Dimension Inference under Polymorphic Recursion

Mikael Rittri
Department of Computing Science
Chalmers University of Technology and Göteborg University
S-412 96 Göteborg, Sweden. rittri@cs.chalmers.se

Abstract

Numeric types can be given polymorphic dimension parameters, in order to avoid dimension errors and unit errors. The most general dimensions can be inferred automatically.

It has been observed that polymorphic recursion is more important for the dimensions than for the proper types. We show that, under polymorphic recursion, type inference amounts to syntactic semi-unification of proper types, followed by equational semi-unification of dimensions. Syntactic semi-unification is unfortunately undecidable, although there are procedures that work well in practice, and proper types given by the programmer can be checked. However, the dimensions form a vector space (provided that their exponents are rational numbers). We give a polynomial-time algorithm that decides if a semi-unification problem in a vector space can be solved and, if so, returns a most general semi-unifier.

1 Introduction

We will combine three good things as far as possible: dimension types, polymorphic recursion, and automatic type inference. Types with dimension parameters can be inferred in two steps: first you infer the types proper, then you infer the dimensions. Under polymorphic recursion, the first step is known to be undecidable, but there are partial procedures that seem to work in practice, and types that are given by the programmer can be checked. We will simply assume that for some program, the first step has been solved (perhaps not with the most general typing), and we study only the remaining step of dimension inference. It will turn out, somewhat surprisingly, that the most general dimensions can be inferred in polynomial time.

The rest of section 1 introduces and motivates the three good things. Section 2 reviews some algebraic concepts that are needed: syntactic and equational (semi-)unification. Section 3 gives an algorithm for semi-unification in the equational theory for vector spaces. Section 4 motivates that the algorithm can be used to infer dimensions under polymorphic recursion; the reasoning mirrors Henglein’s [6]. Section 5 discusses the relative merits of integers and rational numbers as dimension exponents; the integers make the dimensions form a freely generated Abelian group, whereas the rational numbers make them form a vector space. Section 6 discusses how to react to the undecidability of syntactic semi-unification.

1.1 Dimension types

Most type systems allow programmers to make dimension errors, like adding metres to kilogram. (I assume that dimension checking [3, 15, 28] is familiar.) Such errors can be prevented by giving dimension parameters to the numeric types. There have been several suggestions, but the dimensions are usually monomorphic; most of these suggestions are reviewed in [5]. House [9] was the first to suggest a type system with polymorphic dimensions (for Pascal), but did not discuss automatic type inference. Later, Wand and O’Keefe [27] realised that polymorphic dimensions can be inferred in a way that is analogous to Milner’s algorithm. This was re-discovered independently by Goubault [5] and Kennedy [11]; they also made implementations (for ML).

The type systems of the last three papers are similar. To express polymorphism, you need variables in dimensions. Wand and O’Keefe use variables that range over exponents of dimensions; this is a common practice in physics, but gives a cumbersome notation for dimension types. Both Goubault and Kennedy use variables that range over dimensions; the main difference is that Kennedy allows only integers as exponents of dimensions, while Goubault and the others allow rational numbers. I will need rational numbers, so my type system will be closest to Goubault’s.

To give examples with dimension types, I will use some Haskell-like syntax, but not the Haskell class system. I will write quantifiers explicitly in type-schemes, and I will use only one numeric type, which I call R. This type will always have a dimension parameter, written inside angle brackets. For example,

\[
(+), (-) :: \forall x. R(x) \rightarrow R(x) \rightarrow R(x)
\]

\[
(\ast) :: \forall x y. R(x) \rightarrow R(y) \rightarrow R(x \cdot y)
\]

\[
(/) :: \forall x y. R(x) \rightarrow R(y) \rightarrow R(x \cdot y^{-1})
\]

(The full grammar for dimensions is given in section 3.)

All numeric constants are dimensionless; their type is \(R(\mathbb{I})\). Sometimes, we need a polymorphic zero, but we use a special polymorphic identifier instead of 0.

\[
\text{zero} :: \forall x. R(x)
\]

Base dimensions are declared together with their units:

\[
\text{dimension L(metre)}
\]

declares L to be a new base dimension, and metre to be an identifier of type R(L) with the value \(\text{one metre}\), which we can represent in the same way as the dimensionless number 1. If a function needs a length (a value of type R(L)), it is an error to give it a dimensionless number like 42, you must write \(42 \ast \text{metre}\) instead. Of course, you may also define more peculiar units:
The space shuttle Discovery once flew upside down, because a length measured in feet was given to a computer that
eXpected nautical miles [21]. Even though that was a unit error rather than a dimension error, it would have been
prevented by dimension types, since some unit must be given explicitly to construct a length.

Most transcendental functions must be dimensionless. 

\[ \exp, \ln, \sin : R \rightarrow R\]

As Wand and O’Keefe point out, there is a fundamental theorem of dimensional analysis, the Pi theorem [15, section
10.3], [28, section 23], which indicates that it is no restriction that transcendental functions are dimensionless.
First, we define a multiplicative linear combination of physical quantities 
\( c_1, \ldots, c_n \) as a product \( c_1^{\pi_1} \cdots c_n^{\pi_n} \), where the \( \pi_j \) are rational numbers. The Pi theorem says that
any dimensionally meaningful expression, containing physical quantities 
\( c_1, \ldots, c_n \), can be rewritten on the form
\[ \prod_{j=1}^{n} f(\pi_j c_j) \]
where each \( \pi_j \) is a multiplicative linear combination of the \( c_i \), and \( f \) is a function whose result and arguments are
dimensionless quantities. Therefore, any expression containing a function like \( \exp \) or \( \sin \) can (and must) be rearranged so that the function is used at type \( R \rightarrow R \).

The power function has to be dimensionless, as well, because \( a \ast 2 \) and \( a \ast 3 \) have different dimensions, and we do not have access to dependent types. But we can have a series of specialized functions:

\[ \begin{align*}
\text{square} & : \forall x. R(x) \rightarrow R(x^2) \\
\text{cube} & : \forall x. R(x) \rightarrow R(x^3) \\
\text{sqrt} & : \forall x. R(x) \rightarrow R(x^{1/2}) \\
\text{cbrt} & : \forall x. R(x) \rightarrow R(x^{1/3}) \\
\end{align*} \]

It has been noted [11] that the natural way to code the geometric mean,
\[ \text{geoMean1}(cs) = \text{product}(cs) ** (1/\text{length}(cs)) \]
only gets the dimensionless type \( R[\{\}] \rightarrow R[\{\}] \), because both \((**)\) and \(\text{product}\) have to be dimensionless. \(\text{product}\) multiplies a list of values, and the notation \([\pi]\) stands for the
type of lists over \(\pi\). But geometric mean ought to get the type \( \forall x. [R(x)] \rightarrow R(x) \), and the Pi theorem indicates that it is
possible. Indeed we can write
\[ \text{geoMean2} [] = \text{error "undefined"} \]
\[ \text{geoMean2}(c:cs) = \text{if } c == \text{zero then zero else c * geoMean1(map (/c) (c:cs))} \]
which will get the desired type.

If a language like Haskell or ML is extended with dimension types, the old programs will still type-check, because in
the worst case, numbers can always be treated as dimensionless. This is perhaps surprising, because the purpose of
dimension types is to make fewer programs type-check (in order to catch errors). But the old programs represent a
physical quantity like \(42 \text{ metres}\) by a dimensionless number \(42\). Dimension errors cannot be detected unless the old \(42\)
is changed to the correct \(42 \ast \text{metre}\).

