Dimension Types *

Andrew Kennedy

University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
United Kingdom
Andrew.Kennedy@cl.cam.ac.uk

Abstract. Scientists and engineers must ensure that physical equations are dimensionally consistent, but existing programming languages treat all numeric values as dimensionless. This paper extends a strongly-typed programming language with a notion of dimension type. Our approach improves on previous proposals in that dimension types may be polymorphic. Furthermore, any expression which is typable in the system has a most general type, and we describe an algorithm which infers this type automatically. The algorithm exploits equational unification over Abelian groups in addition to ordinary term unification. An implementation of the type system is described, extending the ML Kit compiler. Finally, we discuss the problem of obtaining a canonical form for principal types and sketch some more powerful systems which use dependent and higher-order polymorphic types.

1 Introduction

One aim behind strongly-typed languages is the detection of common programming errors before run-time. Types act as a constraint on the range of allowable expressions and stop 'impossibilities' happening when a program is run, such as the addition of an integer and a string.

In a similar way, scientists and engineers know that an equation cannot be correct if constraints on *dimensions* are broken. One can never add or subtract two values of differing dimension, and the multiplication or division of two values results in values whose dimensions are also multiplied or divided. Thus the sum of values with dimensions speed and time is a dimension error, whereas their product has dimension distance.

The addition of dimensions to a programming language has been suggested many times [KL78, Hou83, Geh85, Män86, DMM86, Bal87]. Some of this work is seriously flawed and most systems severely restrict the kind of programs that can be written. House's extension to Pascal is much better [Hou83]. In a monomorphic language it allows functions to be polymorphic over the dimension of arguments. Since the submission of this paper an anonymous referee has pointed out work by Wand and O'Keefe on dimensional inference in the style of ML type inference [WO91]. In some ways this is similar to the approach taken here and a comparison with their system is presented later in this paper.

* Appeared in European Symposium on Programming '94, LNCS Volume 788

2 Some issues

2.1 Dimension, Unit and Representation

There is often confusion between the concepts of *dimension* and *unit* [Man87]. Two quantities with the same *dimension* describe the same kind of property, be it length, mass, force, or whatever. Two quantities with different *units* but the same dimension differ only by a scaling factor. A value measured in inches is 12 times the same value measured in feet—but both have the dimension *length*. We say that the two units are *commensurate* [KL78, DMM86]. These units have simple scaling conversions. More complicated are units such as temperature measured in degrees Celsius or Fahrenheit, and even worse, amplitude level in decibels.

Base dimensions are those which cannot be defined in terms of other dimensions. The International System of Units (SI) defines seven of these—length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity. *Derived* dimensions are defined in terms of existing dimensions, for example, acceleration is distance divided by time squared. Dimensions are conventionally written in an algebraic form inside square brackets [Lan51], so for example the dimensions of force are written $[\mathrm{MLT}^{-2}]$.

Similarly there are base units—the SI base dimensions just listed have respective units metres, kilograms, seconds, Amperes, Kelvin, moles and Candela. Examples of derived units include inches (0.0254 metres) and newtons (kgms^{-2}).

There is also the issue of *representation*: which numeric *type* is used to store the numeric value of the quantity in question. Electrical quantities are often represented using complex numbers, whereas for distance reals are more common, and for either of these many languages provide more than one level of precision.

Dimensionless quantities are common in science. Examples include refractive index, coefficient of restitution, angle and solid angle. The last two should properly be considered dimensionless though it is tempting to think otherwise. An angle is the ratio of two lengths (distance along an arc divided by the radius) and a solid angle is the ratio of two areas (surface area on a sphere divided by the square of the radius).

2.2 Types and Polymorphism

How do these concepts of dimension, unit, and representation fit with the conventional programming language notion of *type*?

Expressions in a strongly typed language must be *well-typed* to be acceptable to a compiler. In functional languages, for example, the rule for function application insists that an expression $e_1\, e_2$ has type τ_2 if e_1 has an arrow type of the form $\tau_1 \to \tau_2$ and the argument e_2 has type τ_1.

In a similar way, mathematical expressions must be *dimensionally consistent*. Expressions of the form $e_1 + e_2$ or $e_1 - e_2$ must have sub-expressions e_1 and e_2 of identical dimension. But in $e_1 e_2$ (product) the sub-expressions may have any dimension, say δ_1 and δ_2 giving a resultant dimension for the whole expression of $\delta_1 \delta_2$.

So it appears that dimensions can be treated as special kind of type in a programming context. But there is the question of what to do about representation. Do we associate particular dimensions with fixed numeric types (so current is always represented by a complex number, distance by a real), or do we parameterise numeric types on dimension and give the programmer the flexibility of choosing different representations for different quantities with the same dimension?

A *monomorphic* dimension type system is of limited value. For non-trivial programs we would like to write general-purpose functions which work over a range of dimensions. Even something as simple as a squaring function cannot be expressed in a monomorphic system. A modern polymorphic language would use quantified variables to express the idea that this function squares the dimension of its argument, for *any* dimension.

