Physikalische Dimensionen in Ada

Christoph Grein

ESG-Elektroniksystem- und Logistik-GmbH, D-81675 München

Zusammenfassung:

Ein Verfahren zum Rechnen mit dimensionsbehafteten Größen wird vorgestellt, das die dimensionelle Richtigkeit der Operationen zur Laufzeit überprüft. Obwohl es laufzeitaufwändig ist, ist seine Anwendung unter harten Echtzeitbedingungen nicht von vornherein ausgeschlossen, da sich die Dimensionen sehr einfach ausschalten lassen, so dass nur die reine Numerik übrig bleibt.

1. Einleitung

Wie jedem erfahrenen Ada-Programmierer bekannt sein sollte, eignet sich Ada nicht, physikalische Dimensionen mit seinem statischen Typkonzept nachzubilden, das mittels strenger Typbindung fehlerhafte Typmischung schon zur Übersetzungszeit erkennen kann. Obwohl dies auf den ersten Blick recht nahe liegend erscheint, ist das nicht so ganz einfach, wie schon die Beobachtung zeigt, dass zwar Meter plus Meter wieder Meter ergibt, Meter mal Meter jedoch Meter zum Quadrat, und dass Ada-Operatoren stets typtreu arbeiten (für `A`, `B` vom Typ `T` ist `A*B` ebenfalls vom Typ `T`) [1].

Trotzdem gibt es immer wieder Versuche dazu, die alle auf Überladung der Operatoren beruhen. Man geht dazu am besten von einer Dimensionsarithmetik der folgenden Gestalt aus, die nur solche Operatoren enthält, die die Dimension erhalten:

```
generic

  type Real is digits <>;

package Unit is

  type Item is private;

  function "+" (L, R: Item) return Item;
  function "-" (L, R: Item) return Item;

  function "/" (L, R: Item) return Real;

  function "*" (L: Real; R: Item)
    return Item;
  function "*" (L: Item; R: Real)
    return Item;
  function "/" (L: Item; R: Real)
    return Item;

  ...

private
  ...
end Unit;
```

Man erzeugt daraus die dimensionierten Typen durch Ausprägung:

```
package Dimension is new Unit (Float);
type Meter  is new Dimension.Item;
type Second is new Dimension.Item;
type Meter_per_Second is new Dimension.Item;
```

Hierzu werden dann alle Operatoren gemischter Dimensionen hinzugefügt:

```
function "/" (L: Meter; R: Second)
   return Meter_per_Second;
```

Der Vorteil eines solchen Verfahrens, könnte es denn funktionieren, liegt in der Dimensionsprüfung der Gleichungen zur Übersetzungszeit. Für nur zwei Einheiten wie *Meter* und *Sekunde* mit der daraus abgeleiteten Einheit *Meter pro Sekunde* mag das ja noch angehen, aber für drei Einheiten oder gar das volle SI-System mit seinen sieben Grundeinheiten *Meter*, *Kilogramm*, *Sekunde*, *Ampere*, *Kelvin*, *Candela* und *Mol* führt das zu einer kombinatorischen Explosion von Operatoren, die nicht mehr überschaubar ist. Und dann versagt die Methode trotz aller Mühen bei so einfachen Gleichungen wie

$$s = \frac{gt^2}{2} \Leftrightarrow t = \sqrt{\frac{2s}{g}}$$

ganz zu schweigen von der Schottky-Langmuir-Gleichung für die Stromdichte der Anodenemission

$$j = \frac{4}{9}\varepsilon_0\sqrt{\frac{2e_0}{m_0}}\frac{U^{\frac{3}{2}}}{d^2}$$

da Potenzen und Wurzeln mit diesem Verfahren nicht darstellbar sind. Zugegeben, t^2 lässt sich auch als `t*t` schreiben, und die Quadratwurzel kann als weitere Operation hinzugefügt werden, aber höhere Potenzen t^n und ihre Umkehrung $\sqrt[n]{}$ sind nicht mehr darstellbar.