This section ends with some more examples of dimension types, borrowed from [11]. Personally, I am fascinated by
how informative the types are.

\[ \begin{align*}
\text{mean, stdDeviation} & : \forall x. [R(x)] \rightarrow R[\{\}] \\
\text{variance} & : \forall x. [R(x)] \rightarrow R[\{x^2\}] \\
\text{skewness} & : \forall x. [R(x)] \rightarrow R[\{\}] \\
\text{correlation} & : \forall xy. [[R(x), R(y)]] \rightarrow R[\{\}] \\
\end{align*} \]

**1.2 Polymorphic recursion**

Usually, we expect that a function of polymorphic type can be used with any instance of its type. But in Milner’s type
system [2, 19], this is not true in expressions that are part of the function’s definition (via recursion). Consider the
following definition of trapeziform matrices.

\[ \text{infixr :} \]
\[ \text{data Trapez } \alpha \beta = \text{Naught} \ |
\]
\[ \text{triangle :: Trapez } \alpha \beta \rightarrow [\alpha] \]
\[ \text{triangle } = (\langle \text{a}', \langle \rangle \rangle) : \langle \text{b}', \langle \text{c}', \langle \rangle \rangle \rangle : \langle \text{d}', \langle \text{e}', \langle \text{f}', \langle \rangle \rangle \rangle \rangle : \text{Naught} \]

The above is legal Haskell. But try to write a function that picks the first column.

\[ \begin{align*}
\text{fstCol} & : \forall \alpha \beta. \text{Trapez } \alpha \beta \rightarrow [\alpha] \\
\text{fstCol} \text{ (Naught)} & = [ ] \\
\text{fstCol} \text{ ((x,\_):rows)} & = x:fstCol\text{ (rows)} \\
\end{align*} \]

If \(\text{fstCol}\) is given a trapezium of type \(\text{Trapez } A B\), then the first recursive call gets one of type \(\text{Trapez } A (A,B)\). Since
\(\text{fstCol}\) is polymorphic, that should be okay, but such polymorphic recursion is not allowed in Milner’s type system.
The reason is that type inference under polymorphic recursion is equivalent [6, 13] to semi-unification, an undecidable
problem [14]; see section 2. However, if the types are given by the programmer, then polymorphic recursion is allowed in
Hope, Miranda\textsuperscript{TM}. CAML, and one option of the Chalmers Haskell compiler (hbc -fuse-restr), because type checking
is decidable.

Polymorphic recursion can be useful. Mycroft, the originator of the idea, gave a motivating example [20], and
other examples have been discussed on the ML and Haskell mailing lists. Furthermore, polymorphic recursion becomes
much more important when one uses the existentials of Läufner and Odersky [18]. They give an example with a list of
composable functions; in order to apply the composition to a value, polymorphic recursion is needed [18, pages 1418–
1419]. Another example can be found in Prasad [22, tables 3 and 4], where a process algebra is implemented. A process
\(\text{Proc } \alpha\) deals with signals of type \(\alpha\). Among other possibilities, a process \(\text{Proc } \alpha\) can be made from a process \(\text{Proc } \beta\)
together with functions that translate between \(\alpha\) and \(\beta\).

\[ \begin{align*}
\text{data Proc } \alpha & = \text{TRANS } (\beta \\rightarrow \alpha, \alpha \\rightarrow \beta) \ |
\end{align*} \]

\textsuperscript{TM}Miranda is a trademark of Research Software Ltd.
In this type definition, \( \beta \) is existential, since it does not occur in the left hand side. When an \( \alpha \) signal comes to a \( \text{TRANS} \) process, the signal is translated to a \( \beta \) signal and passed on.

\[
\text{listen} :: \forall \alpha. \alpha \rightarrow \text{Proc} \alpha \rightarrow \text{Proc} \alpha
\]

\[
\text{listen} \; \nu \left( \text{TRANS} (g, h) \; p \right) = \text{TRANS} (g, h) \left( \text{listen} \; (h \; \nu) \; p \right)
\]

If \( \text{listen} \) gets a \( \text{Proc} \alpha \), the recursive call gets a \( \text{Proc} \beta \).

### 1.3 Recursion that is polymorphic for dimensions

Kennedy noted that polymorphic recursion is more important for the dimension parameters than for the types proper [11]. He conjectured that type inference would be computable if the recursion in polymorphic only for dimensions. In sections 2–4, we will prove that he was right (at least when rational exponents are used).

To see the importance, let us try to represent a polynomial over these datatypes needs polymorphic recursion. Without polymorphic recursion, we need to use unification which function symbols that are allowed in terms. In syntactic unification, it is usually assumed that any function symbols are allowed. In equational unification, it is usualy weaker than renaming plus equality.

### 1.4 On automatic type inference

Automatic type inference is not merely convenient. Sometimes, the compiler can infer a type that is more general than the programmer expected, which suggests new ways to use the function. So I think automatic type inference is worth pursuing.

### 2 Semi-unification in equational theories

Without polymorphic recursion, we need to use unification in the type inference algorithm. With polymorphic recursion, we need a variant called semi-unification. The semi-unification is normally syntactic, in that unifiers (or semi-unifiers with matchers) have to make types syntacically equal. However, dimensions do not form a free term algebra, and the (semi-)unifiers have to make dimensions equal in an equational theory.

This section reviews these algebraic concepts, and discusses how Baaaz’s procedure for syntactic semi-unification [1] can be extended to equational semi-unification.

#### 2.1 Terms, substitutions, and equivalence

Terms are built in the familiar way from a countably infinite set of variables and a finite set of function symbols with fixed arities. An endomorphism is a total function \( S \) over terms such that

\[
S (f (t_1, \ldots, t_m)) = f (S (t_1), \ldots, S (t_m))
\]

for all function symbols \( f \) and terms \( t_1, \ldots, t_m \). An endomorphism is fully determined by its effect on variables. A substitution is an endomorphism that only changes a finite number of variables; so it is fully determined by an expression of the form \( \{ x_1 := t_1, \ldots, x_n := t_n \} \).

If \( E \) is a set of equational axioms, then \( =_E \) (\( E \)-equality) is the smallest stable congruence relation containing \( E \).

We write \( s \leq_E t \) (\( s \) \( E \)-matches \( t \), or \( t \) is an \( E \)-instance of \( s \)) if there is a substitution \( R \) such that \( R (s) =_E t \), and we call \( R \) an \( E \)-matcher from \( s \) to \( t \). (So \( \leq \) means “less specific than”, which is the more common convention, but be warned that Damas and Milner [2, 19] and others use the opposite convention that \( \leq \) means “less general than”.)

We write \( s \equiv_E t \) if both \( s \leq_E t \) and \( t \leq_E s \); this \( E \)-equivalence is weaker than \( E \)-equality. In fact, although \( \emptyset \)-equivalence is just renaming of variables, \( E \)-equivalence is usually weaker than renaming plus \( E \)-equality.

To extend these relations to substitutions, add a variable set \( X \) as parameter:

\[
S =^X_E T \quad \text{iff} \quad S (x) =_E T (x) \quad \text{for all} \quad x \in X,
\]

\[
S \leq^X_E T \quad \text{iff} \quad R \circ S =^X_E T \quad \text{for some} \quad R,
\]

\[
S \equiv^X_E T \quad \text{iff} \quad S =^X_E T \quad \text{and} \quad T =^X_E S.
\]

We can omit the subscript \( E \) if \( E = \emptyset \), and we can omit the superscript \( X \) if \( X \) is the set of all variables.

#### 2.2 Unification

When you do equational unification, you are trying to find values for variables so that terms become equal in some equational theory; thus, equational unification is just equation solving in various algebras. Fages and Huet [4] give crisp definitions of some basic concepts; Siekmann [25] and Jouannaud and Kirchner [10] give broad surveys.

**Definition 1** An \( E \)-unification problem is a finite set of pairs of terms. Each pair \( (s, t) \) is called an equation and is usually written \( s =^E_E t \). A substitution \( S \) is an \( E \)-unifier of the problem if

\[
S (s) =_E S (t)
\]

for every equation \( s =^E_E t \) in the problem.

Apart from the equational theory, one must also specify which function symbols that are allowed in terms. In syntactic unification, it is usually assumed that any function symbols are allowed. In equational unification, it is common to allow only function symbols that occur in the axioms for
the equational theory. Sometimes, additional constants (free constants) are allowed as well; this is called E-unification with constants. Occasionally, free polyadic function symbols are also allowed. But in this paper, we study only E-unification with constants.

**Definition 2** An E-unifier $S$ of a problem is most general if, for every other E-unifier $S'$,

$$S \leq_{E}^{X} S'$$

where $X$ is the set of variables that occur in the problem.

Unification is unitary in $E$ if every solvable E-unification problem has a most general E-unifier.

A most general E-unifier is usually not unique, but if we have two different most general E-unifiers $S_1$ and $S_2$, then we find that $S_1 \equiv_{E}^{X} S_2$ by using the defining property twice. Hence, a most general E-unifier is unique up to $\equiv_{E}$. An example: for the equation $x = y$, both $\{x := y\}$ and $\{y := x\}$ are most general unifiers; they are equivalent over $\{x, y\}$ but not equal.

The reason to compare only over $X$ is that, in some equational theories, a most general unifier may need to introduce some new, “unused”, variables. It turns out that we cannot compare over the new variables. For the example above, we want $\{x := z \land y := z\}$ to be yet another most general unifier, and indeed, we have that $\{x := z \land y := z\} \leq_{E}^{X} \{x := y\}$, because $\{z := y\} \land \{x := z \land y := z\} \equiv_{E}^{X} \{x := y\}$. But if we wanted to compare over $z$ as well, we would find that $\{x := z \land y := z\} \not\leq_{E}^{X} \{x := y\}$.