2.3 Type inference

Type systems such as that of Standard ML are designed so that the compiler can *infer* types if the programmer leaves them out. It turns out that this is possible for a dimension type system too.

A desirable property of inferred types is that they are the *most general* type, sometimes called *principal*. Any other valid typing can be obtained from this most general type by simple substitution for type variables. Our system does have this feature, and an algorithm is described which finds the principal type if one exists.

3 The idea

The system described here is in the spirit of ML [MTH89, Pau91]. It is *polymorphic*, so functions such as mean and variance can be coded to work over values of any dimension. The polymorphism is *implicit*—dimension variables are implicitly quantified in the same way as ML type variables. It is possible for the system to *infer* dimension types automatically, as well as check types which the programmer specifies.

Although it is described as an extension to ML, any language with an ML-like type system would suffice; indeed, it could even be added as an extension to a monomorphically-typed language as House did with Pascal. It is a conservative extension to ML in the sense that ML-typable programs remain typable, though functions may be given a more refined type than before.

We start with a set of *base* dimensions such as mass, length, and time, perhaps represented by the identifiers M, L and T as is conventional. Dimensions are written inside square brackets, for example [MLT^{-2}]. This notation cannot be confused with the ML list value shorthand, although some languages such as Haskell use $[\tau]$ to denote the list *type*.

For polymorphic dimensions we need dimension variables. We use $d_1, d_2, \ldots$ to distinguish them from ordinary type variables $\alpha, \beta, \ldots$. The unit dimension (for dimensionless quantities) is indicated by $[\mathbf{1}]$.

We assume some kind of construct for declaring base dimensions. This could be extended to provide derived dimensions; we do not discuss this possibility here. The provision of multiple units for a single dimension is also an easy extension to the system.

3.1 Dimension types

We introduce new numeric types parameterised on dimension. The most obvious candidates are `real` and `complex`, with speeds having type $[\mathrm{LT}^{-1}]\,\texttt{real}$ and electric current $[\mathrm{Current}]\,\texttt{complex}$. The parameter is written to the left of the type constructor in the style of Standard ML.

For the remainder of this paper we will only consider a single type constructor. In a type of the form $[\delta]\texttt{real}$, δ is a dimension expression which is completely separate from other type-forming expressions and which may only appear as a parameter to numeric types.

3.2 Arithmetic

We give the following type schemes to the standard arithmetic operations:

$$\begin{array}{ll}
+,- & : \forall d.\,[d]\,\texttt{real} \times [\mathtt{d}]\,\texttt{real} \to [\mathtt{d}]\,\texttt{real} \\
* & : \forall d_1 d_2.\,[d_1]\,\texttt{real} \times [\mathtt{d_2}]\,\texttt{real} \to [\mathtt{d_1 d_2}]\,\texttt{real} \\
/ & : \forall d_1 d_2.\,[d_1]\,\texttt{real} \times [\mathtt{d_2}]\,\texttt{real} \to [\mathtt{d_1 d_2^{-1}}]\,\texttt{real} \\
\texttt{sqrt} & : \forall d.\,[d^2]\,\texttt{real} \to [\mathtt{d}]\,\texttt{real} \\
\texttt{exp}, \texttt{ln}, \texttt{sin}, \texttt{cos}, \texttt{tan} & : [\mathbf{1}]\,\texttt{real} \to [\mathbf{1}]\,\texttt{real}
\end{array}$$

It is often useful to coerce an integer into a dimensionless real, for which we provide a suitable function:

$$\texttt{real} : \texttt{int} \to [\mathbf{1}]\,\texttt{real}$$

Finally, it turns out that we need a polymorphic zero:

$$\texttt{zero} : \forall d.\,[d]\,\texttt{real}$$

3.3 Some examples

Use of zero. Without a polymorphic zero value we would not even be able to test the sign of a number, for example, in an absolute value function:

```
fun abs x = if x < zero then zero-x else x
```

with type $\forall d.\,[d]\,\texttt{real} \to [\mathtt{d}]\,\texttt{real}$. It is also essential as an identity for addition in functions such as the following:

```
fun sum []      = zero
  | sum (x::xs) = x + sum xs;
```

This has the type scheme $\forall d.\,[d]\,\texttt{real list} \to [\mathtt{d}]\,\texttt{real list}$.

Statistical functions. Statistics provides a nice set of example functions because we would want to apply them over a large variety of differently dimensioned quantities. We list the code for mean and variance functions:

```
fun mean xs = sum xs / real (length xs);
fun variance xs =
  let val n = real (length xs)
      val m = mean xs
  in sum (map (fn x => sqr (x - m)) xs) / (n - real 1) end;
```

Their principal types, with those of some other statistical functions, are:

$$
\begin{array}{ll}
\texttt{mean} & : \forall d.\,[d]\,\texttt{real list} \to [\mathtt{d}]\,\texttt{real} \\
\texttt{variance} & : \forall d.\,[d]\,\texttt{real list} \to [\mathtt{d}^2]\,\texttt{real} \\
\texttt{sdeviation} & : \forall d.\,[d]\,\texttt{real list} \to [\mathtt{d}]\,\texttt{real} \\
\texttt{skewness} & : \forall d.\,[d]\,\texttt{real list} \to [\mathbf{1}]\,\texttt{real} \\
\texttt{correlation} & : \forall d_1 d_2.\,[d_1]\,\texttt{real list} \to [\mathtt{d}_2]\,\texttt{real list} \to [\mathbf{1}]\,\texttt{real}
\end{array}
$$

