Technical Report

UCAM-CL-TR-391
ISSN 1476-2986

Number 391

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming languages and dimensions

Andrew John Kennedy

April 1996

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 1996 Andrew John Kennedy

This technical report is based on a dissertation submitted November 1995 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St Catherine's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Scientists and engineers must ensure that the equations and formulae which they use are dimensionally consistent, but existing programming languages treat all numeric values as dimensionless. This thesis investigates the extension of programming languages to support the notion of physical dimension.

A type system is presented similar to that of the programming language ML but extended with polymorphic dimension types. An algorithm which infers most general dimension types automatically is then described and proved correct.

The semantics of the language is given by a translation into an explicitly-typed language in which dimensions are passed as arguments to functions. The operational semantics of this language is specified in the usual way by an evaluation relation defined by a set of rules. This is used to show that if a program is well-typed then no dimension errors can occur during its evaluation.

More abstract properties of the language are investigated using a denotational semantics: these include a notion of invariance under changes in the units of measure used, analogous to parametricity in the polymorphic lambda calculus. Finally the dissertation is summarised and many possible directions for future research in dimension types and related type systems are described.

Acknowledgements

I am very grateful to my supervisor Alan Mycroft for his support, encouragement and faith in my ability to complete a thesis. Discussions with many people were a vital spur to research: thanks go to Dave Aspinall, Nick Benton, Gavin Bierman, Francis Davey, Jean Goubault, Barney Hilken, Jean-Pierre Jouannaud, Alan Mycroft, Mikael Rittri, Claudio Rosso, Ian Stark and Mads Tofte. In particular, I would like to thank Francis Davey for the initial tea-room chat which kicked the whole thing off, Nick Benton for the patience to listen to my ramblings and many a productive visit to the pub, and Ian Stark for numerous helpful discussions and example pathological programs. I also appreciate very much Ian and Nick's careful proofreading of drafts of this thesis.

The UK Engineering and Physical Sciences Research Council (formerly the Science and Engineering Council) supported my work financially through a research studentship, and the EU helped fund the final stages through the LOMAPS project.

I am grateful to my parents for their support and encouragement. Finally I would like to thank Lisa for her patience, love and support.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

Preface to Technical Report

Apart from some minor typographical and stylistic changes this technical report differs from the thesis accepted by the examiners in the following ways:

- The proof of correctness of dimension unification (Theorem 3.1) has been simplified substantially.
- The presentation of the type inference algorithm (Figure 3.3) has been improved with corresponding changes to the proofs of soundness and completeness in Appendix C.

In addition, since the dissertation was completed I have shown how the 'change of basis' technique used to calculate Gen (Definition 2.5) is not necessary and that the NGen method suffices. Essentially this involves proving that a property similar to but weaker than free variable reduced form (Section 2.4) is preserved by the inference algorithm. Details will appear in a future article.

As a final note I would like to thank my examiners, Robin Milner and Gordon Plotkin, for their careful reading of the thesis and helpful remarks.

Contents

1 Introduction **1**
- 1.1 Dimensions . . . 2
- 1.2 Types . . . 4
- 1.3 Dimension types . . . 6
- 1.4 Example programs . . . 9
- 1.5 Applications . . . 11
- 1.6 Outline of dissertation . . . 13

2 A dimension type system **15**
- 2.1 Syntax . . . 16
- 2.2 Typing rules . . . 22
- 2.3 Specialisation of type schemes . . . 24
- 2.4 Generalisation with respect to a type assignment . . . 29
- 2.5 Properties of syntax-directed derivations . . . 35
- 2.6 Equivalence of the two formulations . . . 37
- 2.7 Normalising a typing derivation . . . 39

3 Dimension type inference **41**
- 3.1 Unification . . . 41
- 3.2 Inference . . . 47
- 3.3 Simplification of types . . . 50
- 3.4 Related work . . . 61
- 3.5 Some refinements . . . 63

4 Implementation **68**
- 4.1 Syntax . . . 68
- 4.2 A new type . . . 70
- 4.3 New operations . . . 71
- 4.4 Dimension declarations . . . 72
- 4.5 Modules . . . 73