Daher habe ich in der oben zitierten Arbeit ein Verfahren vorgestellt, das ohne solch unnütze Überladung auskommt, da es auf die Dimensionsüberprüfung zur Übersetzungszeit von Gleichungen im SI-System absieht. Es ermöglicht dadurch das Rechnen mit physikalischen Größen weitestgehend ohne Typumwandlungen, wobei die Vorteile der Typbindung in kritischen Fällen jedoch erhalten bleiben. Grundlegende mathematische Funktionen wie die Quadratwurzel und trigonometrische Funktionen werden dabei zu vordefinierten Operationen der numerischen Basistypen; Formeln wie $x = x_0 \cos\varphi$ mit Winkeln in Grad- und Bogenmaß sind ebenfalls dimensionsrichtig ohne

---

[1] Siehe hierzu:
Vom Urknall zum Universum, Ada Aktuell 1.1 (März 1993), Mitteilungen der Fachgruppe Ada-Deutschland in der Gesellschaft für Informatik; zu finden auch unter:
`<http://home.T-Online.de/home/Christ-Usch.Grein/Ada/Universum.html>`
Hierin ist auch eine Liste älterer Veröffentlichungen zu diesem Thema enthalten.

Typumwandlung darstellbar. Dieses Verfahren wird seit mehr als einem Jahrzehnt erfolgreich in mehreren Avionikprojekten unter harten Echtzeitbedingungen angewandt.

Echte Dimensionssicherheit lässt sich, wie durch obige Überlegungen klar geworden ist, nur durch Überprüfung zur Laufzeit erreichen, indem die Dimension einer Größe als Attribut mitgeführt wird. Ein solches Verfahren soll in den nächsten Abschnitten vorgestellt werden. Obwohl es laufzeitaufwändig ist, ist seine Anwendung auf Echtzeitsysteme nicht von vornherein ausgeschlossen, da sich die Dimensionen sehr einfach ausschalten lassen, so dass nur die reine Numerik übrigbleibt. Allerdings möchte ich vorausschicken, dass es nicht fehlerhafte Gleichungen sind, die die wahren Probleme in großen Softwaresystemen verursachen (die lassen sich auch durch Kodeinspektionen aufspüren); deren Ursachen liegen gewöhnlich viel früher verborgen, nämlich in fehlerhaftem System- oder Softwareentwurf, wie einige spektakuläre Fehlschläge gezeigt haben.

Das Problem des *Mars Lander*s lag in der fehlerhaften Mischung von metrischen mit britischen Einheiten bei der Kommunikation zweier CSCIs, die keines dieser Verfahren hätte verhindern können, da sie stets nur innerhalb eines CSCI wirken (CSCI: Computer Software Configuration Item – ein allein stehendes Programm, das mit anderen Programmen über Datenbusse oder globale Speicherbereiche kommuniziert).

Ariane 5 scheiterte an der Wiederverwendung von Software für Ariane 4 ohne erneute Analyse des Verhaltens im Fehlerfall. Häufig wird dieser Fehlschlag dazu verwendet, Ada madig zu machen. In Wahrheit hätte keine Sprache diesen Fehlschlag verhindern können, da der Fehler nicht in der Programmierung lag, sondern in der *fehlenden* Systemanalyse für Ariane 5. Die Ausnahme, die bei Ariane 5 zum Absturz führte, konnte bei Ariane 4 nur durch fehlerhafte Hardware ausgelöst werden; die Fehlerbehandlung in der Software (Abschaltung der Komponente mit der fehlerhaften Hardware) war richtig für Ariane 4. Ariane 5 hatte jedoch ein ganz anderes Flugprofil, so dass dieselbe Ausnahme auch unter normalen Flugbedingungen auftreten konnte, die Fehlerbehandlung somit unpassend war. Dies war nicht erkannt worden, da eine erneute Systemanalyse dieser Softwarekomponente aus Kostengründen nicht durchgeführt worden war – mit den bekannten Folgen.

Es sei noch auf die theoretische Physik hingewiesen, die bei numerischen Berechnungen gewöhnlich ganz auf Dimensionsprüfung verzichtet, da sie die dimensionslose Form der Gleichungen vorzieht.