Differentiation. We can write a function which differentiates another function numerically. It accepts a function `f` as argument and returns a new function which is the differential of `f`. We must also provide an increment `h`.

```
fun diff h f = fn x => (f (x+h) - f (x-h)) / (real 2 * h)
```

This has type scheme

$$\forall d_1 d_2.\,[d_1]\,\texttt{real} \to ([\mathtt{d}_1]\,\texttt{real} \to [\mathtt{d}_2]\,\texttt{real}) \to ([\mathtt{d}_1]\,\texttt{real} \to [\mathtt{d}_2\mathtt{d}_1^{-1}]\,\texttt{real})$$

Unlike the statistical examples, the type of the result is related to the type of more than one argument.

Root finding. Here is a tiny implementation of the Newton-Raphson method for finding roots of equations:

```
fun newton (f, f', x, eps) =
  let val dx = f x / f' x
      val x' = x - dx
  in if abs dx < eps then x' else newton (f, f', x', eps) end;
```

It accepts a function `f`, its derivative `f'`, an initial guess `x` and an accuracy `eps`. Its type is

$$
\begin{array}{l}
\forall d_1 d_2.\,([d_1]\,\texttt{real} \to [\mathtt{d}_2]\,\texttt{real}) \times ([\mathtt{d}_1]\,\texttt{real} \to [\mathtt{d}_1^{-1}\mathtt{d}_2]\,\texttt{real}) \times \\
\qquad [d_1]\,\texttt{real} \times [\mathtt{d}_1]\,\texttt{real} \to [\mathtt{d}_1]\,\texttt{real}
\end{array}
$$

Powers. To illustrate a more unusual type, here is a function of three arguments.

```
fun f (x,y,z) = x*x + y*y*y + z*z*z*z*z
```

This has the inferred type scheme

$$\forall d.\,[d^{15}]\,\texttt{real} \times [\mathtt{d}^{10}]\,\texttt{real} \times [\mathtt{d}^{6}]\,\texttt{real} \to [\mathtt{d}^{30}]\,\texttt{real}$$

4 A dimension type system

We formalise the system by considering a very small ML-like language. Dimension expressions are defined by:

$$\delta \ ::= \ d \mid \mathrm{B} \mid \delta \cdot \delta \mid \delta^{-1} \mid \mathbf{1}$$

where B is any base dimension and d is any dimension variable. The shorthand d^n ($n \in \mathbb{N}$) will be used to stand for the n-fold product of d with itself, and occasionally we will write $d_1 d_2$ instead of $d_1 \cdot d_2$.

Now we define *monomorphic* type expressions by:

$$\tau \ ::= \ \alpha \mid [\delta]\,\mathtt{real} \mid \tau \to \tau$$

where α is any type variable. *Polymorphic* type expressions, also called *type schemes* are defined by

$$\sigma \ ::= \ \tau \mid \forall \alpha.\sigma \mid \forall d.\sigma$$

We have extended the usual ML-style type schemes with quantification over dimension variables, which must be distinct from type variables in order to distinguish the two kinds of quantification. The flavour of polymorphism used for dimension types is the same as ordinary ML-like polymorphism. This leads to the usual problems but does mean that inference is straightforward. We shall have more to say on this subject later.

Finally, expressions are defined by

$$e \ ::= \ x \mid n \mid e\,e \mid \lambda x.e \mid \mathsf{let}\; x = e \;\mathsf{in}\; e$$

where x is a variable and n is a real-valued constant such as 3.14. The full set of inference rules is now given, based on Cardelli [Car87]. Only two new rules are required—generalisation and specialisation for dimension quantification. A_x denotes the type assignment obtained from A by removing any typing statement for x.

$$\text{VAR} \ \frac{}{A \vdash x : \sigma} \ A(x) = \sigma \qquad\qquad \text{REAL} \ \frac{}{A \vdash n : [\mathbf{1}]\,\mathtt{real}}$$

$$\text{GEN} \ \frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha.\sigma} \ \alpha \text{ not free in } A \qquad\qquad \text{SPEC} \ \frac{A \vdash e : \forall \alpha.\sigma}{A \vdash e : \sigma[\tau/\alpha]}$$

$$\text{DGEN} \ \frac{A \vdash e : \sigma}{A \vdash e : \forall d.\sigma} \ d \text{ not free in } A \qquad\qquad \text{DSPEC} \ \frac{A \vdash e : \forall d.\sigma}{A \vdash e : \sigma[\delta/d]}$$

$$\text{ABS} \ \frac{A_x \cup \{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'} \qquad\qquad \text{APP} \ \frac{A \vdash e : \tau \to \tau' \qquad A \vdash e' : \tau}{A \vdash e\,e' : \tau'}$$

$$\text{LET} \ \frac{A \vdash e : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash \mathsf{let}\; x = e \;\mathsf{in}\; e' : \tau}$$

