Towards a Technique of Incorporating Domain Knowledge for Unit Conversion in Scientific Reasoning Systems

Joseph Phillips

De Paul University
School of Computing and Digital Media
243 S. Wabash Ave
Chicago, IL 60604, USA
jphillips@cdm.depaul.edu

Abstract

Unit conversion is often considered a straightforward task using analytical knowledge like the definition of centimeters in terms of meters. However, conversions like the computation of a photon's frequency from its wavelength implicitly use domain knowledge. We present an update on ongoing work on how a scientific reasoning system may intelligently convert between units using domain knowledge *and* tag data thus produced as dependent upon this domain knowledge. This is part of a project to intelligently use meta-data for scientific value manipulation (Phillips 2010).

1. Introduction

Computers should be made to understand scientists, not other way around! Unfortunately scientists are far from uniform in their notation, even within a single domain.

A good example of this non-uniformity is with unit usage. True, the metric system is widely used. However even here there are some applications where meters/kilograms/seconds (giving energy units Joules) are preferred, and others where centimeters/grams/seconds (giving energy units dynes) are common.

Beyond this we see some applications where different units are used, even for the same dimension, to keep the system on a common or intuitive scale and/or so that values naturally fall between ranges 0.1 to 1.0, or 1.0 to 10. Examples include the agricultural rainfall or irrigation unit hectare-mm (as opposed to liters), the electrical energy unit kilowatt-hours (as opposed to Joules), and the astronomical unit parsecs, from *par*allax *sec*onds (as opposed to meters).

A third class of units actually changes dimensions. Domain knowledge is needed implicitly to convert between the dimensions. This is often seen with light, where a photon's wavenumbers (in inverse centimeters), wavelength (in meters, millimeters, microns, or nanometers), or its energy (in electron-volts) all may be taken as stand-ins for its frequency.

Relying on domain knowledge is particularly tricky because it can change. For example, since the development of special relativity we believe that the speed of light, the "c" in the equation $\nu=c/\lambda$ needed to convert from wavelength or wavenumbers to frequency, is constant for all observers in any reference frame. Prior to Einstein this would not have been a common belief.

Lastly, some units are actually for dimensionless values that have been normalized by being divided by some common standard. For example, the masses of planets in locations other than in orbit around our own Sun are commonly given in terms of how many "Jupiters" they are, rather than in kilograms. Also, the energy released in powerful events like large detonations is commonly given in kilotons of trinitrotoluene (TNT).

Herein we describe knowledge and algorithms to interconvert between pairs of units of the four cases given above. This work is a continuation and elaboration of on going work into writing a computer language which can reason about scientific meta-data intelligently (Phillips 2010).

The outline is as follows. We briefly discuss previous attempts to handle meta-data in the next section and follow with a brief description of the language we are building for scientific representation and reasoning. With this background we then present our approach and algorithms. We follow with a discussion of its limitations, and then conclude.

2. Prior Work

Several extensions exist to popular numeric and symbolic packages like *Mathematica* (Khanin 2001) and *Matlab* (deCarvalho 2006) enabling them to do dimensional analysis and unit conversion. Prior to that the Unix™ command *units* could do some multiplicative dimension and unit recognition, like converting 1000 cm^3 to 1 liter (SunOS 1992) (Mariano 2004). Also, systems that do scientific discovery like the *Bacon* series of programs are able to invent new units to describe new phenomena (Langley *et al* 1987). To the best of our knowledge, however, such systems do not tag the resulting calculations as dependent upon the correctness of the domain knowledge used to convert between dimensions (*e.g.* from wavenumbers to frequency), or are unable to do such dimension-changing conversions at all.

Fundamentally all these systems make the same mistake that Logical Positivists philosophers of science made when ignoring the extent to which data is theory-laden. They strive to carry forward only as much meta-data as needed to ensure the numeric or algebraic stability of the answer. They do not, however, even bother to capture, ask for, carry-forward, or exploit much domain knowledge[1].

3. A Brief Introduction to *StructProc*

StructProc is our language for a frame representation for scientific knowledge. Like other frame representations, *StructProc* knowledge is built around the <subject,attribute,value> triplet. *StructProc*, however, has a feature which may be less common. Numbers are not represented as being objects distinct from "symbols" (called "ideas" in StructProc) – rather numbers are represented as a special sort of idea. This allows numbers to be given properties (<attribute,value> pairs) like any other idea, for numbers to be queried on their properties like any other idea, *etc*.