Unification is unitary in the free term algebra (syntactic unification), in Abelian groups with constants [16], in vector spaces with constants, and in a few other equational theories [10, 25].

2.3 Semi-unification

Semi-unification is a combination of unification and matching. The applications of syntactic semi-unification are surveyed by Kloury et al. [14], but I have not found any literature on equational semi-unification, apart from my own paper on Abelian groups [24]. If you let $E = \emptyset$ in my definitions, you get syntactic semi-unification (since $=_E$ is then syntactic equality).

**Definition 3** An E-semi-unification problem is a finite set of boxes, which can be numbered $1, \ldots, n$. Each box is a finite set of pairs of terms. Each pair $(s, t)$ is called an inequation and is usually written $s \leq_{E} t$. A substitution $S$ is an E-semi-unifier of the problem if there exist other substitutions $R_1, \ldots, R_m$ that for each box no. $i$ in the problem, and for each inequation $s \leq_{E} t$ in that box, we have that

$$R_i(S(s)) = E S(t)$$

The $R_i$ are called residual substitutions (or matchers). We say that a problem (or just a box) is solvable if it has a semi-unifier; we say that it is solved if the identity substitution is a semi-unifier.

It is a little strange that $S$ acts on both sides of inequations, while the $R_i$ only act on the left hand sides. One way to understand it, is that $S$ really acts on terms with meta-variables, which it replaces by terms with object-variables, between which we need matching relations. But most authors make no technical difference between meta- and object-variables.

**Definition 4** An $E$-semi-unifier $S$ of a problem is most general if, for every other $E$-semi-unifier $S'$,

$$S \leq_{E}^{X} S'$$

where $X$ is the set of variables that occur in the problem.

Semi-unification is unitary in $E$ if every solvable E-semi-unification problem has a most general E-semi-unifier.

An instance of an E-unifier must be an E-unifier, but an instance of an $E$-semi-unifier does not have to be an $E$-semi-unifier. For example, the inequation $x \leq y$ has the identity substitution as a semi-unifier, whose instance $\{x := c\}$, where $c$ is a constant, is not a semi-unifier. This fact is one of the major complications with semi-unification. For example, a box that happens to be solved by a potential semi-unifier cannot be removed, because further instantiations of the substitution may unsolve it again.

2.4 A procedure for equational semi-unification

Syntactic semi-unification is undecidable [14], but Baaz has given a simple semi-decision procedure [1]. The procedure is not guaranteed to terminate, but will terminate with a most general semi-unifier whenever a semi-unifier exists. Other procedures for syntactic semi-unification [6] can give negative answers much more often, but they can be rewritten as variants of Baaz’s procedure, together with detection of most causes of non-termination.

For Baaz, his procedure was a way to prove that syntactic semi-unification is unitary. For us, his procedure is convenient because it is based on unification, and many algorithms for equational unification are already known. Hence, his procedure can be extended to equational theories in which unification is decidable and unitary.

Let $E$ be such an equational theory, and let an $E$-semi-unification problem be given.

1. Let $S$ be the identity substitution.
2. If all boxes are now solved, return $S$ as the most general $E$-semi-unifier.
3. Otherwise, there is an unsolved box

$$\{s_j \leq_{E} t_j\}_{j=1}^{m} \quad (1)$$

Try to let $T$ be a most general substitution such that the test-box

$$\{s_j \leq_{E} T(t_j)\}_{j=1}^{m} \quad (2)$$

is solved, i. e., $R(s_j) = E T(t_j)$ for all $j$ and some $R$. If none such $T$ exists, stop with failure.
4. Otherwise, apply $T$ to all boxes of the problem, on both the left and right hand sides of the inequations.
5. Let $S := T \circ S$.
6. Go to step 2.

If $S$ is a semi-unifier of the box (1), with the residual matcher $R_S$, then the corresponding test-box

$$\{s_j \leq_{E} S(t_j)\}_{j=1}^{m}$$

is solved, as witnessed by the residual matcher $R_S \circ S$. Therefore, any semi-unifier $S$ must be an instance of the
most general substitution \( T \) that makes the test-box (2) solved. Finding such a \( T \) is a matter of \( E \)-unification: rename temporarily the variables in the \( s_j \) terms to be distinct from those in the \( t_j \) terms, then find a most general \( E \)-unifier of the system \( \{ s_j = \frac{c_j}{q_j} t_j \}_{j=1}^{m} \) and restrict it to the variables in the \( t_j \) terms.

The procedure above is not exactly the same as Baaz’s. I have inserted some \( E \)s, of course, and I have also preferred to let the procedure look at one box at a time, while Baaz’s original procedure looks at all boxes simultaneously. But these are superficial differences, and when I refer to “Baaz’s procedure”, I mean the version above.

Even though Baaz’s procedure eventually finds every syntactic semi-unifier, it is not guaranteed to find every equational semi-unifier.

In the theory \( AG \) of Abelian groups with constants, unification is decidable and unitary [16]. I have given an algorithm for \( AG \)-semi-unification of a single inequation [24], which I now think may be possible to write as an instance of Baaz’s procedure. But consider the problem

\[
\{ x^2 \leq_{AG} y \}, \{ y^2 \leq_{AG} x \}
\]

where we use multiplicative notation for the group. With Baaz’s procedure, the substitution \( T \) will alterately be \( \{ y := y^2 \}, \{ x := x^2 \} \) forever, because the first box requires the exponent of \( y \) to be divisible by \( 2 \) (two times the exponent of \( x \)), but the second box requires the exponent of \( x \) to be divisible by the exponent of \( y \). The solution is to make both exponents 0, but this semi-unifier, \( \{ x := 1, y := 1 \} \), can never be found by the step-wise unifications of Baaz’s procedure.

I have failed to construct an algorithm for semi-unification of several boxes in Abelian groups. But when I instead started to look at semi-unification in vector spaces, I found to my surprise that Baaz’s procedure works. It will give a most general semi-unifier, if a semi-unifier exists, and terminate with failure, otherwise.

In the next section, I will present the algorithm for vector spaces, and in section 4, I will show how this can be used to infer dimension parameters.

3 Semi-unification in vector spaces

Dimensions are generated by the grammar

\[
\delta ::= 1 \quad (\text{dim. of a “dimensionless” number}) \\
| c \quad (\text{constant, “base dimension”}) \\
| x \quad (\text{variable}) \\
| \delta^q \quad (\text{where } q \in \mathbb{Q}) \\
| \delta \cdot \delta
\]

The constants are a finite set, depending on the base dimensions we want. For mechanics, we would use \( M, L, \) and \( T \) for mass, length, and time. The variables form a countably infinite set. The exponents \( q \) are rational numbers.

The equational theory \( D \) for dimensions says, precisely, that the dimensions form a vector space over \( \mathbb{Q} \) [28, page 236], [15, page 462]. Multiplication of dimensions behaves like addition of vectors, and raising a dimension to the \( q \)-th power corresponds to multiplying a vector by the scalar \( q \). In this multiplicative notation, the axioms for a vector space are

\[
\begin{align*}
\delta \cdot c &= c \\
\delta \cdot (\varepsilon \cdot \zeta) &= (\delta \cdot \varepsilon) \cdot \zeta \\
\delta^1 &= \delta \\
\delta^{p+q} &= \delta^p \cdot \delta^q \\
\delta \cdot \delta^{-1} &= 1 \\
(\delta \cdot \varepsilon)^p &= \delta^p \cdot \varepsilon^p
\end{align*}
\]

where \( p \) and \( q \) stand for elements in a field (in our case, \( \mathbb{Q} \)), and \( \delta, \varepsilon, \) and \( \zeta \) stand for vectors (in our case, dimensions). [Any vector space is “over” some field. A field is a set with addition and multiplication, such that the set is an Abelian group under addition, the non-zero elements of the set form an Abelian group under multiplication, and the distributive law holds: \( p(q + r) = pq + pr \).]

The space of dimensions is spanned by the constants and the variables. Since there are infinitely many variables, it has infinitely many degrees of freedom, but in a particular problem, only finitely many variables can occur.

It takes some time to get used to this multiplicative notation. For example, a linear combination of dimensions \( \delta_1, \ldots, \delta_m \) is a product \( \delta_1^{q_1} \cdot \ldots \cdot \delta_m^{q_m} \), and the dimensions are linearly dependent if there is a linear combination that equals 1 and has at least one non-zero exponent.