In addition to these rules we have equations relating dimensions:

$$\begin{array}{rll} \delta_1\delta_2 &=_D \delta_2\delta_1 & \text{(commutativity)} \\ (\delta_1\delta_2)\delta_3 &=_D \delta_1(\delta_2\delta_3) & \text{(associativity)} \\ \mathbf{1}\cdot\delta &=_D \delta & \text{(identity)} \\ \delta\delta^{-1} &=_D \mathbf{1} & \text{(inverses)} \end{array}$$

and an inference rule relating equivalent types:

$$\text{DEQ}\ \frac{A \vdash e : \tau_1 \qquad \vdash \tau_1 =_D \tau_2}{A \vdash e : \tau_2}$$

where $=_D$ is lifted to types by the obvious congruence.

It will be observed that none of the rules explicitly introduces types involving base dimensions. We assume that there is a means of declaring constants which represent a base unit for a particular base dimension. For the length dimension, for example, we might have have a constant `metre` of type [L] `real`.

5 Dimensional Type Inference

5.1 Unification—algorithm *Unify*

At the heart of most type inference algorithms is the process of *unification*. Given an equation of the form

$$\tau_1 \stackrel{?}{=} \tau_2$$

we wish to find the *most general unifier*, a substitution S such that

1. $S(\tau_1) = S(\tau_2)$
2. For any other unifier S' there is a substitution S'' such that $S'' \circ S = S'$.

If equality is purely syntactic, there is a straightforward algorithm first devised by Robinson. It accepts a pair of types τ_1 and τ_2 and returns their most general unifier or fails if there is none.

$$\begin{array}{ll} \mathit{Unify}(\alpha,\alpha) & = \text{the identity substitution} \\ & \\ \mathit{Unify}(\alpha,\tau) = \mathit{Unify}(\tau,\alpha) & = \text{if } \alpha \text{ is in } \tau \text{ then fail (no unifier exists)} \\ & \quad\ \text{else return the substitution } \{\alpha \mapsto \tau\} \\ & \\ \mathit{Unify}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) & = S_2 \circ S_1 \\ & \quad\ \text{where } S_1 = \mathit{Unify}(\tau_1,\tau_3) \\ & \quad\ \text{and } S_2 = \mathit{Unify}(S_1(\tau_2), S_1(\tau_4)) \end{array}$$

To extend this to deal with types of the form $[\delta]$ `real`, we unify dimensions using another algorithm *DimUnify*. The additional clause is simply

$$\mathit{Unify}([\delta_1]\,\mathtt{real}, [\delta_2]\,\mathtt{real}) = \mathit{DimUnify}(\delta_1,\delta_2)$$

5.2 Dimensional Unification—algorithm *DimUnify*

We require an algorithm *DimUnify* which accepts two dimension expressions δ_1 and δ_2 and returns a substitution S over the dimension variables in the expressions such that

1. $S(\delta_1) =_D S(\delta_2)$
2. For any other unifier S' there is a substitution S'' such that $S'' \circ S =_D S'$.

This kind of unification is sometimes called *equational*, in contrast to ordinary Robinson unification which is *syntactic* or *free*. In our dimension type system, we want to unify with respect to the four laws listed earlier: associativity, commutativity, identity and inverses. It turns out that this particular brand of unification is decidable and *unitary* [Baa89, Nut90]: there is a single most general unifier if one exists at all. This has the consequence that, as for ML polymorphic types, if an expression is typable then it has a most general type from which any other type may be derived by simple substitution for dimension variables.

We will use Lankford's algorithm for Abelian group unification [LBB84]. It relies on the solution of linear equations in integers, for which there exist several algorithms including one by Knuth [Knu69]. Our treatment is slightly different in that we consider only a single equation.

First we transform the equation to the normalised form

$$d_1^{x_1} \cdot d_2^{x_2} \cdots d_m^{x_m} \cdot \mathrm{B}_1^{\mathrm{y}_1} \cdot \mathrm{B}_2^{\mathrm{y}_2} \cdots \mathrm{B}_\mathrm{n}^{\mathrm{y}_\mathrm{n}} \stackrel{?}{=}_\mathrm{D} \mathbf{1}$$

where d_i and $\mathrm{B_j}$ are distinct dimension variables and base dimensions.

Start by setting S to the empty substitution. If $m = 0$ and $n = 0$ then we are finished already. If $m = 0$ and $n \neq 0$ then fail: there is no unifier. Otherwise, find the dimension variable with exponent x_k of smallest absolute value in the equation. If x_k is negative, first take reciprocals of both sides by negating all exponents. Without loss of generality, we can assume that $k = 1$.