The chief property to assign to numbers is their domain. A domain is an idea with corresponding dimensions, units, limits, *etc*. which is distinct from but obviously related to the attribute. For example, a building may be said to have a length, width and height. In general, all are distinct numbers corresponding to the attributes `lengthA`, `widthA`, and `heightA` (the postfixed `A` is *StructProc*'s recommended way to designate attributes). Though they are three different numbers, all have the same domain (`defaultMetersDomain`) with the unit being `meters`, dimension being `length`, and limitation being that values less than 0 are illegal.

Besides being annotated by their domains, numbers (and any other idea) may have subjects, attributes and assumptions specified. The *StructProc* expression

```
45.4{defaultMetersDomain,
      heightA,
      dePaulCenter,
      ^assumeS
      {measure1->angleOfElevationA =
        45.0{defaultDegreesDomain},
       measure1->distanceA =
        41.0 {defaultMetersDomain},
       measure1->heightA =
        1.4 {defaultMetersDomain}
      }
```

[1] With the exception of perhaps boundary conditions for differential equations and simple notions of units and dimensions.

```
      }
```

represents the number 45.4 which has been annotated as being the height of the building the DePaul Center in meters. Further, this value assumes the validity of three other measurements gathered under frame `measure1`: that 44.0 meters away from the building its top was sighted at an angle of 45.0 degrees above the horizon, and that the angle was measured from 1.4 meters above the ground.

4. Our Approach

The basic algorithm that underpins our approach is a straightforward conversion from one of unit to another of the same "fundamental" dimension as detailed in `sameSingleDimensionalConvert()`. Because both units share the same dimension, and because the compatibility of their subjects and attributes is checked elsewhere, no other checks need to be done or knowledge needs to be assumed.

The function `sameSingleDimensionalConvert()` returns linear conversion expression that converts from a value in `fromUnits` to one in `toUnits`. It does this by attempting to convert to the primary units of the "fundamental" dimension as an intermediate step. If it successfully finds a conversion then it saves it so it can be easily applied next time. (The expression `thisExpr.sub(expr)` takes the algebraic expressions `expr(`x_1`)` and `thisExpr(`x_2`)` and returns the new expression `thisExpr(expr(`x_1`))`. The expression `subject.get(attrA)` returns the first found value of `subject`'s attribute `attrA`, or returns `null` if none are found. The expression `covertToA(units)` builds an attribute that represents a conversion to the specified `units`.)

This function allows the conversion from parsecs to kilometers by way of meters, and from Celsius to Kelvin with a slope of 1 and intercept of 273.15.

```
LinearExpression
  sameSingleDimensionalConvert
          (Units fromUnits, Units toUnits)
begin
LinearExpression expr, thisExpr;
Dimension dim;
Units     prime;

if  (fromUnits == toUnits)
  Number slope     := 1;
  Number intercept := 0;
  return new LinearExpression(slope,intercept);
endif

expr := store.retrieve(fromUnits,toUnits)

if  (expr != null)
  return(expr);
```

```
endif

dim   := fromUnits.get(dimensionA);
prime := dim.get(primaryUnitsA);
expr  := fromUnits.get(covertToA(prime))

if (expr == null)
  throw new InsufficientInformation();
endIf

thisExpr := prime.get(covertToA(toUnits));

if (thisExpr == null)
  throw new InsufficientInformation();
endIf

expr := thisExpr.sub(expr);
store.save(fromUnits,toUnits,expr);
return expr;
end;
```

Two functions are used to convert between multi-dimensional units. The function `incorporateUnits()` is given a list of dimension entries (`dimList`). Each entry tells a dimension and itself has a list of basic units of that dimension and the powers to which those units have been raised. The function adds to this list the new dimensions and basic units it finds for the `unit` parameter it has been given. The basic units of `unit` are incorporated into `dimList`, but their powers are multiplied by `sign`: either +1 or -1. This allows basic units that appear in both a `fromUnits` (line 2 with sign -1) and `toUnits` (line 3 with sign +1) to cancel each other at line 1.

```
DimList
  incorporateUnits
    (DimList dimList, Units unit, int sign)
begin
Units     basicU;
int       power;
DimEntry  dimEntry;
UnitEntry unitEntry;
Dimension dimen;

forall basic unit pairs (basicU,power) in unit do
  dimen     := basicU.get(dimension);
  dimEntry  := dimList.find(dimen);

  if (dimEntry == null)
    dimEntry := new DimEntry(dimen);
    dimList.prepend(dimEntry);
  endif

  unitEntry := dimEntry.unitList.find(basicU);
  if  (unitEntry == null)
    unitEntry := new UnitEntry(basicU);
    dimEntry.unitList.prepend(unitEntry);
  endif;

  unitEntry.addPower(power*sign); // Line 1
endfor

return(dimList);
end;
```