5 Operational semantics **74**
- 5.1 An explicitly-typed language . . . 74
- 5.2 Syntax . . . 76
- 5.3 Operational semantics . . . 80
- 5.4 Semantic soundness . . . 82

5.5 Translation from ML-like language 85

6 Denotational semantics **89**
6.1 Motivation 89
6.2 The denotational semantics 91
6.3 Relating the two semantics 94
6.4 Dimensional invariance 98

7 Dimensional invariance **106**
7.1 Theorems for free 107
7.2 Type inhabitation 110
7.3 Dimensional analysis and type isomorphisms 112
7.4 A more abstract model 115

8 Conclusion **119**
8.1 Further work 119
8.2 Summary 125

A Example derivations **127**

B An algebraic view of dimension types **131**
B.1 Module theory 131
B.2 Dimensions 132
B.3 Types 135
B.4 Type schemes 136

C Omitted proofs **139**

Bibliography **145**

Summary of notation

General

$\mathbb{Z}$	the integers
$\mathbb{N}$	the natural numbers including zero
$\mathbb{B}$	the booleans (true and false)
$\mathbb{Q}, \mathbb{Q}^+$	the rationals, the positive rationals
$\mathbb{R}, \mathbb{R}^+$	the reals, the positive reals
$\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$	finite map
$f[x \mapsto v]$	extension of a finite map
$X \setminus Y$	set difference
b	element of $\mathbb{B}$
i, j	indexing over $\mathbb{N}$
m, n	counting over $\mathbb{N}$
r	element of $\mathbb{R}$ or $\mathbb{Q}$
k	element of $\mathbb{R}^+$ or $\mathbb{Q}^+$
f, g	functions
$\lfloor r \rfloor$	$\max\{\, x \in \mathbb{Z} \mid x \leqslant r \,\}$
$x \bmod y$	remainder: $x - y\lfloor x/y \rfloor$
$x \vert y$	x divides y

Dimensions and types

$\mathbf{1}$	unit dimension	16
B	base dimension (from *DimCons*)	16
d	dimension variable (from *DimVars*)	16
δ	dimension expression	16
$\exp_\delta(d), \exp_\delta(\mathrm{B})$	dimension exponent of d or B in δ	16
$\mathrm{nf}(\delta)$	normal form of a dimension expression	17
t	type variable (from *TyVars*)	17
τ	simple type	17
$\mathsf{bool}, \mathsf{real}\ \delta$	base types	17
$\tau_1 \rightarrow \tau_2$	function type	17
$\forall t.\tau$	type polymorphism	17
$\forall d.\tau$	dimension polymorphism	17
σ	type scheme	17
v	type or dimension variable	18
$\vec{v}$	list of type or dimension variables	18

Substitutions and free variables

R, S, T	substitutions	18
I	the identity substitution	18
U	invertible substitution	18
$\mathrm{dom}(S)$	domain of a substitution	18
$\mathcal{V}$	set of dimension or type variables	19
$S\|_{\mathcal{V}}$	restriction of S to the variables in $\mathcal{V}$	19
fdv	free dimension variables	19
ftv	free type variables	20
fv	free type and dimension variables	20

Expressions

x, y, z	variables (from *Vars*)	21
e	expression	21
$\lambda x.e$	abstraction	21
$\mathsf{if}, \mathsf{let}$, *etc.*	reserved words	21
$\lambda x : \tau.\ e$	typed abstraction	79
$\Lambda d.e$	dimension abstraction	79

Type systems

Γ	type assignment	22
$\Gamma \vdash e : \tau$	typing judgment in ML_δ	22
$\Gamma \vdash_{\mathrm{nsd}} e : \sigma$	non-syntax-directed derivation	22
$\Gamma \vdash_{\mathrm{sd}} e : \tau$	syntax-directed derivation	22
$\mathrm{Gen}(\Gamma, \tau)$	generalisation of a type w.r.t. an assignment	29
$\mathrm{NGen}(\Gamma, \tau)$	naive generalisation	29
$\mathcal{V}; \Gamma \vdash e : \tau$	typing judgment in Λ_δ	77
$\Gamma \vdash e \rightsquigarrow e' : \tau$	translation from ML_δ to Λ_δ	86

Relations

$=_D$	dimension equivalence	16
$\preccurlyeq_D$	type scheme specialisation and ordering	24
$\cong_D$	type scheme equivalence	27