1. If $\forall i.\, x_i \bmod x_1 = 0$ and $\forall j.\, y_j \bmod x_1 = 0$, then the unifier is the following, composed with S.

$$d_1 \mapsto d_2^{-x_2/x_1} \cdots d_m^{-x_m/x_1} \cdot \mathrm{B}_1^{-\mathrm{y}_1/\mathrm{x}_1} \cdots \mathrm{B}_\mathrm{n}^{-\mathrm{y}_\mathrm{n}/\mathrm{x}_1}$$

2. Otherwise introduce a new variable d and compose with S the substitution

$$d_1 \mapsto d \cdot d_2^{-\lfloor x_2/x_1 \rfloor} \cdots d_m^{-\lfloor x_m/x_1 \rfloor} \cdot \mathrm{B}_1^{-\lfloor \mathrm{y}_1/\mathrm{x}_1 \rfloor} \cdots \mathrm{B}_\mathrm{n}^{-\lfloor \mathrm{y}_\mathrm{n}/\mathrm{x}_1 \rfloor}$$

   to transform the equation to

$$d^{x_1} \cdot d_2^{x_2 \bmod x_1} \cdots d_m^{x_m \bmod x_1} \cdot \mathrm{B}_1^{\mathrm{y}_1 \bmod \mathrm{x}_1} \cdots \mathrm{B}_\mathrm{n}^{\mathrm{y}_\mathrm{n} \bmod \mathrm{x}_1} \stackrel{?}{=}_\mathrm{D} \mathbf{1}$$

   If at this stage there are no variables in the equation other than d then there is no solution—no unifier exists.
   Otherwise find the smallest exponent again and repeat the procedure.

This method must terminate because on each iteration we reduce the size of the smallest nonzero coefficient in the equation.

5.3 Inference—algorithm *Infer*

The type inference algorithm for ML is well-known and has been presented in many places. Our version differs in two respects—quantified dimension variables are instantiated at the same time as quantified type variables (when e is a variable), and generalization over free dimension variables is added to the usual generalization over free type variables (when e is a let-expression).

Given a type assignment A and an expression e, the algorithm *Infer* determines a pair (S, τ) where τ is the most general type of e and S is a substitution over the type and dimension variables in A under which this is true.

$$\begin{array}{l}
\mathit{Infer}(A, \mathrm{x}) = (I, \tau[d_1'/d_1, \ldots, d_m'/d_m, \alpha_1'/\alpha_1, \ldots, \alpha_n'/\alpha_n]) \\
\quad \text{where} \\
\quad\quad A(x) \text{ is } \forall d_1 \ldots d_m.\forall\alpha_1 \ldots \alpha_n.\tau \\
\quad\quad d_1', \ldots, d_m' \text{ are fresh dimension variables} \\
\quad\quad \alpha_1', \ldots, \alpha_n' \text{ are fresh type variables}
\end{array}$$

$$\begin{array}{l}
\mathit{Infer}(A, e_1 e_2) = (S_3 \circ S_2 \circ S_1, S_3(\alpha)) \\
\quad \text{where} \\
\quad\quad (S_1, \tau_1) = \mathit{Infer}(A, e_1) \\
\quad\quad (S_2, \tau_2) = \mathit{Infer}(S_1(A), e_2) \\
\quad\quad S_3 = \mathit{Unify}(S_2(\tau_1), \tau_2 \to \alpha) \\
\quad\quad \alpha \text{ is a fresh type variable}
\end{array}$$

$$\begin{array}{l}
\mathit{Infer}(A, \lambda\mathrm{x}.e) = (S, S(\alpha) \to \tau) \\
\quad \text{where} \\
\quad\quad (S, \tau) = \mathit{Infer}(A_\mathrm{x} \cup \{\mathrm{x} : \alpha\}, e) \\
\quad\quad \alpha \text{ is a fresh type variable}
\end{array}$$

$$\begin{array}{l}
\mathit{Infer}(A, \mathsf{let}\ \mathrm{x} = e\ \mathsf{in}\ e') = (S_2 \circ S_1, \tau_2) \\
\quad \text{where} \\
\quad\quad (S_1, \tau_1) = \mathit{Infer}(A, e) \\
\quad\quad (S_2, \tau_2) = \mathit{Infer}(S_1(A_\mathrm{x}) \cup \{\mathrm{x} : \forall d_1, \ldots, d_m.\forall\alpha_1, \ldots, \alpha_n.\tau_1\}, e') \\
\quad\quad d_1, \ldots, d_m \text{ are free dimension variables in } \tau_1 \text{ not in } S_1(A) \\
\quad\quad \alpha_1, \ldots, \alpha_n \text{ are free type variables in } \tau_1 \text{ not in } S_1(A)
\end{array}$$

The algorithm's correctness is shown by two theorems [Lei83, Dam85].

Theorem 1 (Soundness of *Infer*). *If $\mathit{Infer}(A, e)$ succeeds with result (S, τ) then there is a derivation of $S(A) \vdash e : \tau$.*

Theorem 2 (Syntactic Completeness of *Infer*). *If there is a derivation of $S(A) \vdash e : \tau$ then $\mathit{Infer}(A, e)$ is a principal typing for e, i.e. it succeeds with result (S_0, τ_0) and $S =_D S' \circ S_0$, $\tau =_D S'(\tau_0)$ for some substitution S'.*

To prove these theorems we first devise a syntax-oriented version of the inference rules and prove that they are equivalent to the rules given here. Then the proofs follow more straightforwardly by induction on the structure of e; these will appear in a fuller version of this paper.