2. Vorüberlegungen

Wie die Einleitung gezeigt hat, müssen die Dimensionen als Attribute der physikalischen Größen mitgeschleppt werden. Dabei genügt es nicht, nur ganzzahlige Exponenten zu betrachten, zum Wurzelziehen brauchen wir rationale Zahlen. Die natürliche Verwendung rationaler Zahlen erfordert dann aber auch gemischte Zahlen, da die Umwandlung von `n+a/b` in `(a+nb)/b` gekünstelt wirkt.

Wir definieren das Paket `Rational_Arithmetics` mit allen Operationen für gemischte Zahlen. Der Operator

```
function "/" (L, R: Whole) return Rational;
```

dient uns dabei als Konstruktor. Die Auswahl der Operatoren muss mit großer Sorgfalt durchgeführt werden, damit mögliche Zweideutigkeiten bei Ausdrücken wie `1+3/(-7)` vermieden werden. Daher darf der Operator

```
function "/" (L, R: Whole) return Whole;
```

nicht verfügbar sein sowie einige weitere Operatoren mit ganzzahligem Ergebnis. Wir können die Sichtbarkeit der Operatoren wählen, indem wir die ganzen Zahlen in einem gesonderten Paket `Invisible` unterbringen und nur die gewünschten Operatoren importieren. (Wie schon sein Name andeutet, darf das Paket `Invisible` nirgends sonst verwendet werden.)

```
with Invisible;

package Rational_Arithmetics is

  -- Ganze Zahlen

  subtype Whole is Invisible.Whole;

  function "=" (L, R: Whole) return Boolean
    renames Invisible."=";
  function "+" (L, R: Whole) return Whole
    renames Invisible."+";
  ...
  function "<" (L, R: Whole) return Boolean
    renames Invisible. "<";
  ...

  -- Rationale Zahlen

  type Rational is private;

  -- Konstruktoren

  function "+" (   R: Whole) return Rational;
  function "/" (L, R: Whole) return Rational;

  -- Rationale Operatoren

  function "+" (L, R: Rational)
    return Rational;
  function "-" (L, R: Rational)
    return Rational;
  function "*" (L, R: Rational)
    return Rational;
  function "/" (L, R: Rational)
    return Rational;
  ...
  function "<" (L, R: Rational)
    return Boolean;
  ...

  -- Gemischte Operatoren

  ...

private
  ...
end Rational_Arithmetics;
```

3. Das Paket `Unconstrained_Checked_SI`

Wie wollen wir nun dimensionierte Größen darstellen? Nahe liegend wäre die Darstellung der Dimension als Diskriminante ohne Vorbelegungswerte

```
type Item
      (kg, m, s, A, K, cd, mol: Rational) is
  private;
```

da auf diese Weise bei jeder Variablenvereinbarung die Dimension angegeben werden müsste:

```
Speed: Item (m => +1, s => -1,
             others => +0);
```

Dies scheitert allerdings an der Spracheinschränkung, dass Diskriminanten diskret sein müssen.
Daher verzichten wir zunächst auf die Diskriminanten und damit auf Angabe der Dimension bei der Deklaration und definieren eine dimensionierte Größe als zusammengesetzt aus einer Dimensionskomponente und ihrem Zahlenwert (dies entspricht einem Verbund mit Vorbelegungswerten der Diskriminanten):

```
type Dimension is record
  m, kg, s, A, K, cd, mol: Rational;
end record;

type Item is record
  Unit : Dimension;
  Value: Float;
end record;
```