Operational semantics

v	value	80
E	environment	80
$E \vdash e \Downarrow v$	evaluation relation	80
$\langle r, \delta \rangle$	dimensioned real	80
$\langle E, \lambda x \colon \tau.\, e \rangle$	function closure	80
$\langle E, \mathsf{rec}\, y(x \colon \tau_1) \colon \tau_2.\, e \rangle$	recursive function closure	80
$\mathrm{Val}(\mathcal{V}, \tau)$	values of type τ with variables $\mathcal{V}$	83
$\mathrm{Env}(\mathcal{V}, \Gamma)$	environments matching Γ with variables $\mathcal{V}$	83
$\mathrm{Expr}(\mathcal{V}, \Gamma, \tau)$	expressions of type τ under Γ and $\mathcal{V}$	83
$\tau^\star, e^\star$, *etc.*	dimension erasure	83

Denotational semantics

$D_\bot, f_\bot, R_\bot$	lifted domains, functions, relations	91
$D \to_c D'$	continuous functions between D and D'	91
w	value	92
ρ	environment	92
$[\![\tau]\!]^\star$	domain of values of type τ	91
$[\![\Gamma]\!]^\star$	domain of environments matching Γ	92
$[\![e]\!]^\star(\rho)$	meaning of expression e in environment ρ	92
$\mathcal{C}[\phi]$	program context	94
$\approx$	contextual equivalence	94
$\mathcal{A}^{\mathcal{V}}_{\tau}, \mathcal{A}^{\mathcal{V}}_{\Gamma}$	logical relations between two semantics	96
ψ	scaling environment	99
ψ^{-1}	pointwise inverse of scaling environment	99
$\mathcal{R}^{\psi}_{\tau}, \mathcal{R}^{\psi}_{\Gamma}$	scaling relations	100
$\Phi^{\psi}_{\tau}, \Phi^{\psi}_{\Gamma}$	scaling maps	107
$\cong$	type isomorphism	114
$\mathcal{R}^{\mathcal{V};\Gamma}_{\tau}$	partial equivalence relation over meanings	116
$[f]^{\mathcal{V};\Gamma}_{\tau}$	equivalence class in PER	117
$[\![e]\!]$	parametric meaning of expression e	117

Chapter 1

Introduction

Dimensions are to science what types are to programming. In science and engineering, dimensional consistency provides a first check on the correctness of an equation or formula, just as in programming the typability of a program or program fragment eliminates one possible reason for program failure.

What then of dimensions in programming? After all, the writing of a scientific program involves turning a set of equations which model some physical situation into a set of instructions making up a computer program. It still makes sense to talk about the dimensions of a formula, even when represented as a program fragment written in some programming language.

Modern programming languages are *strongly-typed*: this prevents errors such as the addition of a string to a real number, or the passing of too few or too many arguments to a function or procedure. But there are many errors which a conventional type system cannot catch:

- Attempting to index an array outside its bounds. In most languages, this results in a run-time error, perhaps safely raising an exception (as in Standard ML), or simply leading to undefined behaviour (as in C).
- Passing a negative number to a function which expects a non-negative argument, such as square root. Usually it is not possible for a compiler to determine statically when some value is negative.
- Passing two lists of different lengths to a function which expects lists of the same length, such as `zip` in a functional language.

Finally, there are dimension errors:

- The addition, subtraction or comparison of two numbers which represent quantities with different dimensions.

Existing programming languages, even those with secure type systems, treat all numeric quantities as dimensionless. The aim of this thesis is to show how the type system of a programming language can be modified to support the notion of physical dimension.

This chapter introduces some necessary concepts from science and from the study of programming languages, and discusses the issues involved in taking the idea of *dimension* from science and incorporating it into the notion of *type* in programming. The rest of the dissertation is described in outline.

This introduction also contains most of the example programs used in this dissertation, in order to motivate the more formal study of dimensions and programming languages in subsequent chapters. All these programs are written in Standard ML [43, 51], and a subset of this language forms the basis for theoretical investigations. Clearly all of the ideas would apply to any functional language in the style of ML: these include Haskell, Miranda and Hope. They could also be applied to other language paradigms—in fact, to any language with a secure type system. There are the usual intricacies of assignment in the presence of polymorphism to deal with (considered by Tofte amongst others [64]), but dimensions do not introduce any new problems here.