6 Implementation

The dimension type system described in this article has been implemented as an extension to the ML Kit compiler [Rot92], which is a full implementation of Standard ML as defined in [MTH89].

In order to fit naturally with the rest of Standard ML, the concrete syntax of dimension types is necessarily messy. Dimension variables are distinguished from ordinary type variables and identifiers by an initial underline character, as in `_a`. Base dimensions are ordinary identifiers declared by a special construct. This might also be used to introduce constants representing the base units for the dimension specified, as mentioned in section 4:

```
dimension M unit kg;
dimension L unit metre;
dimension T unit sec;
```

It would be easy to extend this to permit derived dimensions, in a fashion similar to ML type definition.

Dimension expressions are enclosed in square brackets, as is conventional. This happens to fit nicely with the notation for parameterised types. The unit dimension is simply `[]`. Exponents are written after a colon (e.g. area is `[L:2]`) and product is indicated by simple concatenation (e.g. density is `[M L:~3]`).

Any new type or datatype may be parameterised by dimension, by type, or by a mixture of both. Assuming a built-in `real` type we could define `complex` by

```
datatype [_a] complex = make_complex of [_a] real * [_a] real
```

Built-in functions as defined in the prelude are given new types, for example:

```
val sqrt : [_a:2] real -> [_a] real
val sin : [] real -> [] real
val + : [_a] real * [_a] real -> [_a] real
val * : [_a] real * [_b] real -> [_a _b] real
```

The one major problem is ML's overloading of such functions. The Definition of Standard ML gives types such as `num*num -> num` to arithmetic and comparison functions. A type-checker must use the surrounding context to determine whether `num` is replaced by `real` or `int`. We want to give dimensionally polymorphic types to these functions. This makes the Definition's scheme unworkable, especially in the case of multiplication. The current implementation has alternative names for dimensioned versions of these operations.

7 Some Problems

7.1 Equivalent types

ML type inference determines a most general type, if there is one, up to renaming of type variables. For example, the type scheme $\forall\alpha\beta.\, \alpha \times \beta$ is equivalent to $\forall\alpha\beta.\, \beta \times \alpha$. This equivalence is easy for the programmer to understand.

For dimension types, we have principal types with respect to the equivalence relation $=_D$, but there is no obvious way of choosing a canonical representative for a given equivalence class—there is no "principal syntax". Type scheme $\forall d_1 \dots d_n.\tau_1$ is equivalent to $\forall d_1 \dots d_n.\tau_2$ if there are substitutions S_1 and S_2 over the bound variables d_1 to d_n such that

$$\begin{gathered} S_1(\tau_1) =_D \tau_2 \\ \text{and} \\ S_2(\tau_2) =_D \tau_1 \end{gathered}$$

This is *not* just $=_D$ plus renaming of type and dimension variables. For example, the current implementation of the system described in this article assigns the following type scheme to the `correlation` example of section 3.3.

$$\forall d_1 d_2.\,[d_1]\,\texttt{real list} \to [\mathtt{d_2 d_1^{-1}}]\,\texttt{real list} \to [\mathbf{1}]\,\texttt{real}$$

which is equivalent to

$$\forall d_1 d_2.\,[d_1]\,\texttt{real list} \to [\mathtt{d_2}]\,\texttt{real list} \to [\mathbf{1}]\,\texttt{real}$$

by the substitutions $d_2 \mapsto d_2 d_1$ (forwards) and $d_2 \mapsto d_2 d_1^{-1}$ (backwards). The second of these types is obviously more "natural" but I do not know how to formalise this notion and modify the inference algorithm accordingly.

In some cases there does not even appear to be a most natural form for the type. The following expressions are different representations of the principal type scheme for the differentiation function of section 3.3.

$$\begin{gathered} \forall d_1 d_2.\,[d_1]\,\texttt{real} \to ([\mathtt{d_1}]\,\texttt{real} \to [\mathtt{d_2}]\,\texttt{real}) \to ([\mathtt{d_1}]\,\texttt{real} \to [\mathtt{d_2 d_1^{-1}}]\,\texttt{real}) \\ \text{and} \\ \forall d_1 d_2.\,[d_1]\,\texttt{real} \to ([\mathtt{d_1}]\,\texttt{real} \to [\mathtt{d_1 d_2}]\,\texttt{real}) \to ([\mathtt{d_1}]\,\texttt{real} \to [\mathtt{d_2}]\,\texttt{real}) \end{gathered}$$

7.2 Dependent types

Consider a function for raising real numbers to integral powers:

```
fun power 0 x = 1.0
  | power n x = x*power (n-1) x
```