Eine weitere Forderung ist, dass sich numerische Werte möglichst natürlich darstellen lassen, also z. B. ähnlich wie v=`5m/s`. Dies führt zu folgender Definition des Pakets `Unconstrained_Checked_SI`.

```
with Rational_Arithmetics;
use  Rational_Arithmetics;

generic

  type Real is digits <>;

package Unconstrained_Checked_SI is

  type Item is private;

  -- Zugriff auf den reinen Zahlenwert

  function Value (X: Item) return Real;

  -- Basiseinheiten

  Radian: constant Item;
  Meter : constant Item;
  Second: constant Item;
  ...

  -- Operatoren

  function "+" (R: Item) return Item;
  function "-" (R: Item) return Item;

  function "+" (L, R: Item) return Item;
  function "-" (L, R: Item) return Item;

  function "*" (L, R: Item) return Item;
  function "/" (L, R: Item) return Item;

  function "*" (L: Item; R: Real)
               return Item;
  function "*" (L: Real; R: Item)
               return Item;
  function "/" (L: Item; R: Real)
               return Item;
  function "/" (L: Real; R: Item)
               return Item;

  function "**" (Base: Item; Exp: Whole   )
                return Item;
  function "**" (Base: Item; Exp: Rational)
                return Item;
  function "**" (Base: Item; Exp: Real    )
                return Item;

  -- Mathematik

  function Sqrt (X: Item) return Item;

  function Log (X: Item) return Item;
  function Exp (X: Item) return Item;

  function Sin (X: Item) return Item;
  ...

  Unit_Error: exception;

private

  type Dimension is record
    m, kg, s, A, K, cd, mol: Rational;
  end record;

  type Item is record
    Unit : Dimension;
    Value: Real;
  end record;

  Radian: constant Item :=
            (Unit  => (others => +0),
             Value => 1.0);
  Meter : constant Item :=
            (Unit  => (m => +1,
                       others => +0),
             Value => 1.0);
  Second: constant Item :=
            (Unit  => (s => +1,
                       others => +0),
             Value => 1.0);
  ...

end Unconstrained_Checked_SI;
```

In den Aggregaten zur Definition der Einheitskonstanten dient der unäre Plusoperator als Konstruktor der rationalen Zahlen 0 und 1.

Hiermit können wir ganz natürlich schreiben

```
declare
  Weg, Zeit:  Item;
  g: constant Item :=
       9.81 * Meter / Second ** 2;
begin
  Weg  := 10.0 * Meter;
  Zeit := Sqrt (2.0 * Weg / g);
end;
```

und können sicher sein, dass die innere Konsistenz der Dimensionen erhalten wird. Fehlerhafte Additionen wie `Weg+Zeit` führen zur Auslösung der Ausnahme `Unit_Error`. Leider haben wir jedoch keine Möglichkeit, fehlerhafte Zuweisungen wie

```
Länge := Sqrt (2.0 * Weg / g);
```

zu verhindern – `Unit_Error` wird hier nicht ausgelöst. Dies ließe sich nur erreichen, wenn die Deklaration von `Länge` die Dimension `Meter` einschlösse, was wir jedoch ausschließen mussten, da die Dimensionen keinen diskreten Wertebereich haben.

Neben den sieben Basiseinheiten gibt es noch eine Vielzahl von abgeleiteten Einheiten wie *Joule*, *Newton*, *Ohm* usw. Wir bringen sie in einem untergeordneten Paket (Ada nennt solche Pakete Kinder) `Unconstrained_Checked_SI.Generic_Units` unter, das natürlich wie sein Elter generisch sein muss:

```
generic package
  Unconstrained_Checked_SI.Generic_Units is

  Newton: constant Item :=
         Kilogram*Meter/Second**2;
  ...  -- weitere abgeleitete Einheiten

end Unconstrained_Checked_SI.Generic_Units;
```

Weitere (generische) Kinder sind denkbar: So kann `Unconstrained_Checked_SI.Generic_Text_IO` die Ein- und Ausgabe auf Dateien definieren, wobei sinnvollerweise die Dimension berücksichtigt wird, und `Unconstrained_Checked_SI.Generic_Constants` kann Naturkonstanten enthalten.