1.1 Dimensions

Units *vs* dimensions

Physical quantities are measured with reference to a *unit of scale*. When we say that something is '6 metres long' we mean that six metre-lengths placed end-to-end would have the same length. The unit 'metre' is acting as a point of reference, not just for the purpose of *comparison* (X is longer than a metre), but also for *measurement* (X is six times as long as a metre). Scales of measurement are usually linear but need not be so: amplitudes are often measured in decibels, with reference to some fixed amplitude. There may even be a 'false origin', as with measurements of temperature in degrees Celsius or degrees Fahrenheit.

A single quantity may be measured in many different systems of units. Crucially, though, all the units are interconvertible—if X can be measured with reference to a unit U, and X can also be measured with reference to a unit V, then U can be related to V. This leads to the notion of physical *dimension*, which is an abstraction of the idea of units of measure. In a sense, the dimension of some quantity represents all possible units in which the quantity can be measured: it is a *class of similar scales*. Quantities which have the same dimension can usually be compared meaningfully; quantities with different dimensions cannot. Moreover, the former can be compared *quantitatively* by determining their ratio. One colour can be compared to another, and may be described as different, lighter, darker, or whatever, but 'colour' is not a dimension. On the other hand, 'light intensity' certainly is a dimension, because it can be expressed as the product of a numeric value and a unit of scale.

Base *vs* derived

It is usual in science to fix a set of *base dimensions* which cannot be defined in terms of each other. 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 then defined in terms of existing dimensions by product and quotient operations; for example, acceleration is distance divided by time squared. Dimensions are conventionally written in an algebraic form inside square brackets [35], so the dimensions of force are written $[\mathrm{MLT}^{-2}]$, for example.

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 ($\mathrm{kg\ m\ s}^{-2}$).

Of course, this division of dimensions and units into base and derived is arbitrary, and one could easily work with, say, force, acceleration and velocity instead of mass, length and time. In some circumstances it even makes sense to adopt a more refined set of base dimensions, for example to use three different dimensions for length measured along three different axes.

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—after all, angles and solid angles are expressed in 'units' of radians and steradians respectively. Nevertheless, they are just ratios: an angle is the ratio of two lengths (distance along an arc divided by the radius it subtends) and a solid angle is the ratio of two areas (surface area on a sphere divided by the square of the radius).

Dimensional consistency

Quantities with the same dimension may be added, subtracted and compared, but quantities with different dimensions may not. In contrast, quantities of any dimension may be multiplied or divided, giving a result whose dimension is the product or quotient of the dimensions of the two quantities. Thus the sum of values with dimensions velocity $[\mathrm{LT}^{-1}]$ and time $[\mathrm{T}]$ is a dimension error, whereas their product has dimension length $[\mathrm{L}]$.

If an expression is free of dimension errors, then it is said to be *dimensionally consistent*. For scientists and engineers, the dimensional consistency of a formula or equation is a very handy check on its correctness, for dimensional inconsistency certainly indicates that something is amiss.

On the other hand, it is not necessarily true that two quantities of the same dimension can meaningfully be compared. For example, it usually does not make sense to compare a coefficient of restitution with a solid angle, or to compare torque with energy, both of which have the dimension $[\mathrm{ML}^2\mathrm{T}^{-1}]$.

Dimensional analysis

An equation is said to be *dimensionally invariant* [33, 34] or *unit-free* [6, 39] if it is invariant under changes in the units of measure used. We have already seen how a dimension can be seen as a class of units, and this interpretation leads directly to the fact that dimensionally consistent equations are dimensionally invariant. Often the two terms are not even distinguished, and the phrase *di-*

mensionally homogeneous is used to refer to either or both. Note, however, that a dimensionally inconsistent equation may be dimensionally invariant in a trivial way. Taking v to stand for velocity and t for time, the equation $(v-v)(v+t) = 0$ is clearly wrong dimensionally, but is valid under any system of units, and hence is dimensionally invariant.