Because the dimension of the result depends on an integer *value*, our system cannot give any better type than the dimensionless

$$\texttt{int} \to [\mathbf{1}]\,\texttt{real} \to [\mathbf{1}]\,\texttt{real}$$

This seems rather limited, but variable exponents are in fact rarely seen in scientific programs except in dimensionless expressions such as power series. A *dependent* type system would give a more informative type to this function:

$$\forall d.\ \Pi \mathrm{n} \in \texttt{int}\ .\ [\mathtt{d}]\,\texttt{real} \to [\mathtt{d^n}]\,\texttt{real}$$

There are also functions which intuitively should have a static type expressible in this system, but which cannot be inferred. Geometric mean is one example. It seems as though its type should be $\forall d.[d]$ `real list` $\to$ [d] `real`, like the arithmetic mean mentioned earlier. Unfortunately its definition makes use of `rpower` and `prod` both of which have dimensionless type:

```
fun rpower (x,y) = exp(y*ln x);
fun prod []      = 1.0
  | prod (x::xs) = x*prod xs;
fun gmean xs = rpower(prod xs, 1.0 / real (length xs))
```

7.3 Polymorphism

Recursive definitions in ML are not polymorphic: occurrences of a recursively defined function *inside* the body of its definition can only be used monomorphically. For the typical ML programmer this problem rarely manifests itself. Unfortunately it is a more serious irritation in our dimension type system.

```
fun prodlists ([], [])       = []
  | prodlists (x::xs, y::ys) = (x*y) :: prodlists (ys,xs)
```

The function `prodlists` calculates products of corresponding elements in a pair of lists, but bizarrely switches the arguments on the recursive call. Naturally this makes no difference to the result, given the commutativity of multiplication, but whilst a version without the exchange is given a type scheme

$$\forall d_1 d_2.\, [d_1]\,\texttt{real list} \times [\mathtt{d}_2]\,\texttt{real list} \to [\mathtt{d}_1\mathtt{d}_2]\,\texttt{real list}$$

the version above has the less general

$$\forall d.\, [d]\,\texttt{real list} \times [\mathtt{d}]\,\texttt{real list} \to [\mathtt{d}^2]\,\texttt{real list}$$

An analagous example in Standard ML is the (useless) function shown here:

```
fun funny c x y = if c=0 then 0 else funny (c-1) y x
```

This has inferred type $\forall\alpha.\,\texttt{int} \to \alpha \to \alpha \to \texttt{int}$ but might be expected to have the more general type $\forall\alpha\beta.\,\texttt{int} \to \alpha \to \beta \to \texttt{int}$. Extensions to the ML type system to permit polymorphic recursion have been proposed. It has been shown that the inference problem for such a system is undecidable [Hen93, KTU93].

The lack of polymorphic lambda-abstraction also reduces the generality of inferred types:

```
fun twice f x = f (f x);
fun sqr x     = x*x;
fun fourth x  = (twice sqr) x;
```

The following type schemes are assigned:

$$\begin{aligned}
\texttt{twice} &: \forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)\\
\texttt{sqr} &: \forall d.\,[d]\,\texttt{real} \to [\mathtt{d}^2]\,\texttt{real}\\
\texttt{fourth} &: [\mathbf{1}]\,\texttt{real} \to [\mathbf{1}]\,\texttt{real}
\end{aligned}$$

We would like `fourth` to have type $\forall d.\,[d]\,\texttt{real} \to [\mathtt{d}^4]\,\texttt{real}$ but cannot have it because this would require `sqr` to be used at two different instances inside `twice`, namely $\forall d.\,[d]\,\texttt{real} \to [\mathtt{d}^2]\,\texttt{real}$ and $\forall d.\,[d^2]\,\texttt{real} \to [\mathtt{d}^4]\,\texttt{real}$.

This is a serious problem but not unpredictable so long as the programmer fully understands the nature of ML-style polymorphism. The same situation occurs in ordinary ML if we change the definition of `sqr` to be $(\mathtt{x}, \mathtt{x})$. This time we expect `fourth` to have the type $\forall \alpha.\, \alpha \to (\alpha \times \alpha) \times (\alpha \times \alpha)$ but the expression is untypable because `sqr` must be used at the two instances $\forall \alpha.\, \alpha \to \alpha \times \alpha$ and $\forall \alpha.\, \alpha \times \alpha \to (\alpha \times \alpha) \times (\alpha \times \alpha)$. In fact, we cannot even write such a term in the second-order lambda calculus. It requires either a higher-order type system such as F_ω, or a system with intersection types, in which we could give `twice` the type

$$\forall \alpha\beta\gamma.\, (\alpha \to \beta) \wedge (\beta \to \gamma) \to (\alpha \to \gamma)$$

and pass in `sqr` at two instances.

8 Related work

8.1 House's extension to Pascal

Before Wand and O'Keefe's recent work, the only attempt at a polymorphic dimension type system was the extension to Pascal proposed by House [Hou83]. In that system, types in procedure declarations may include a kind of dimension variable, as in the following example:

```
function ratio(a : real newdim u; b : real newdim v)
  : real dim u/v;
begin
  ratio := a/b
end;
```

Compared with modern notions of polymorphism, this is rather strange; the `newdim` construct introduces a new variable standing for some dimension, and `dim` makes use of already-introduced variables. It is as though `newdim` contains an implicit quantifier.