If we choose a fixed order for constants and variables, then the usual normal form of a dimension

\[
c_1^{q_1} \cdot \ldots \cdot c_m^{q_m} \cdot x_1^{q_1} \cdot \ldots \cdot x_n^{q_n}
\]

can be represented by the two \( p \) and \( q \) vectors. We will write them as column vectors, and stack \( p \) on top of \( q \):

\[
\begin{bmatrix}
p_1 \\
\vdots \\
p_m \\
q_1 \\
\vdots \\
q_n
\end{bmatrix}
\]

The horizontal line indicates that we mean a pair of vectors, that is, we can distinguish between the components of \( p \) and \( q \). We can append zeros downwards to \( p \) and \( q \) without changing the dimension they represent. Therefore, if two dimensions are represented by vectors, we can always append zeros downwards to make them have the same shape. The 1 is represented by two zero vectors of arbitrary length.

The algorithm for semi-unification does not depend on the exponents being rational numbers; any field will do.

3.1 Some matrix and vector notation

Matrices are written \( A, B, C, \ldots \), and column vectors are written \( a, b, c, \ldots \). A zero vector is written \( \mathbf{0} \), but a zero matrix is just written \( 0 \).

In the following, let \( A \) be any matrix with \( r \) rows and \( k \) columns. A vector \( w \) belongs to the nullspace of \( A \) if \( Aw = \mathbf{0} \). Of course, \( w \) must have \( k \) components, otherwise the product would not be defined. We write the nullspace of \( A \) as Nullsp(\( A \)). A vector \( u \) belongs to the range of \( A \) if there is some vector \( v \) such that \( Av = u \). The vector \( u \) must have \( r \) components. We write the range of \( A \) as Range(\( A \)). The range can also be regarded as the space spanned by the columns of \( A \). One gets the transpose of \( A \) by replacing each component \( a_{ij} \) by \( a_{ji} \), so the transpose has \( k \) rows and \( r \) columns, and we write it as \( A^T \). Two column vectors \( a \) and \( b \) are orthogonal if \( a^T b = 0 \).
It is unfortunate that the word “dimension” is used in linear algebra. However, instead of saying “an n-dimensional vector space”, we can say “a vector space with n degrees of freedom”. In this paper, a “dimension” is always a physical vector space”, we can say “a vector space with n degrees of freedom”. In this paper, a “dimension” is always a physical

3.2 Substitutions on dimensions

A substitution, in this context, is a linear transformation that does not change constants and only changes a finite number of variables. Since it is a linear transformation, it can be represented by a matrix, but since it must not change constants, the matrix must have the special form

$$\begin{bmatrix} I & K \\ 0 & A \end{bmatrix}$$

where $I$ is an identity matrix, 0 is a zero matrix, and $K$ and $A$ are arbitrary. The number of rows (or columns) in $I$ is the same as the number of constants for base dimensions. We use an abbreviation for this form of matrix, and write it as

$$\begin{bmatrix} K \\ A \end{bmatrix}$$

If the columns of $K$ are $k_1, k_2, \ldots$, and the columns of $A$ are $a_1, a_2, \ldots$, then $\begin{bmatrix} K \\ A \end{bmatrix}$ represents the substitution

$$x_1 := \begin{bmatrix} k_1 \\ a_1 \end{bmatrix}, x_2 := \begin{bmatrix} k_2 \\ a_2 \end{bmatrix}, \ldots,$$

but we can also let it represent the tuple of dimensions

$$\begin{bmatrix} k_1 \\ a_1 \\ k_2 \\ a_2 \\ \ldots \end{bmatrix}$$

What it represents depends on the context, and from a more abstract viewpoint, it does not matter.

Due to the abbreviation, we get special rules for multiplying a matrix by a column vector or by another matrix:

$$\begin{bmatrix} K \\ A \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} p + Kq \\ Aq \end{bmatrix}$$

$$\begin{bmatrix} K \\ A \end{bmatrix} \begin{bmatrix} P \\ Q \end{bmatrix} = \begin{bmatrix} P + KQ \\ AQ \end{bmatrix}$$

3.3 Semi-unification of dimensions

A semi-unification problem for dimensions is a set of boxes of inequations. Each box

$$\{ \delta_1 \preceq \varepsilon_1, \ldots, \delta_m \preceq \varepsilon_m \}$$

can be written in an alternative notation with tuples:

$$\begin{bmatrix} \delta_1, \ldots, \delta_m \end{bmatrix} \preceq \begin{bmatrix} \varepsilon_1, \ldots, \varepsilon_m \end{bmatrix}$$

The box size $m$ can vary between boxes, but this will cause no problem. Now, we represent the dimensions in the tuples by column vectors as before, and we append zero entries downwards to make them have the same shape. The box written with tuples then turns into a matrix inequation

$$\begin{bmatrix} L \\ B \end{bmatrix} \preceq \begin{bmatrix} M \\ C \end{bmatrix}$$

Here, the matrix $L$ is formed by the constant parts of the $\delta_i$, the matrix $B$ is formed by the variable parts of the $\delta_i$, and so on. (An example is given in section 4.1.) When we use the matrix notation, we omit the subscript $D$.

Furthermore, since constants are not changed by substitutions, but can be inverted, it is possible to move all constants from the left to the right hand side. That is, the box above have the same semi-unifiers as

$$\begin{bmatrix} 0 \\ B \end{bmatrix} \preceq \begin{bmatrix} M - L \\ C \end{bmatrix}$$

so we can assume that boxes never have constants in their left hand sides. To ensure this after a substitution, we use a special rule to apply a substitution $\begin{bmatrix} K \\ A \end{bmatrix}$ to a box:

$$\begin{bmatrix} K \\ A \end{bmatrix} \begin{bmatrix} 0 \\ B \end{bmatrix} \preceq \begin{bmatrix} M \\ C \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ AB \end{bmatrix} \preceq \begin{bmatrix} M + KC - KB \end{bmatrix}$$

(4)

3.4 When to stop

We can now give a criterion on when a box is solved. The intuition is that, whenever the dimensions to the left are linearly dependent in a certain way, then those on the right must be dependent in the same way. Consider the box (3), assuming that no constants occur in the left hand side. If $\delta_1^{q_1} \ldots \delta_m^{q_m} =_{\mathbb{D}} \mathbf{1}$, then we also need $\varepsilon_1^{q_1} \ldots \varepsilon_m^{q_m} =_{\mathbb{D}} \mathbf{1}$. The exponents $q_1, \ldots, q_m$ form a column vector $q$ in the nullspace of the left hand side’s matrix.

Theorem 5 The statement

$$\begin{bmatrix} 0 \\ B \end{bmatrix} \preceq \begin{bmatrix} M \\ C \end{bmatrix}$$

holds if, and only if,

$$\begin{cases} \text{Nullsp}(B) \subseteq \text{Nullsp}(M) \\ \text{Nullsp}(B) \subseteq \text{Nullsp}(C) \end{cases}$$

Proof. The statement holds iff there is a substitution $\begin{bmatrix} K \\ A \end{bmatrix}$ such that

$$\begin{bmatrix} K \\ A \end{bmatrix} \begin{bmatrix} 0 \\ B \end{bmatrix} = \begin{bmatrix} M \\ C \end{bmatrix}$$

that is,

$$\begin{bmatrix} K \delta \mid B \end{bmatrix} = \begin{bmatrix} M \mid C \end{bmatrix}$$

or

$$\begin{cases} KB = M \\ AB = C \end{cases}$$

So it is enough to show that, given matrices $B$ and $C$,

$$((\exists A) \ AB = C) \iff \text{Nullsp}(B) \subseteq \text{Nullsp}(C),$$

since we then can get the other half by replacing $A$ and $C$ by $K$ and $M$. We can transpose the matrices, and we know that $AB = C$ iff $B^T A^T = C^T$. By the definition of range, this equation has a solution $A^T$ iff every column of $C^T$ is
in $\text{Range}(B^T)$, that is, iff $\text{Range}(B^T) \supseteq \text{Range}(C^T)$. So we need to show that

$$\text{Range}(B^T) \supseteq \text{Range}(C^T) \iff \text{Nullsp}(B) \subseteq \text{Nullsp}(C)$$

$$(\Rightarrow)$$ Suppose $\text{Range}(B^T) \supseteq \text{Range}(C^T)$, and let $q \in \text{Nullsp}(B)$. Since $\text{Nullsp}(B)$ is the orthogonal complement of $\text{Range}(B^T)$ [26, section 2.5], the vector $q$ is orthogonal to every vector in $\text{Range}(B^T)$, and hence also to every vector in the subspace $\text{Range}(C^T)$. Therefore, $q$ is in the orthogonal complement of $\text{Range}(C^T)$, which is $\text{Nullsp}(C)$ [same ref.]. Since $q$ was an arbitrary vector in $\text{Nullsp}(B)$, we have that $\text{Nullsp}(B) \subseteq \text{Nullsp}(C)$.

$$(\Leftarrow)$$ The opposite direction is analogous.