The function `sameMultiDimensionalConvert()` uses both `incorporateUnits()` and `sameSingleDimensionalConvert()` to return a conversion expression from a value in `fromUnits` to one in `toUnits`. It uses the former function to decompose (and hopefully partially cancel) both incoming units into their basic units, isolated by their dimensions (lines 2 and 3).

For each dimension it does the following. It finds the next occurrences of both a basic unit raised to a positive power (loop 4) and raised to a negative power (loop 5). They are different units of the same dimension, so they should be converted. If there are no more units to convert then it goes on to the next dimension (line 6).

When it finds units to convert it uses `sameSingleDimensionalConvert()`. Allowances are made when the units are raised to higher powers (lines 7, 8), those units are cancelled (lines 9, 10), and then it continues looking more units to convert. It also checks the expression returned from `sameSingleDimensionalConvert()` (not shown) to see if it has an added constant other than zero. It must do so because it builds an expression of multiplied (and divided terms): added terms would throw-off the computation. Also, they probably signify that the wrong units were used in the expression. For example, the Maxwell-Boltzmann distribution equation uses temperature in Kelvin (with an absolute 0) instead of in Celsius (with the additive term 273.15).

At the very end it returns an algebraic expression that uses the product of the slopes of all conversions.

```
Expression
  sameMultiDimensionalConvert
          (Units fromUnits, Units toUnits)
begin
DimList   dimList := null;
double    product := 1.0;
DimEntry  dimEntry;
UnitEntry posEntry, negEntry;
int       power;
Expression expr;

// Line 2
dimList:= incorporateUnits(dimList,fromUnits,-1);
// Line 3
dimList:= incorporateUnits(dimList,  toUnits,+1);

forall dimEntry in dimList do
  posEntry := negEntry := dimEntry.unitList.head;
  while (true)
    // Loop 4
    while (posEntry!=null AND posEntry.power<=0)
      posEntry := posEntry.next;
    endwhile

    // Loop 5
    while (negEntry!=null AND negEntry.power>=0)
      negEntry := negEntry.next;
    endwhile

    if (posEntry == negEntry == null)   // Line 6
      break;
    endif

    expr  := sameSingleDimensionalConvert
                (negEntry.unit,posEntry.unit);
    // Line 7
    power := min(posEntry.power,-negEntry.power);

    for (i := 0; i < power;  i := i+1)
        product := product * expr.slope; //Line 8
    endfor

    posEntry.addPower(-power); // Line 9
    negEntry.addPower(+power); // Line 10
  endwhile
endfor

expr := new LinearExpression(product,0);
store.save(fromUnits,toUnits,expr);
return(expr);
end;
```

This function allows the conversion from hectare-mm to liters by restating hectares as being hundred-meters squared, restating liters as decimeters cubed, and converting millimeters to decimeters (once) and hundred-meters to decimeters (twice). The second conversion from hundred-meters to decimeters benefits from the cached results generated by the first conversion.

Similarly, it can also convert kilowatt-hours to Joules. This particular conversion highlights the generality of the algorithm. Our system knows kilowatt-hours as:

$$\frac{\text{(kilograms)*(meters)*(kilometers)*(hours)}}{\text{(seconds}^3)}$$

The kilograms, meters, kilometers and inverse cubed seconds terms multiply to give kilowatts, and multiplying that by hours gives kilowatt-hours.

This is a natural way to define kilowatt-hours but it represents the time dimension in an atypical fashion as hours/seconds^3. Hours and seconds do not need to be converted between `fromUnits` and `toUnits` but within `fromUnits` itself.

However, because our algorithm builds just one list (`dimEntry.unitList`) of units to convert for each dimension, whether the units to convert are between `fromUnits` and `toUnits` or within either makes no difference.

Conversion between units of different dimensions necessarily uses domain knowledge. This knowledge includes the expression used to tie the two dimensions (and thus implicitly any knowledge on which the stating of that expression depends). This expression has some limited scope for which it holds, thus the search for a dimension converting expression starts with an ontological search from a most specific (among the smallest) ontological sets that encompass the subjects of both values and systematically considers increasingly broader sets.

