und solche mit gemeinsamem Adressraum unterschieden. Die Angabe des gewünschten Modus der Parallelität wird über die Argumente der Annotation `parallel` erreicht; weitere Angaben zur grundlegenden Parallelität sind nicht nötig, da ein Mapping der benötigten Prozesse auf die verfügbaren Recheneinheiten automatisch ausgeführt wird. Um im Programmiermodell mit gemeinsamem Adressraum den konfliktfreien Informationsaustausch zu gewährleisten, stehen die Schlüsselwörter `lock` und `unlock` zum Sperren von Variablen zur Verfügung. Zusätzlich werden Prozesse blockiert, die eine bereits gesperrte Variable mit einer weiteren Sperre belegen wollen; die Blockierung wird erst aufgehoben, wenn der erste sperrende Prozess seine Sperre aufhebt (Deadlock-Situationen müssen also immer noch durch den Programmierer vermieden werden). Die Prozesskommunikation im verteilten Adressraum erfolgt dagegen über die Schlüsselwörter `send`, `receive` und `scatter`; die Identifizierung bestimmter Prozesse erfolgt hierbei über die automatisch durch den Compiler global definierten Variablen `processID`, `processIDs` und `activeProcesses`.
Die `parallel`-Annotation kann im Zusammenhang mit bedingten Verzweigungen und der Blockbildung benutzt werden, wobei hierbei jede Anweisung im der Annotation untergeordneten Block in einem separaten Prozess ausgeführt wird. Der folgende Code zeigt zwei kurze Beispiele:

```
parallelFunction: () -> () :
     a = ( i, j ) element T^2;
     mainProcess <- processID element ProcessID;

     // 1: synchron, gemeinsamer Adressraum
     [parallel]
          i <- f();
          j <- g();

     // 2: synchron, getrennter Adressraum
     [parallel(distinct)]
          if processID = processIDs.last()[0] :
               i <- f();
               receive j from processIDs.last()[1];

          if processID = processIDs.last()[1] :
               send g() to processIDs.last()[0];
```

Neben Blöcken und bedingten Anweisungen lassen sich auch Iteratorschleifen parallelisieren, wobei für jedes Element der Quelldatenmenge der Schleifencode in einem separaten Prozess ausgeführt wird. Durch die Angabe eines scatter-Argumentes können zusätzlich angegebene weitere Datenmengen auf die verschiedenen Prozesse aufgeteilt werden, um so eine gleichmäßige Bearbeitung größerer Datenmengen durch mehrere Prozesse zu erlauben.

Didaktischer Wert und Fazit

Welcher didaktische Mehrwert entsteht nun daraus, dass in ansprechender, knapper und präziser Form beschriebene Algorithmen direkt mit Beispielen ausführbar sind? Anhand weniger Lehr-/Lernszenarien soll dies kurz illustriert werden.
In der reinen Präsenzlehre bietet der ausführbare Pseudo-Code eine höhere Anschaulichkeit, da mit kleinen Beispielen Details wie das Umhängen von Zeigern bei einer Baumrotation oder die Berechnungsreihenfolge bei den optimalen Suchbäumen als direkter Effekt des langsam ablaufenden Algorithmus erlebbar werden. Selbst mit reinen Textausgaben von Zwischenständen wird der Ablauf des abstrakt formulierten Algorithmus „erlebbar“ – wünschenswert ist zukünftig jedoch eine zusätzliche Visualisierungskomponente.
Für Übungsaufgaben bietet Pseudo alternative Aufgabenstellungen: Durch das Profiling von Algorithmen (z.B. Quicksort und Bottom-Up-Heapsort) oder Datenstrukturen (z.B. AVL- und Rot/Schwarz-Bäume) bzgl. der Laufzeit (oder der Anzahl der Schlüsselvergleiche oder Lese-/Schreibzugriffe) können Studenten direkt die Eigenschaften eines ablaufenden Algorithmus mit den Ergebnissen der asymptotischen Analyse vergleichen. Bisher war eine derartige Aufgabenstellung nur durch erheblichen, nicht mit einem Mehrwert für die Lehre einhergehenden zusätzlichen Programmieraufwand möglich. Das Experimentieren mit Algorithmen zielt auf die Kompetenz ab, die Techniken später in „echten“ Anwendungen einsetzen und verschiedene Ansätze im Praxiseinsatz gegeneinander abwägen zu können. Auch die Erstellung von Varianten einer Datenstruktur (z.B. bei B-Bäumen) kann leicht und spielerisch von Studenten bewältigt werden.
Darüber hinaus ermöglicht der Ansatz von Pseudo als vollständige Programmiersprache den leichten Einsatz und die Kombination von Datenstrukturen sowie Algorithmen; so können schnell kleine Anwendungsszenarien von den Studierenden durchgespielt wer-