8.2 Wand and O'Keefe's system

Wand and O'Keefe define an ML-like type system extended with a single numeric type paramaterised on dimension [WO91]. This takes the form $Q(n_1, \ldots, n_N)$ where n_i are *number expressions* formed from number variables, rational constants, addition and subtraction operations, and multiplication by rational constants. It differs from the $[\delta]$ `real` type of this paper in two ways:

1. A fixed number of base dimensions N is assumed. Dimension types are expressed as a N-tuple of number expressions, so if we have three base dimensions M, L and T, then $Q(n_1, n_2, n_3)$ represents the dimension $[\mathrm{M}^{n_1} \mathrm{L}^{n_2} \mathrm{T}^{n_3}]$.

2. Dimensions have rational exponents. This means, for instance, that the type of the square root function can be expressed as

$$\forall i, j, k.\, Q(i, j, k) \rightarrow Q(0.5 * i, 0.5 * j, 0.5 * k)$$

in contrast to

$$\forall d.\, [d^2]\, \texttt{real} \rightarrow [\texttt{d}]\, \texttt{real}$$

in our system, and this function may be applied to a value of type $Q(1, 0, 0)$, whereas our system disallows its application to [M] `real`.

Their inference algorithm, like ours, generates equations between dimensions. But in their system there are no "dimension constants" (our base dimensions) and equations are not necessarily integral, so Gaussian elimination is used to solve them.

Wand and O'Keefe's types are unnecessarily expressive and can be nonsensical dimensionally. Consider the type $\forall i, j, k.\, Q(i, j, k) \rightarrow Q(i, 2 * j, k)$ which squares the length dimension but leaves the others alone, or $\forall i, j, k.\, Q(i, j, k) \rightarrow Q(j, i, k)$ which swaps the mass and length dimensions. Fortunately no expression in the language will be assigned such types. Also, non-integer exponents should not be necessary—polymorphic types can be expressed without them and values with fractional dimension exponents do not seem to occur in science.

They propose a construct `newdim` which introduces a *local* dimension. In our system the `dimension` declaration could perhaps be used in a local context, in the same way that the `datatype` construct of ML is used already.

The problem of finding canonical expressions for types presumably occurs in their system too, as well as the limitations of implicit polymorphism described here.

9 Conclusion and Future Work

The system described in this paper provides a natural way of adding dimensions to a polymorphically-typed programming language. It has been implemented successfully, and it would be straightforward to add features such as derived dimensions, local dimensions, and multiple units of measure within a single dimension.

To overcome the problems discussed in section 7 it might be possible to make the system more polymorphic, but *only* over dimensions in order to retain decidability. An alternative which is being studied is the use of intersection types.

So far no formal semantics has been devised for the system. This would be used to prove a result analagous to the familiar "well-typed programs cannot go wrong" theorem for ML.

Acknowledgements

This work was supported financially by a SERC Studentship. I would like to thank Alan Mycroft, Francis Davey, Nick Benton and Ian Stark for discussions on the subject of this paper, and the anonymous referees for their comments.

References

[Baa89] F. Baader. Unification in commutative theories. *Journal of Symbolic Computation*, 8:479–497, 1989.

[Bal87] G. Baldwin. Implementation of physical units. *SIGPLAN Notices*, 22(8):45–50, August 1987.

[Car87] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[Dam85] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Department of Computer Science, University of Edinburgh, 1985.

[DMM86] A. Dreiheller, M. Moerschbacher, and B. Mohr. PHYSCAL—programming Pascal with physical units. *SIGPLAN Notices*, 21(12):114–123, December 1986.

[Geh85] N. H. Gehani. Ada's derived types and units of measure. *Software—Practice and Experience*, 15(6):555–569, June 1985.

[Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, April 1993.

[Hou83] R. T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, 1983.

[KL78] M. Karr and D. B. Loveman III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, May 1978.

[Knu69] D. Knuth. *The Art of Computer Programming, Vol. 2*, pages 303–304. Addison-Wesley, 1969.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, April 1993.

[Lan51] H. L. Langhaar. *Dimensional Analysis and Theory of Models*. John Wiley and Sons, 1951.

[LBB84] D. Lankford, G. Butler, and B. Brady. Abelian group unification algorithms for elementary terms. *Contemporary Mathematics*, 29:193–199, 1984.

[Lei83] D. Leivant. Polymorphic type inference. In *ACM Symposium on Principles of Programming Languages*, 1983.

[Män86] R. Männer. Strong typing and physical units. *SIGPLAN Notices*, 21(3):11–20, March 1986.

[Man87] R. Mankin. letter. *SIGPLAN Notices*, 22(3):13, March 1987.

[MTH89] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1989.

[Nut90] W. Nutt. Unification in monoidal theories. In *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 618–632. Springer-Verlag, July 1990.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Rot92] N. Rothwell. Miscellaneous design issues in the ML Kit. Technical Report ECS-LFCS-92-237, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[WO91] M. Wand and P. M. O'Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.