At each set it considers all stated dimension conversions looking for one that converts from the from attribute to the to attribute. Unlike the algorithms for `sameSingleDimensionalConvert()` and `sameMultiDimensionalConvert()` which just used the units being converted, we now must use the attributes because we are trying to convert between different but specific aspects of the subject objects. Further, we assume the attributes imply specific dimensions.

Upon finding a candidate dimensional conversion we use `sameMultiDimensionalConvert()` to see if we can convert from the given `fromUnits` to the units expected by the expression, and from the units returned by the expression to the given `toUnits`. If we find such an expression and its required auxiliary conversions then we return that expression in which both auxiliary conversions have been substituted. We throw an exception otherwise. This algorithm is given as `differentDimensionalConvert()`.

While `differentDimensionalConvert()` can use any expression for which it is clever enough to algebraically manipulate we anticipate many of its expressions will be simple linear or inverse linear 1/x functions. For example, conversion from either a photon's wavenumber in inverse centimeters (as is common in infra-red spectroscopy) or its energy in electron volts (as is common in material science) to its frequency in Hertz (inverse seconds) would be built around a simple linear dimension conversion in which the from units would either be pre-converted to inverse meters (from cm^{-1}) or to joules (from eV). Conversion from wavenumbers would tag the result as depending on the assumptions related to $\nu=c/\lambda$, including those related to the speed of light c. Conversion from energy would tag the result as depending on assumptions related to $\nu=E/h$, including those related to Planck's constant h. Conversion from a photon's wavelength to its frequency would be very similar to its conversion from wavenumbers, except that it would use an inverse linear dimension conversion.

```
<Expression,AssumptionSet>
    differentDimensionalConvert
```

```
       (Subject fromSubj,
        Attribute fromAttr,
        Units fromUnits,
        Subject toSubj,
        Attribute toAttr,
        Units toUnits)
begin
Conversion convert;
Units      givenFromUnits, givenToUnits;
Expression expr, exprTo, exprFrom;
Set s := mostSpecificCommonSet(fromSubj,toSubj);

while  (s != null)
  for all convert := s.get(diffDimConvA
                            (fromAttr,toAttr)
                          ) do
    givenFromUnits := convert.get(fromUnitsA);
    givenToUnits   := convert.get(toUnitsA);
    expr := convert.get(exprA);

    try
      exprFrom :=
        sameMultiDimensionalConvert
            (fromUnits,givenFromUnits);
      exprTo :=
        sameMultiDimensionalConvert
            (givenToUnits,toUnits);
    catch InsufficientInformation
      continue;
    endCatch

    return <exprTo.sub(expr.sub(exprFrom)),
             expr.get(assumptionsA)>
  endFor
  s := s.get(nextMoreEncompassingSetA)
endWhile

throw new insufficientInformation();
end;
```

The algorithm `differentDimensionalConvert()` also lets us handle dimensionless values that have been normalized by some empirical standard. For example, since the mid-1990s astronomers have found about 500 or so "exo-planets", planets in places other than in orbit around our Sun. One way to detect such planets is by looking for its gravitational tug on the star which they orbit, and the more massive the planet the larger the tug, thus many of the planets that have been found are massive. To keep the mass numbers in intuitive ranges rather than as "so many kilograms times 10 to the such-and-such power" it is common to express them as multiples of the mass of our own giant, Jupiter.

Conversion from the dimensionless unit M_J (how many "Jupiters") to the conventional mass unit kilograms can be done with `differentDimensionalConvert()` by giving the knowledge base a simple linear dimension conversion rule telling it to convert from attributes with dimensionless values but normalized units to attributes for domains with units by multiplying by the normalization factor (in this case the mass of Jupiter: $1.8986\text{x}10^{27}$ kg (Williams 2010)). Although this rule is analytically true and thus makes no assumptions, many assumptions probably went in to the normalization factor. Thus the routine for `expr.get(assumptionsA)` must be clever enough to gather the assumptions of normalization factor too. Additionally, because assumptions are cumulative any assumption that went into why, for example, we believe planet μAra b to be 1.68 M_J (Butler *et al* 2001), such as our estimate of the mass of its star μAra, this assumption would also be included in the resulting value.