4. Das Paket `Constrained_Checked_SI`

Wie können wir das oben angesprochene Problem der fehlerhaften Zuweisung

```
Länge := Sqrt (2.0 * Weg / g);
```

lösen? Der Weg führt nur über Diskriminanten, und so müssen wir in den sauren Apfel beißen und für jede Dimension Zähler und Nenner trennen und einzeln als Diskriminante einführen! Entsetzlicher Gedanke… Wo bleibt die Einfachheit der Notation?
Nun, gar so schlimm ist es nicht – Ada lässt es zu, diese Hässlichkeit zu verbergen. Wir benennen das obige Paket um in `Constrained_Checked_SI` und müssen ansonsten im sichtbaren Teil nur die Deklaration von `Item` ein wenig ändern:

```
  type Item (<>) is private;
```

`(<>)` bezeichnet unbekannte, das sind für den Anwender unsichtbare, Diskriminanten. Mit ihnen ist er gezwungen, jede Deklaration mit einem Initialwert zu vervollständigen; der danach unveränderliche Wert der Diskriminanten wird aus dem Initialwert übernommen:

```
Länge: Item := Meter;

Länge := Sqrt (2.0 * Weg / g);
```

führt jetzt unweigerlich zu einer Ausnahme. (Der Ehrlichkeit halber muss allerdings angemerkt werden, dass

```
Länge: Item := Sqrt (2.0 * Weg / g);
```

immer noch möglich ist, `Länge` übernimmt hier die Dimension `Second` aus dem Initialausdruck.)

Die Vervollständigung der Deklarationen von `Item` im privaten Teil sieht folgendermaßen aus, wobei `x_N` für den Zähler, `x_D` für den Nenner der Dimension `x` steht, `x=x_N/x_D`:

```
private

  subtype Positive_Whole is Whole
            range 1 .. Whole'Last;

  type Item (m_N, kg_N, s_N, A_N, K_N,
             cd_N, mol_N: Whole;
             m_D, kg_D, s_D, A_D, K_D,
             cd_D, mol_D: Positive_Whole) is
    record
      Value: Real;
    end record;

  Meter: constant Item :=
     (m_N | m_D => +1,
      kg_N => +0, kg_D => +1,
      ...,
      Value => 1.0);
```

Nach außen bleibt alles fast unverändert, das obige Beispiel sieht jetzt folgendermaßen aus:

```
declare
  Weg : Item := Meter;
  Zeit: Item := Sekunde;
  g: constant Item :=
       9.81 * Meter / Second ** 2;
begin
  Weg  := 10.0 * Meter;
  Zeit := Sqrt (2.0 * Weg / g);
end;
```

Ein wenig gewöhnungsbedürftig ist diese Notation ja schon, doch für sicherheitskritische Anwendungen wird häufig die Initialisierung jeder Variablen bei der Deklaration zwingend vorgeschrieben aus der Erfahrung heraus, dass uninitialisierte Variablen zu schwer zu entdeckenden Programmfehlern führen – und der ganze Sinn unserer Bemühungen ist ja gerade erhöhte Sicherheit.
Es muss allerdings gesagt werden, dass mit unbekannten Diskriminanten die Deklaration von Feldern unmöglich gemacht wird. Für Vektorarithmetik müssen wir uns etwas Spezielles ausdenken. Dies soll hier nicht weiter ausgeführt werden.

```
type Vector is                -- unmöglich
     array (1 .. 3) of Item;
```

Die Anwendung sei kurz an einem Beispiel erläutert.

```
with Constrained_Checked_SI;
package SI is
        new Constrained_Checked_SI (Float);

with Constrained_Checked_SI.Generic_Units;
package SI.Units is new SI.Generic_Units;

with SI.Units;
use  SI, SI.Units;
procedure Test_SI is
  g: constant Item :=
       9.81 * Meter / Second**2;
  Dist: Item := Meter;
  Time: Item := Second;
  Charge: Item := 1.0 * Coulomb:
  Speed : Item := 5.0 * Meter / Second;
  Electric_Field: Item := 10.0 * Volt /
                               Meter;
  Magnetic_Field: Item := 1.0E-3 * Tesla;
  Force: Item := Newton;
begin
  Dist := 10.0 * Meter;
  Time := SQRT (2.0 * Dist / g);
  Force := Charge *
    (Electric_Field + Speed * Magnetic_Field);
end Test_SI;
```

5. Anwendung unter Echtzeitbedingungen

Da rationale Arithmetik sehr rechenzeitaufwändig ist, kann das oben vorgestellte Verfahren nicht ohne Abwandlung unter harten Echtzeitbedingungen angewendet werden. Wir gehen dazu von folgenden Überlegungen aus.