### 3.5 The algorithm

Now we can solve a semi-unification problem. Although the algorithm can be described as Baaž’s procedure in a vector space, we will use the explicit nullspace requirement of theorem 5, which will simplify the termination proof.

Let a semi-unification problem in a vector space be given.

1. Let $S$ be the identity substitution.
2. If the nullspace requirement is now satisfied for all boxes, then $S$ is a most general semi-unifier. Stop.
3. Otherwise, there is a box,

   \[
   \begin{pmatrix}
   0 \\
   B
   \end{pmatrix} \not\in \begin{pmatrix}
   M \\
   C
   \end{pmatrix}
   \]

   which does not satisfy the requirement. That is, $\text{Nullsp}(B) \not\subseteq \text{Nullsp}(M) \cap \text{Nullsp}(C)$.

   Choose a basis for $\text{Nullsp}(B)$; put its vectors as columns into a matrix $W$. Then $BW = 0$, but

   \[
   \begin{pmatrix}
   M \\
   C
   \end{pmatrix} \begin{pmatrix}
   0 \\
   W
   \end{pmatrix} \neq \begin{pmatrix}
   0 \\
   0
   \end{pmatrix}.
   \]

   Seek a most general substitution \(\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}\) such that

   \[
   \begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix} \begin{pmatrix}
   M \\
   C
   \end{pmatrix} \begin{pmatrix}
   0 \\
   W
   \end{pmatrix} = \begin{pmatrix}
   0 \\
   0
   \end{pmatrix},
   \]

   that is,

   \[
   \begin{pmatrix}
   MW + KCW \\
   ACW
   \end{pmatrix} = \begin{pmatrix}
   0 \\
   0
   \end{pmatrix}
   \]

   or

   \[
   KCW = -MW \quad \text{(6)}
   \]

   \[
   ACW = 0 \quad \text{(7)}
   \]

   There may be no $K$ that satisfies the inhomogeneous equation (6). If so, there is no semi-unifier — stop with failure.

4. Otherwise, choose any particular $K$ that solves (6), and let $A$ be a matrix such that the columns of its transpose $A^T$ form a basis for the nullspace of $W^TC^T$. Then, the solutions to (5) are precisely the substitution $\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}$ and its instances.

5. Apply $\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}$ to every box, and let $S := \begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix} \circ S$.

6. Go to step 2.

An example of how the algorithm works can be found in section 4.1.

#### 3.6 Partial correctness

The substitution $\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}$ transforms the original box

\[
\begin{pmatrix}
   0 \\
   B
   \end{pmatrix} \not\in \begin{pmatrix}
   M \\
   C
   \end{pmatrix}
\]

into

\[
\begin{pmatrix}
   0 \\
   AB
   \end{pmatrix} \not\in \begin{pmatrix}
   M + KC - KB \\
   AC
   \end{pmatrix}
\]

according to rule (4). By construction of $K$ and $A$, we have that

\[
\begin{cases}
\text{Nullsp}(B) \subseteq \text{Nullsp}(M + KC) \\
\text{Nullsp}(B) \subseteq \text{Nullsp}(AC),
\end{cases}
\]

and clearly

\[
\text{Nullsp}(B) \subseteq \text{Nullsp}(KB)
\]

so we get

\[
\begin{cases}
\text{Nullsp}(B) \subseteq \text{Nullsp}(M + KC - KB) \\
\text{Nullsp}(B) \subseteq \text{Nullsp}(AC).
\end{cases}
\]

Thus, the new right hand side satisfies the requirements from the old left hand side. However, there is no guaranty that it satisfies all requirements from the new left hand side, because $\text{Nullsp}(AB)$ may be larger than $\text{Nullsp}(B)$. Fortunately, $\text{Nullsp}(AB)$ must be a superset of $\text{Nullsp}(B)$, so all requirements from $B$ still remain. This means that we have no reason to regret the choice of $\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}$; there is no need to backtrack. But we may need to iterate the process.

#### 3.7 Termination

To see that the algorithm terminates, we first recall that we will do step 3 only if

\[
\begin{pmatrix}
   M \\
   C
   \end{pmatrix} \begin{pmatrix}
   0 \\
   W
   \end{pmatrix} \neq \begin{pmatrix}
   0 \\
   0
   \end{pmatrix}
\]

So $MW \neq 0$ or $CW \neq 0$. Suppose $MW \neq 0$, but $CW = 0$. Then, we do need a substitution $\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}$, but the requirement (6) will be impossible to satisfy. Thus, the substitution $\begin{pmatrix}
   K \\
   \lambda
   \end{pmatrix}$ has a chance to exist only if $CW \neq 0$. In that case, there is at least one column $w$ of $W$ such that $Cw \neq 0$, but since $A$ is required to satisfy $ACW = 0$, we have in particular that $ACw = 0$.

Consider now the entire system. It may have several boxes, each with a matrix corresponding to $C$. Call them $C_1, \ldots, C_n$. By construction, they have the same number of rows, so we can put them together horizontally into a larger matrix

\[
C_* = [C_1 \cdots C_n]
\]

If the current iteration step of the algorithm was performed on box no. $i$, we have shown above that there is a column vector $w$ such that $C_iw \neq 0$, but $AC_iw = 0$. Now let us construct $w_i$ as the vertical concatenation of $n$ column vectors, $w_1, \ldots, w_n$. The heights of the $w_i$ are chosen so that
all products $C_i w_j$ exist, but all $w_j$ are zero vectors, except $w_i$, which is the vector $w$ above. Then

$$C_i w_j = C_i w \neq 0,$$

but

$$AC_i w_j = AC_i w = 0.$$  

Hence, $w_j \in \text{Nullsp}(AC_i)$, but $w_j \notin \text{Nullsp}(C_i)$. Since the nullspace of $AC_i$ must be a superset of the nullspace of $C_i$, the existence of $w_j$ shows that it is a proper superset, which means that it has more degrees of freedom.

For any matrix $U$, let $\text{Nullity}(U)$ be the number of degrees of freedom in $\text{Nullsp}(U)$, let $\text{Rank}(U)$ be the number of degrees of freedom in $\text{Range}(U)$, and let $\text{Columns}(U)$ be the number of columns. It is a fundamental theorem of linear algebra [26, section 2.4] that

$$\text{Nullity}(U) + \text{Rank}(U) = \text{Columns}(U)$$

In our algorithm, one effect of an iteration step is that each matrix $C_i$ is replaced by a new matrix $AC_i$, so $C_i$ is replaced by $AC_i$. But

$$\text{Columns}(AC_i) = \text{Columns}(C_i),$$

so

$$\text{Nullity}(AC_i) + \text{Rank}(AC_i) = \text{Nullity}(C_i) + \text{Rank}(C_i).$$

We have shown above that

$$\text{Nullity}(AC_i) > \text{Nullity}(C_i),$$

so

$$\text{Rank}(AC_i) < \text{Rank}(C_i).$$

Hence, the size of a semi-unification problem can be defined as the natural number $\text{Rank}(C_i)$, which must decrease after each successful iteration step of the algorithm. The algorithm must terminate.

The rank of $C_i$ is not greater than its number of rows, which equals the number of distinct variables in the original semi-unification problem.

### 3.8 Complexity

Let $n$ be the number of boxes, and let $m$ be the maximal numbers of rows, or columns, in a matrix. Let us just count the number of operations on scalars. Then, each time they are performed, step 2 of the algorithm needs $O(n^3m)$ operations. steps 3 and 4 need $O(m^3)$ operations, and step 5 needs $O(n^3m)$ operations [26, sections 1.2.1.4]. The loop may be iterated $m$ times, so the total number of operations should be $O(n^3m^2)$.

Can we really assume a constant time for the operations (addition, multiplication, and division) on scalars? If we want to use exact arithmetic on rational numbers, there is a risk that the space needed for each rational number grows and grows. But this will hardly happen in our application. Krantz et al. [15] list over 100 physical quantities, and note that the exponents lie between $-4$ and $4$. House [9] notes that fractional exponents are usually not more complex than $2$. Indeed, Goubault represents fractional exponents reliably by floating point numbers [5].

Still, my algorithm has not been implemented, and perhaps some program, which contain dimension errors, could cause the algorithm to produce quite large exponents before the errors are detected.

### 4 Inference of dimension types

Henglein [6] as well as Kfoury et al. [13] have shown that under polymorphic recursion, syntactic semi-unification is sufficient and necessary for type inference. For dimension inference under polymorphic recursion, I intend to show only that semi-unification in vector spaces is sufficient. It turns out that Henglein’s proof of sufficiency is easy to adapt.