This one rule covers a variety of normalizations. For example, consider the domain of powerful events, especially nuclear detonations, with its common unit "kilotons of trinitrotoluene (kT)". By stating that 1 kiloton of TNT is $4.184*10^{12}$ joules, our system can convert such values.

6. Discussion

We have presented the first automated approach, to the best of our knowledge, that both tags values with metadata describing the theory and measurements upon which their computation depend, and that carries this metadata forward for the computation of subsequent values.

At least two outstanding issues remain including handling data that contradict in terms of accuracy, and how to handle conversions where the domain knowledge tells us to consider three or more attributes.

First, what should we do if one value assumes Jupiter's mass is $1.898\underline{\mathit{6}}\text{x}10^{27}$ kg while a later one assumes it is $1.898\underline{\mathit{57}}\text{x}10^{27}$ kg? One could convert between the two by dividing out what one considers the "less" accurate value and multiplying with what considers the "more" accurate. This is arithmetically sound, but what of the assumptions that went into both kilogram figures for the mass of Jupiter? They may be contradictory in a deeper manner than "same formula with more precise numbers used" by, for example, considering secondary effects (*e.g.* relativity) or by being derived from a different formula altogether. In such cases one could dig deeper to look for potential contradictions, or take a precautionary stand and throw a `PotentiallyContradictoryAssumption` exception.

A second problem occurs when we handle the special relativity equation $E=mc^2$. No dimension conversion is necessary because both sides have the dimension "energy". However, the set applicability of `differentDimensionalConvert()` is still needed because only in certain circumstances like matter-antimatter annihilation do we observe mass to energy interconversion.

Even if we add this knowledge and its necessary restrictions we still have handled only a special case of matter-antimatter annihilation of particles *at rest*. If particles have significant speed then their kinetic energy also should be considered. This would necessitate revising `differentDimensionalConvert()` or writing a new function to handle multiple attributes (*e.g.* mass, rest energy, and relative speed by, for example, the Lorentz transformations).

7. Conclusion

We have presented algorithms and knowledge structures needed to safely handle interconversion among four types of units in common usage in the sciences. Further, we have discussed its limitations.

It would be a mistake to think the solution to the second issue as merely extending an algorithm over more attributes. Fundamentally we should give our system the ability to (re-)define its own dimensions as needed. (The *StructProc* knowledgebase is being built with an eye towards this. This is the reason why we had the word "fundamental" in quotes when describing dimensions.) Thus, it should be able to define the Lorentz transformations, not just apply them. Such searches for both accurate and at least somewhat intuitive definitions of time, space, *etc*. are at the heart of modern physics quest to unify quantum mechanics with relativity (Callender 2010, Musser 2011).

Artificial intelligence may play a role this search through the space of representations. If it does we must be clear about what our systems actually represent and symbolically manipulate. This paper attempts to do so for a limited domain.

References

Butler, R. Paul; Tinney, C. G.; Marcy, Geoffrey W.; Jones, Hugh R. A.; Penny, Alan J.; Apps, Kevin. 2001. "Two new planets from the Anglo-Australian planet search" *Astrophysical Journal*. 555 : 410-417, 2001 July 1.

Callender, Craig. 2010. "Is Time an Illusion?" *Scientific American*. 2010 June.

deCarvalho, Rob. 2006. "Simple Units and Dimensions for Matlab"http://www.mathworks.com/matlabcentral/fileexch ange /9873. Originally appeared Feb 2, updated Mar 3.

Khanin, Raya. 2001. "Dimensional analysis in computer algebra." *International Symposium on Symbolic and Algebraic Computation*. ACM. 2001.

Langley, Pat. Simon, Herbert. Bradshaw, Gary. Zytkow, Jan. *Scientific Discovery: Computational Explorations of the Creative Processes*. MIT Press. Cambridge, MA. 1987.

Mariano, Adrian. "Manual page for GNU Units version 1.85" 2004.

Musser, George. 2011. "Forces to Reckon With: Does gravity muck up electromagnetism?" *Scientific American*. 2011 February.

Phillips, Joseph. 2010. "A Proposed Semantics for the Sampled Values and Metadata of Scientific Values." *Midwest Artificial Intelligence and Cognitive Science Conference*.

SunOS "Manual page for SunOS' `units` command" 1992 Sep 14.

Williams, David R. 2010. "Jupiter Fact Sheet." http://nssdc.nasa.gov/planetary/factsheet/jupiterfact.html. Last updated 2010 November 17.