Ein Echtzeitsystem wird gewöhnlich auf einem so genannten Host-Rechner entwickelt, bevor es auf dem Target-Rechner zum Einsatz kommt. Auf dem Host werden auch die Unit-Tests durchgeführt. Dabei spielt das richtige Echtzeitverhalten keine Rolle; es kann auf Mehrpersonensystemen sowieso nicht funktionieren. Während der Entwicklung bis einschließlich zum Unit-Test wird somit das obige Verfahren verwendet.

Erst bei der Überführung auf das Target werden die Dimensionen ausgeschaltet. Dazu definieren wir ein weiteres Paket `Unchecked_SI`, das dieselbe sichtbare Spezifikation besitzt wie eines der oben vorgestellten Pakete, nur der private Teil verzichtet auf die Dimensionen, es bleibt nur der numerische Wert der Größen übrig; alle Konstanten wie `Meter` werden zum reinen Wert 1, sie können also leicht wegoptimiert werden. Im Paketrumpf kann daher die gesamte Dimensionsarithmetik entfernt werden, und damit entfällt natürlich auf dem Target die Dimensionsprüfung. Wird auf dem Host mit Unit-Tests 100%ige Testüberdeckung nachgewiesen, ist gleichzeitig auch bewiesen, dass alle Gleichungen dimensionsmäßig korrekt sind – die Laufzeitprüfung kann unbeschadet abgeschaltet werden.

```
with Rational_Arithmetics;
use  Rational_Arithmetics;

generic

  type Real is digits <>;

package Unchecked_SI is

  -- Der sichtbare Teil ist unverändert.
  -- Er enthält genau eine der beiden
  -- folgenden Vereinbarungen, je nachdem,
  -- welches Modell benutzt wird.
  type Item (<>) is private;
  type Item      is private;

  ...

private

  type Item is record
    Value: Real;
  end record;

  One: constant Item := (Value => 1.0);

  Radian: constant Item := One;
  Meter : constant Item := One;
  Second: constant Item := One;
  ...

end Unchecked_SI;
```

Bemerkenswert ist, dass die Vervollständigung von `Item` im privaten Teil in keinem Fall eine Diskriminante enthält. Trotzdem bleibt für den Anwender alles unverändert, Deklarationen ohne Initialwert sind weiterhin illegal, wenn im sichtbaren Teil eine Diskriminante spezifiziert ist. Die Implementierung der Kind-Pakete muss gegebenenfalls angepasst werden, insbesondere kann das Kindpaket `Unchecked_SI.Generic_Text_IO` keine Dimension mehr ausgeben und beim Einlesen muss die Dimensionsangabe schlicht übersprungen werden.

Wie gehen wir nun beim Übergang vom Host zum Target vor? Alles, was wir zu tun haben, ist, die Ausprägungen der Pakete `SI` und `SI.Units` von `Unconstrained_Checked_SI` beziehungsweise `Constrained_Checked_SI` auf `Unchecked_SI` umzustellen, da der sichtbare Teil beider Pakete gleich ist:

```
with Unchecked_SI;
package SI is new Unchecked_SI (Float);

with Unchecked_SI.Generic_Units;
package SI.Units is new SI.Generic_Units;
```

Der Rest des Programms (hier das Beispiel `Test_SI`) bleibt völlig unverändert, wie groß es auch immer sein mag.

Die vollständige Implementierung der Pakete `Unconstrained_Checked_SI`, `Constrained_Checked_SI` und `Unchecked_SI` samt einiger Kinder für Ein- und Ausgabe und Polynomarithmetik und Beispielprogramme dazu unterliegt der GMGPL (Gnat modified GNU General Public License) und kann von folgender Adresse heruntergeladen werden:
<http://home.T-Online.de/home/Christ-Usch.Grein/Ada/SI.html>