We will use a simple lambda-calculus with let- and fix-expressions, and with real numbers as the only predefined datatype (except functions). The grammar for dimension types is then

$$\tau ::= \alpha \quad \text{(type variable)}$$
$$\mid \mathbb{R}(\delta) \quad \text{(numeric type with dim. parameter)}$$
$$\mid \tau \rightarrow \tau \quad \text{(function type)}$$

The grammar for dimensions $\delta$ was given in section 3.

We shall soon give a type inference system, such that semi-unification of dimension types is sufficient for type inference. Earlier, we studied semi-unification of dimensions only, not dimension types. But we will see that any semi-unifier of dimension types can be constructed as a composition of two substitutions: one is (essentially) a syntactic semi-unifier of proper types; one is a semi-unifier of dimensions.

The notion of a substitution on dimensions was defined in section 3.2. Such a substitution, call it $S$, can be lifted to dimension types in an obvious way:

$$S(\alpha) = \alpha$$
$$S(\mathbb{R}(\delta)) = \mathbb{R}(S(\delta))$$
$$S(\tau \rightarrow \tau') = S(\tau) \rightarrow S(\tau')$$

We can also define a substitution on dimension types as any function $T$ that only changes a finite number of type variables and also satisfies the following equalities, for some substitution $S$ on dimensions:

$$T(\tau \rightarrow \tau') = T(\tau) \rightarrow T(\tau')$$
$$T(\mathbb{R}(\delta)) = S(T(\mathbb{R}(\delta)))$$

If $S$ is the identity substitution, we say that $T$ is pure.

To apply a substitution on dimension types, we can make two passes over the argument: the first pass only changes a finite number of type variables; the second pass only changes dimension variables. That is, any substitution on dimension types can be expressed on the form

$$S \circ T$$

where $S$ is a substitution on dimensions (lifted to dimension types), and $T$ is a pure substitution on dimension types. Now, to separate the responsibilities of $S$ and $T$ even further, we wish that all decisions about dimensions are made by $S$. We are usually concerned about how $S \circ T$ behaves on a finite set of dimension types, which can contain only a finite set of dimension variables. Then, construct $U$ such that, for each occurrence of a dimension parameter in some output from $T$, the function $U$ will instead produce a new unique dimension variable. For example if

$$T = \{ \alpha_1 ::= (\mathbb{R}(x) \rightarrow \mathbb{R}(x^{-1})), \quad \alpha_2 ::= (\mathbb{R}(L)) \},$$

then we should make

$$U = \{ \alpha_1 ::= (\mathbb{R}(y) \rightarrow \mathbb{R}(z)), \quad \alpha_2 ::= \mathbb{R}(w) \},$$
where $y$, $z$, and $w$ are new unique dimension variables. If we really need to make $y$ the inverse of $z$, and to make $w$ equal to $L$, we can let the substitution $S$ on dimensions handle it.

Finally, suppose that we have a problem of semi-unification of dimension types. If there is a semi-unifier, it can be written on the form

$$S \circ \mathcal{U}$$

as above. The proper part of $\mathcal{U}$, which you get by erasing all dimension parameters, is a syntactic semi-unifier of (the proper part of) the problem. If we apply $\mathcal{U}$ to the problem, we get a new problem that is solved for the types proper, but not for the dimensions. The remaining part of the problem is now a semi-unification problem of dimensions.

Let us now, finally, look at our programming language. The grammar for expressions is

$$e ::= i \quad \text{(identifier)}$$
$$r \quad \text{(numeric constant)}$$
$$\lambda e \quad \text{(abstraction)}$$
$$e e \quad \text{(application)}$$
$$\text{let } i = e \text{ in } e$$
$$\text{fix } e$$

Although numeric constants are not allowed after lambda, let, or fix, they can otherwise be treated just like identifiers, whose types are always $\mathbb{R}$. In an initial type environment, Hence, we shall not discuss numeric constants further. Our language has no construct for declaring new base dimensions; instead, we assume that the types of special identifiers like litre, kilogram, (>, <), and zero are also in the initial type environment.

We shall construct a first order, syntax-directed, type inference system, which allows polymorphic recursion. It will be a copy of one of Henglein’s inference systems, except that his (ordinary) types will be replaced by dimension types. For brevity, I shall often write “types” instead of “dimension types” in the following.

A type environment is a finite mapping from identifiers to types. If $\Gamma$ is a type environment, then $\Gamma[i : \tau]$ is another one that agrees with $\Gamma$ everywhere except on the argument $i$, for which it gives $\tau$ instead. A sequent has the form

$$\Gamma; \varphi \vdash e : \tau$$

where $\varphi$ is a list of types, called non-generic types. The idea is that when a dimension type is assumed for a lambda-bound identifier, the type is also added to the list of non-generic types. When a type is assumed for a let- or fix-bound identifier, it is not added to the list.

A type inference system for dimension types and polymorphic recursion is given in table 1. I got it from Henglein [6, figure 4], simply by adding “$D$” subscripts. I have also corrected a subtle error.\footnote{Instead of my equivalence (mutual matchability) between $\tau$, and $\varphi$ in (FIX-$P_{D}^{\varphi}$), Henglein’s rule required them to be equal, but this often destroys the polymorphic recursion; for example, fix i. k. (i) k cannot be typed. An inference tree will need $\tau_i$ to be a variant of $\tau$ with all generic variables renamed; my version allows such renaming. Another idea would be to make the variable renaming unnecessary; see section 4.2.}

When a lambda-bound identifier is used in several places, all occurrences must get the same type. This ensured by the premiss $(\tau, \varphi) \leq_D (\tau', \varphi')$ in rule (TAUT$^{P_{D}}$). Any substitution that instantiates $\tau$ to $\tau'$ is required not to change the non-generic types in $\varphi$.

It is easy enough to see that typing can be done via semi-unification. For every expression to be typed, there is only one possible shape of the inference tree; the shape is isomorphic with the syntax tree of the expression. Construct this tree, using new unique type variables instead of the meta-variables $\tau, \tau', \ldots$. It remains to replace these type variables by the most general types, for which all boxes of inequations that appear in the inference tree are satisfied. This is a semi-unification problem over dimension types. As explained earlier, this can be factored into one problem of syntactic semi-unification, and, if we can solve that, a remaining problem of semi-unification of dimensions.

### 4.1 An example of dimension inference

As an example, let us infer the dimension type for the expression

$$\text{fix } f. \lambda x. f(\sqrt{x})$$

in a type environment $\Gamma$ in which $\sqrt{x}$ has the type $\mathbb{R} \xrightarrow{D} \mathbb{R}$. There is only one possible shape of an inference tree for the expression (table 2). It remains to replace the meta-variables $\tau$ by the most general types that satisfy all equations and inequations in the tree. Since our substitutions on dimension types act on object-variables rather than meta-variables, we replace each meta-variable $\tau$ by a unique object-variable $\alpha$. Then we extract the equations and inequations from the tree.

First, we disregard all dimensions and find a most general substitution (on proper types) that solves the proper parts of (I)–(VIII). One such substitution is

$$\mathcal{T} = \{ \alpha_3 := R \to \alpha_r, \quad \alpha_f := R \to \alpha_1, \quad \alpha_s := R, \quad \alpha_{\mathbb{R}} := R \to \alpha_2, \quad \alpha'_{\mathbb{R}} := R \to R, \quad \alpha'_{\mathbb{R}} := R, \quad \alpha_1 := R, \quad \alpha'_{\mathbb{R}} := R \to \alpha_1 \}$$

Next, we need a most general substitution $\mathcal{U}$ on dimension types, whose proper part is $\mathcal{T}$. One such substitution is

$$\mathcal{U} = \{ \alpha_3 := \mathbb{R}(x_2) \to \alpha_r, \quad \alpha_f := \mathbb{R}(x_3) \to \alpha_1, \quad \alpha_s := \mathbb{R}(x_4), \quad \alpha_{\mathbb{R}} := \mathbb{R}(x_5) \to \alpha_2, \quad \alpha'_{\mathbb{R}} := \mathbb{R}(x_6) \to \mathbb{R}(x_7), \quad \alpha'_{\mathbb{R}} := \mathbb{R}(x_8), \quad \alpha_1 := \mathbb{R}(x_9), \quad \alpha'_{\mathbb{R}} := \mathbb{R}(x_{10}) \to \alpha_1 \}$$

Then, $\mathcal{U}$ is applied to the problem, which is simplified to

$$(\mathbb{R}(x_3) \to \alpha_1, \mathbb{R}(x_9)) \leq_D (\mathbb{R}(x_{10}) \to \alpha_r, \mathbb{R}(x_9))$$