den. Diese Synthese-Kompetenz in realistischen Problemstellungen ist bisher bei den Studierenden eher schwach ausgeprägt.
Ein weiterer wichtiger didaktischer Aspekt wird vor dem Hintergrund der zunehmenden Anzahl der Prozessorkerne deutlich – eine Herausforderung, der die heutigen Programmierer kaum gewachsen sind [PT08]. In Anbetracht dieses grundlegenden Wandels in der alltäglichen Softwareentwicklung bietet Pseudo mit seiner annotationsgesteuerten Parallelisierung einige einfache, aber grundlegende Konzepte für die Vermittlung von Parallelisierung im selben Namensraum – erneut mit der Möglichkeit, die Basistechniken durch Experimentieren selbst zu erfahren.
Den Autoren ist kein anderer Ansatz bekannt, der leichte Lesbarkeit (z.B. eines Pseudocodes) mit der direkten Ausführbarkeit (und einem Profiling) verbindet. Zur Beschreibung von Algorithmen geht am ehesten Knuths MIXAL auf einer maschinennahen Ebene einen ähnlichen Weg [K97]; auch eine Beschreibung von Algorithmen durch eine funktionale Sprache kann natürlich einige Aspekte ähnlich gut erfüllen.
Wo steht das Pseudo derzeit? Die Konzeption der Sprache ist abgeschlossen und es liegt eine prototypische Implementation vor. Für eine vollständige Entwicklungs- und Simulatorumgebung liegt derzeit das Hauptaugenmerk auf der Umsetzung des Profilings und der Parallelität sowie dem Export in eine mathematische Notation, da hier die großen Stärken von Pseudo liegen.
Hinsichtlich Nutzen und Akzeptanz liegen noch keine empirischen Daten vor, es ist jedoch geplant, die Sprache Pseudo im Sommersemester 2011 in größerem Umfang in einer Lehrveranstaltung des zweiten Fachsemesters einzusetzen. Wir werden dies durch eine entsprechende Befragung der Studierenden beziehungsweise gegebenenfalls auch durch eine Vergleichsstudie begleiten. Im Sommersemester 2010 wurden bereits einzelne Details der Pseudo-Notation ohne eine Werkzeugunterstützung an den Studierenden „erprobt“. Naturgemäß wird Pseudo durch Rückmeldungen beim Lehreinsatz Anpassungen im Sprachdesign unterliegen. In diesem Zuge soll auch geprüft werden, ob andere Lehrinhalte, wie etwa Entwurfsmuster, einfach in Pseudo realisierbar sind. Für eine größere Verbreitung bei Lehrenden kann auch die Definition eigener Operatorsymbole und Syntaxstrukturen geprüft werden.

Literatur

[PT08] V. Pankratius, W. F. Tichy: Die Multicore-Revolution und ihre Bedeutung für die Softwareentwicklung, Objektspektrum 2008

[K97] D. E. Knuth: Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition), 3rd ed. Addison-Wesley Professional, 1997