Philosophically, it is a profound fact that physical laws are dimensionally invariant: why should they have the same form at all scales? Leaving the philosophy aside, the *assumption* that physical laws work at all scales leads to a very useful technique called *dimensional analysis*. The idea is simple: when investigating some physical phenomenon, if the equations governing the phenomenon are not known but the parameters are known, one can use the dimensions of the parameters to narrow down the possible form the equations may take. For example, consider investigating the equation which determines the period of oscillation t of a simple pendulum. Possible parameters are the length of the pendulum l, the mass m, the initial angle from the vertical θ and the acceleration due to gravity g. After performing dimensional analysis it is possible to assert that the equation must be of the form $t = \sqrt{l/g}\,\phi(\theta)$ for some function ϕ of the angle θ. Of course it turns out that for small angles $\phi(\theta) \approx 2\pi$, but dimensional analysis got us a long way—in particular, the period of oscillation turned out to be independent of the mass m.

In its general form, this is known as the *Pi Theorem*, which states that any dimensionally consistent equation over several variables can be reduced to an equation over a smaller number of dimensionless terms which are products of powers of the original variables. The books by Birkhoff [6] and by Langhaar [35] are good references for this subject; unlike many other authors they give a proper mathematical treatment to the problem of dimensional analysis.

1.2 Types

Static types and type checking

Programmers make mistakes. Therefore the process of writing software is improved if the compiler can catch some of these mistakes before a program is run. This is one purpose of a *static* type system. Every identifier and expression in the program is assigned some *type*, the types of larger expressions being determined by the types of their sub-expressions according to certain rules. Usually the rules insist that types of certain sub-expressions must match up, and this leads to the idea that some programs are *ill-typed* and must be rejected by the compiler.

In imperative languages, one rule states that in an assignment command of the form $x := e$ the type of the expression e must match the type of the variable x. Furthermore, the types of parameters passed to procedures must match the types of the formal parameters in the procedure's definition. In functional languages, the rule for function application is crucial: an expression $e_1(e_2)$ is given a type τ_2 only if the type of e_1 is a function type of the form $\tau_1 \to \tau_2$ and the argument e_2 has type τ_1.

A properly-designed static type system results in a *type-safe* language, one in which "well-typed programs do not go wrong" [42]; that is, execution of a well-typed program in the language cannot lead to a type error.

Polymorphism

The danger with insisting on a safe, static type system is that the programming language may be too restrictive as a result. This is not just a problem of recalcitrant programmers: a restrictive type system can inhibit *code re-use*. A good example is the language Pascal. Its type system is almost safe, and its single type loophole (variant records) is the *only* way in which generic code may be written, for example to implement linked lists for any element type.

An unfortunate consequence of the restrictive nature of static type systems of the past is that many programmers and programming language designers are unaware of the solution: *parametric polymorphism.* Instead of a function or procedure having a single type, it has many. Moreover, all of its types can be encapsulated in a single *type scheme* which contains type variables acting as placeholders for other types. The function can then be re-used in many ways simply by substituting different types for its type variables.

Type inference

A disadvantage of static type systems of the Algol/Pascal variety is that programs must be 'decorated' with type declarations, one for each new identifier introduced in the program. Programming languages with *dynamic* type systems, such as LISP, do without such declarations and tend to be seen by programmers as more interactive and more suitable for prototyping. Milner's seminal paper on type polymorphism [42] showed how to have one's cake and eat it too: by means of a type inference algorithm a valid type can be deduced for any type-correct program without the need for type declarations in the program.

Types as properties

Types have another role: they *inform.* Naturally, how informative they are depends on the expressiveness of the type system.

The type of `qsort` in C, for instance, just tells us that it needs a 'pointer to anything', a couple of integers, and a strange function which takes two 'pointers to anything' as argument and returns an integer as result. But in Standard ML, the same function would have the type

```
val qsort : ('a*'a -> bool) -> 'a list -> 'a list
```

which immediately tells us that the elements of the list argument must have the same type as both arguments to the function. Furthermore, the polymorphic type of a function can say something about the function's *behaviour.* For instance, suppose that some function written in Standard ML has the type

```
val f : 'a list -> int
```

Then we know from this type that `f` cannot possibly 'look at' the values in the list passed as an argument. This can be expressed formally by showing that the mapping of another function down the list before passing it as argument does not affect the value of the integer which is returned.