(\text{II})
whose proper part is solved, that is, if we disregard the dimension parameters, the problem above is solved. Since the remaining task only concerns dimensions, we can lift them out to get a clean problem over dimensions:

\[(x_3, x_9) \leq_D (x_{10}, x_9)\]  
\[(x_1, x_9) \leq_D (x_6, x_7, x_9)\]  
\[(x_9, x_9) \leq_D (x_8, x_9)\]  
\[x_3 \equiv_D x_2\]  
\[x_2 \leq_D x_5\]  

\[(\tau, \overline{\tau}) \leq_D (\tau, \overline{\tau})\]  
\[\Gamma[i : \tau_i]; \overline{\tau} \vdash i : \tau\]  
\[\Gamma[i : \tau_i]; \overline{\tau} \vdash e : \tau' \quad \tau' =_D \tau_2 \to \tau\]  
\[\Gamma; \overline{\tau} \vdash \lambda_i.e : \tau'\]  
\[\Gamma; \overline{\tau} \vdash \lambda_i.e : \tau''\]  
\[\Gamma; \overline{\tau} \vdash (e e') : \tau''\]  
\[\Gamma; \overline{\tau} \vdash \lambda e (\text{let } i = e' \text{ in } e) : \tau''\]  
\[\Gamma; \overline{\tau} \vdash (\text{fix } i.e) : \tau''\]  

Table 1: A first order, syntax directed, inference system for dimension types with polymorphic recursion, similar to [7, fig. 4]. A premiss \((\tau_1, \ldots, \tau_m) \leq_D (\tau'_1, \ldots, \tau'_m)\) is true if there is a substitution \(\mathcal{R}\) such that \(\mathcal{R}(\tau_k) = \tau'_k\) for all \(k\). A premiss \(\overline{\tau} \equiv_D \overline{\tau}\) is true if both \(\overline{\tau} \leq_D \overline{\tau}\) and \(\overline{\tau} \geq_D \overline{\tau}\) are true.

\[\Delta; \tau_1 \vdash f : \tau'_j\]  
\[\Delta; \tau_1 \vdash \text{sqrt} : \tau'_s\]  
\[\Delta; \tau_1 \vdash i : \tau'_i\]  
\[\Delta; \tau_1 \vdash (\text{sqrt } i) : \tau_s\]  
\[\Delta; \tau_1 \vdash f (\text{sqrt } i) : \tau_r\]  
\[\Delta; \tau_1 \vdash \text{fix } f.\lambda i.f (\text{sqrt } i) : \tau_k\]  

Table 2: An inference tree, in which \(\Delta\) abbreviates the environment \(\Gamma\{f : \tau_j, i : \tau_i\}\), and \(\Gamma\) is some initial environment in which \(\text{sqrt}\) has the type \(R(x_1^2) \to R(x_1)\). The symbol \(\emptyset\) denotes an empty sequence of non-generic types.

\[R(x_7^2) \to R(x_1), R(x_9) \leq_D (R(x_6) \to R(x_7), R(x_9))\]  
\[R(x_9), R(x_9) \leq_D (R(x_8), R(x_9))\]  
\[R(x_9) \to \alpha_2 \equiv_D R(x_9) \to \alpha_2\]  
\[R(x_9) \to \alpha_2 \equiv_D R(x_9) \to \alpha_2\]  
\[R(x_10) \to \alpha_2 \equiv_D R(x_9) \to \alpha_2\]  
\[R(x_2) \to \alpha_2 \equiv_D R(x_9) \to \alpha_2\]  

Apart from the inequations (i)–(v), there are equations (vi)–(viii). Equations in a vector space can be solved with Gaussian elimination, and we find that the substitution

\[S_1 = \{x_7 := x_4, x_8 := x_6, x_9 := x_2, x_{10} := x_4\}\]

is a most general unifier of the equations. When this substitution is applied to the problem, the equations become solved, and they will remain solved under any further instantiation, since an instance of a unifier is also a unifier. Hence, we apply the substitution and remove the equations, getting a simplified problem:

\[(x_3, x_9) \leq_D (x_{10}, x_9)\]  
\[(x_1, x_9) \leq_D (x_6, x_7, x_9)\]  
\[(x_9, x_9) \leq_D (x_8, x_9)\]  
\[x_3 \equiv_D x_2\]  
\[x_2 \leq_D x_5\]  

\[(\text{TAUT}'_D)\]  
\[(\text{ABS}'_D)\]  
\[(\text{APPL}'_D)\]  
\[(\text{LET}'_D)\]  
\[(\text{FIX-P}'_D)\]
We shall solve this problem using the algorithm from section 3.5.

First we do it quite informally, without switching to the matrix notation. The boxes (i′′) and (iii′′) are not solved yet. We choose box (iii′′), and find that the two dimensions to the left are linearly dependent, since they are equal. The two dimensions to the right must also be made equal, and a most general substitution that does so is

\[ S_2 = \{ x_6 := x_2 \} \]

After we apply \( S_2 \) the problem, we get

\[
\begin{align*}
(x_3, x_2) & \leq_D (x_4, x_2) \quad (i′′) \\
(x_1, x_2) & \leq_D (x_2, x_4) \quad (ii′′) \\
(x_2, x_2) & \leq_D (x_2, x_2) \quad (iii′′) \\
x_3 & \equiv_D x_2 \quad (iv′′) \\
x_2 & \leq_D x_5 \quad (v′′)
\end{align*}
\]

Only box (ii′′) is unsolved now. In its left hand side, the first dimension is the square of the second. This linear dependency must be mirrored in the right hand side, for example by instantiating \( x_4 \) to the square root of \( x_2 \). Applying the substitution \( S_3 = \{ x_4 := x_2^{1/2} \} \), we get the problem

\[
\begin{align*}
(x_3, x_2) & \leq_D (x_2^{1/2}, x_2) \quad (i′′′) \\
(x_1, x_2) & \leq_D (x_2, x_2^{1/2}) \quad (ii′′′) \\
(x_2, x_2) & \equiv_D x_2 \quad (iii′′′) \\
x_3 & \equiv_D x_5 \quad (iv′′′)
\end{align*}
\]

in which all boxes happen to be solved. A most general solution of the original problem (I)–(VIII) is \( S_3 \circ S_2 \circ S_1 \circ U \). Applying it to \( \alpha_{KS} \), we find that the inferred type for expression (8) is \( R(x_5) \rightarrow \alpha_2 \).

Now, let us compute \( S_2 \) in detail, using the matrix notation. The problem (i′)–(v′) is first written with matrices, and we get

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

(If you miss the horizontal lines from section 3.5, you can put them on top of the matrices. The matrices written \( M \) and \( K \) in the algorithm have vanished in our example. They would contain the exponents of base dimensions like length, and no base dimensions have appeared, because the original expression (8) did not contain any units like \textit{metre}.)

When we check all boxes, we will find, in particular, that box (iii′′′) is unsolved, because the nullspace of the left hand matrix is spanned by the column in the matrix

\[
W = \begin{pmatrix} 1 \\ -1 \end{pmatrix}
\]

which is not in the nullspace of the right hand matrix (its nullspace contains just the zero vector). According to steps 3 and 4 in the algorithm, we should construct a matrix \( A^T \) such that its columns form a basis for the nullspace of

\[
W^T C^T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}
\]

One way to choose \( A^T \) is

\[
A^T = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

The untransposed matrix \( A \) then represents the substitution \( \{ x_6 := x_2 \} \), that is, \( S_2 \), as we can recognize by reading \( A \) using the rule in section 3.2. To continue with the algorithm, we should apply \( S_2 \) to each box, using rule (4).

The rest of the example, consisting of finding \( S_3 \) in the matrix notation, is similar and the details are omitted.

### 4.2 An alternative first order type system

Recently, Henglein has told me that there is a better way to design a first order type system [7]. Instead of just having one global list of non-generic types in a sequent, one can also have a local list for each identifier in the environment. This list contains the types that were non-generic when the identifier was bound. This should be simpler to implement and allow more efficient type inference. Indeed, Augustsson has told me that this is how type inference is implemented in the Chalmers Haskell compiler.
However, I have chosen not to change table 1, because it may be more familiar, being similar to the first-order systems in Milner’s [19] and Henglein’s [6] and to Reade’s implementation [20].

4.3 On type-schemes and semantic soundness

The inference system was designed to make it clear that semi-uniﬁcation is sufﬁcient for type inference. However, we should show that the inference system is sound: to paraphrase Milner, we should show that “well-dimensional” expressions do not go wrong. Essentially, we must show that the type of an expression does not change during evaluation.