1.3 Dimension types

The previous two sections have illustrated the clear analogy between dimension-checking of mathematical formulae and type-checking of programs. The connection is so obvious that already there have been many proposals for dimension checking in programming languages [29, 24, 16, 40, 15, 3] including two along similar lines to the system discussed here [68, 18]. There has even been some work on applying dimension checking to formal specification of software [22]. Nevertheless, some of the usual arguments in favour of static type checking do not apply to dimensions. First, there is no way in which a dimension error can have unpredictable, catastrophic consequences similar to those of a conventional type error, such as treating an integer as though it were a pointer. Secondly, unless dimension checking is already present in the run-time system, static dimension checking does not catch any existing run-time errors, such as would be avoided by static checking of array indexing, for instance. Instead, dimension checking aims to avoid *wrong answers* by ruling out programs which make no sense dimensionally.

Dimension types and dimension checking

The basic idea common to all proposals is the following: parameterise numeric types on a dimension or unit of measure. The form that this takes varies. Some choose to fix a set of base dimensions and then express a dimension such as $[\mathrm{MLT}^{-2}]$ as a vector of exponents $(1, 1, -2)$. Others are more flexible and let the programmer choose the base dimensions, combining these to form derived dimensions in the traditional algebraic style. Three questions arise:

1. Which types are parameterised?
2. Are they parameterised on dimensions or units?
3. Are exponents of base dimensions integers or rationals?

In answer to the first question, all systems allow the type of floating-point numbers (loosely referred to as reals) to be parameterised on a dimension or unit. Some also allow other types, such as complex numbers, to have a dimension, and more sophisticated systems extend this to user-defined types too. For the moment, we will discuss only real numbers, and write $\mathsf{real}\,\delta$ to indicate the type of reals with dimension δ.

The choice of dimensions or units is mostly a matter of taste. The relations which hold between dimensions also hold between the units in which those dimensions are measured, so for the purposes of checking consistency it does not matter which concept we choose. (Of course this assumes that the units are linear with origin at zero—it makes no sense to add two amplitudes measured in decibels or to double a temperature measured in degrees Celsius). In a sense, a dimension is an *abstract data type* which 'hides' the actual units used (it is a *class* of units), and choosing to parameterise quantities on a particular unit of measure makes this representation explicit. Hence it is only necessary to be aware of the distinction if the language permits multiple systems of units for a single dimension and converts automatically between different systems when appropriate. In the languages which we study we assume some global unit of measure for each base dimension; therefore we use the phrase *dimension type* to refer to types such as $\mathsf{real}\,\delta$. Sometimes, though, we will use the units-of-measure interpretation to give intuition to the ideas which we present.

The most important decision is whether or not to allow fractional exponents of dimensions. The argument against them is philosophical: a quantity with a dimension such as $[\mathrm{M}^{\frac{1}{2}}]$ makes no sense physically, and if such a thing arose, it would suggest revision of the set of base dimensions rather than a re-evaluation of integral exponents. The argument in favour is pragmatic: sometimes it is easier to write program code which *temporarily* creates a value whose dimension has fractional exponents. In this dissertation the former view prevails, and fractional exponents are not considered. However, most of the theory would apply just the same; any potential differences are highlighted as they arise.

For the examples which follow, we will adopt the following notation which is defined formally in Chapter 2. Base dimensions are alphabetic identifiers, usually a single upper-case letter such as M (for mass) or L (for length). In a real programming language, these would be predeclared to the compiler. Then dimension expressions are formed by exponentiation, such as L^3 (for volume). Dimension products are written explicitly using '$\cdot$', such as $\mathrm{M}\cdot\mathrm{L}^{-3}$ (for density). Finally, dimensionless quantities are written using the symbol $\mathbf{1}$, so the type of dimensionless reals is then $\mathsf{real}\,\mathbf{1}$.

Polymorphism

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; in fact, even something as simple as a squaring function requires this. Of the proposals for dimension checking in Pascal-like languages, only House recognised this need for polymorphism [24]. In this thesis we take a more modern view of polymorphism: the parametric polymorphism adopted by languages such as ML and Haskell. Hence the syntax of dimension expressions is extended with *dimension variables*, written d, d_1, d_2, etc. Then the type of a squaring function `sqr` would be expressed by $\mathsf{real}\,d \to \mathsf{real}\,d^2$. This says that `sqr` accepts an argument with *any* dimension d and returns a result which has dimension d^2.