Since my inference system is similar to Milner’s [19], Henglein’s [6], and Mycroft’s [20], I expect that their soundness proofs can be adapted to my system. The best way is probably to go via an inference system that uses type-schemes. I will not construct such an inference system here, but I will argue that it is possible.

In a type-scheme, type variables and dimension variables may be universally quantiﬁed at the top level. Type-schemes are generated by the grammar

\[
\sigma ::= \forall \alpha. \sigma \quad \text{(type variable)} \\
| \forall x. \sigma \quad \text{(dim. variable)} \\
| \tau \quad \text{(simple type)}
\]

From a type-scheme, we get its simple generic instances by substituting simple types for its bound type variables and dimensions for its dimension variables. The bound variables serve the same purpose as the the list \( \vec{\tau} \) of non-generic types did in table 1. However, that list represented what could not be instantiated in rules (TAUT\(\tau\)) and (FIX-P\(\tau\)), whereas the bound variables in a type-scheme represent what can be instantiated.

In table 1, suppose that an identiﬁer has a type \( \tau \) in an environment where \( \vec{\tau} \) is the list of non-generic types. Then, the identiﬁer can be used with any type \( \vec{\tau}' \) that is an instance of \( \vec{\tau} \), provided that the matcher from \( \vec{\tau} \) to \( \vec{\tau}' \) does not change any type in \( \vec{\tau} \). Is there always a type-scheme that corresponds to \( \vec{\tau} \) and \( \vec{\tau}' \)?

Well, for ordinary type variables, the rule is well-known: construct a type-scheme from \( \vec{\tau} \) by quantifying all type variables in \( \vec{\tau} \) that do not occur in \( \vec{\tau} \). However, this is not correct for dimension variables. Let \( \vec{\tau} = R(x) \rightarrow R(y) \), and let \( \vec{\tau}' \) contain only the type \( R(x \cdot y) \). Then, there are many way to instantiate \( \vec{\tau} \), using matchers that do not change \( R(x \cdot y) \). (In the \( x \cdot y \)-plane, there are many linear transformations that keep the line \( x = y \) unchanged.) But of the four obvious ways to construct a type-scheme,

\[
\forall x. y. R(x) \rightarrow R(y), \forall x. R(x \rightarrow R(y), \forall x. R \rightarrow R(y), \forall y. R \rightarrow R(y),
\]

there is none that expresses the same thing. In particular, the rule that we used for ordinary type variables would give too little polymorphism.

The trick is to make a transformation ﬁrst. In our example, we need to transform the \( x \cdot y \)-plane into an isomorphic plane, but keeping ﬁxed the line (or subspace) that is spanned by \( x \cdot y \). To do this, choose a basis for the \( x \cdot y \)-plane, in which as many vectors as possible come from the ﬁxed subspace. Then, choose a transformation that does not change the basis vectors from the ﬁxed subspace, but maps the other basis vectors to new, unused variables. After this is done, quantify the new variables. In our case, we can pick \( x \) and \( x \cdot y \) for the basis. We want to map \( x \) to a new variable \( z \), while keeping \( x \cdot y \) unchanged; this forces \( y \) to be mapped to \( x \cdot y \cdot z^{-1} \). After this transformation, the type \( \tau \) becomes \( R(z) \rightarrow R(x \cdot y \cdot z^{-1}) \), and the correct type-scheme is \( \forall x. R(z) \rightarrow R(x \cdot y \cdot z^{-1}) \).

The procedure above works generally. In the general case, instead of the \( x \cdot y \)-plane, we use a space that we can call \( \dim \vec{\tau} \) (which is spanned by the individual dimension variables that occur in \( \vec{\tau} \). Instead of the ﬁxed line \( x = y \), we use the subspace of \( \dim \vec{\tau} \) that must be ﬁxed according to \( \vec{\tau} \). To ﬁnd this ﬁxed subspace, we can ﬁrst erase all dimension constants that occur in \( \vec{\tau} \), because a requirement to keep for example \( \sqrt{R^2 \cdot L^3 \cdot x^4 \cdot y^5} \) ﬁxed, is the same as a requirement to keep \( x^4 \cdot y^5 \) ﬁxed. After this, the remaining dimensions in \( \vec{\tau} \) span a space which we can call \( \dim \vec{\tau} \) (erase DimConst(\( \vec{\tau} \))). (Note that this space is spanned by entire dimensions, not by individual dimension variables. For example, \( x \cdot y \) spans the line \( x = y \); it does not span the whole \( x \cdot y \)-plane.) The ﬁxed subspace of \( \dim \vec{\tau} \) is then just the intersection of \( \dim \vec{\tau} \) and \( \dim \vec{\tau} \) (erase DimConst(\( \vec{\tau} \)).

Thus, it should be possible to construct an inference system with type-schemes, which corresponds closely to a ﬁrst-order system. It is probably easiest to start from Henglein’s modiﬁed ﬁrst-order system (section 4.2). Then, the standard techniques to prove semantic soundness should work.

In fact, the previous papers on polymorphic dimensions [5, 11, 27] use the same rule to quantify dimension variables as for ordinary type variables. But the trouble in our example was that \( x \cdot y \), but neither \( x \) nor \( y \) by itself, was a dimension parameter in the type of a lambda-bound identiﬁer. I think that such situations can always be avoided; therefore, the type systems in [5, 11, 27] should give as much polymorphism as my table 1 after all (apart from the polymorphic recursion).

A full account of these matters will appear in Kennedy’s thesis [12].

5 Integers versus rational numbers

Kennedy [11] and Hilfinger [8] believe that integers sufﬁce for dimension exponents, but most other authors allow rational numbers.

A hundred years ago, integers would have failed, because it was common not to use new base dimensions for electric charge or current, instead, they were expressed in terms of mass, length and time (simply by deﬁning either permeability or permittivity to be dimensionless). The dimensions for charge and current then got fractional exponents. But with the seven base dimensions of SI, it is not clear that we need fractional exponents.

Let us look at another example. The dimension type of the square root function can be the same

\[
sqrt ::= \forall x. R(x^2) \rightarrow R(x)
\]

but without rational exponents, we cannot take the square root of a temperature \( R(\text{Temp}) \), because \( x \) would need to be instantiated to \( \text{Temp}^{1/2} \). Now, the speed of sound in a gas is proportional to the square root of the temperature! But by placing the (square of the) proportionality constant under the root, we can avoid the fractional exponents. It seems that this trick will always work; more care is needed when programming, but perhaps the programs become easier to understand.
The amount of polymorphism can differ, though. The expression (8) in section 4.1 got the type \( \forall \alpha. R(\alpha) \rightarrow \alpha \), but with integer exponents, it only gets the less polymorphic type \( \forall \alpha. R(\alpha) \rightarrow \alpha \). (Expression (8) iterates the square root function infinitely many times when given an argument \( \alpha \). Each iteration halves the dimension exponents of \( \alpha \). The only integer that can be halved arbitrarily many times is 0, and \( \delta^0 = 1 \).) But expression (8) denotes a useless function that never returns a result, and I have not found any meaningful examples of this difference in polymorphism.

Due to my algorithm for dimension inference, the rational exponents have a temporary advantage now, but someone may yet solve the corresponding problem for integers. However, since Baaz’s procedure does not find every AG-semi-unifier, it is not obvious that semi-unification in Abelian groups is unitary.

6 Conclusions

Once we have found proper types, the remaining work of inferring the most general dimension parameters can be done in polynomial time, even under polymorphic recursion. But the existence of proper types is known to be undecidable under polymorphic recursion. If we want automatic type inference always to work, we can disallow polymorphic recursion for the proper types. Personally, I would prefer polymorphic recursion to be allowed generally, to make the language more consistent and expressive. Types that cannot be inferred must be given by the programmer, but the dimension parameters can be omitted. If a type is given without dimension parameters, the compiler should regard them as unspecified. This is important also for backwards compatibility.

Henglein [6, section 6] expects syntactic semi-unification to behave as well in practice as Milner’s type inference [2, 19], even though it is not decidable.

7 Acknowledgments

Andrew Kennedy suggested the problem and inspired me. Fritz and Jean Goubault gave many valuable comments on this paper. Fritz Henglein helped me with the inference system. I also got help from Håkan Andreasson, Lennart Augustsson, Bernard Berthomieu, Thomas Ericsson, A. J. Kfoury, Michael Hedberg, Jacques Huitfeldt, Magnus Lundin, Christina Nystedt, K. V. S. Prasad, Gernot Salzer, Roger Shepherd, and Göran Starius.

References


[7] F. Henglein. Personal communication. (henglein@diku.dk)