Arithmetic

With dimension polymorphism, the types of built-in arithmetic and comparison functions can be expressed directly in the system, as follows:

$$\begin{aligned}
\texttt{+}, \texttt{-} &: \mathsf{real}\, d \times \mathsf{real}\, d \rightarrow \mathsf{real}\, d \\
\texttt{*} &: \mathsf{real}\, d_1 \times \mathsf{real}\, d_2 \rightarrow \mathsf{real}\, d_1 \cdot d_2 \\
\texttt{/} &: \mathsf{real}\, d_1 \times \mathsf{real}\, d_2 \rightarrow \mathsf{real}\, d_1 \cdot d_2^{-1} \\
\texttt{sqrt} &: \mathsf{real}\, d^2 \rightarrow \mathsf{real}\, d \\
\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=} &: \mathsf{real}\, d \times \mathsf{real}\, d \rightarrow \mathsf{bool}
\end{aligned}$$

Trigonometric and logarithm functions are dimensionless:

$$\texttt{exp}, \texttt{ln}, \texttt{sin}, \texttt{cos}, \texttt{tan} : \mathsf{real}\, \mathbf{1} \rightarrow \mathsf{real}\, \mathbf{1}$$

We will sometimes want to convert integers into dimensionless reals:

$$\texttt{real} : \mathsf{int} \rightarrow \mathsf{real}\, \mathbf{1}$$

Constants

Numeric constants are dimensionless, *except* for zero, which can have any dimension. 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 < 0.0 then 0.0 - x else x
```

This has type $\mathsf{real}\, d \rightarrow \mathsf{real}\, d$. Polymorphic zero is also essential as an identity for addition in functions such as the following:

```
fun sum []      = 0.0
  | sum (x::xs) = x + sum xs
```

This has the polymorphic dimension type $\mathsf{real}\, d\ \mathsf{list} \rightarrow \mathsf{real}\, d$, where $\tau\ \mathsf{list}$ is the type of lists with elements of type τ.

It is assumed that for each base dimension B introduced, a value of type $\mathsf{real}\,\mathrm{B}$ is available which acts as the default unit of measure associated with that dimension. Other values of type $\mathsf{real}\,\mathrm{B}$ are then constructed by multiplying this unit by a dimensionless numeric constant.

Note that if *non*-zero numeric constants were polymorphic in dimension then a value of some dimension could be coerced into a value of any other dimension simply by multiplying by a polymorphic constant, and this could be used to break dimensional consistency.

Dimension inference

Having to decorate programs with types is an inconvenience. Having to decorate all numeric types with dimensions, even when one has no particular dimension in mind, is even more of a nuisance. Fortunately it is possible for the compiler

to infer polymorphic dimension types just as ML infers ordinary types, and as for ML the inferred type is *most general* in the sense that any valid type can be derived by substituting types and dimensions for its type and dimension variables. The types of all of the example programs in this introduction were deduced by the inference algorithm which is presented later.

Dimension types as scaling properties

As with conventional types, polymorphic dimension types say something about the behaviour of a function. In this case, we can deduce certain properties of functions with respect to *scaling*, that is, multiplying values of type $\mathsf{real}\ \delta$ by some positive scale factor—in effect, changing the units of measure used. For example, knowing that a function has the polymorphic type $\mathsf{real}\,d \times \mathsf{real}\,d \to \mathsf{real}\,d$ tells us that if the two arguments to the function are both scaled by the same multiplier $k > 0$, then the result of applying the function to these arguments will scale accordingly.

1.4 Example programs

In this section the idea of dimension types is illustrated by a series of example programs. They are simple from the numerical analyst's point of view, but they serve to demonstrate the utility of dimension types whilst being small enough to understand at a glance.

Statistics

Statistics is a rich source of examples, and most of them would commonly be used with quantities of many different dimensions, even in the same program. Here are some Standard ML functions which calculate the mean, variance and standard deviation of a list of dimensioned quantities:

```
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) / real(n - 1)
  end

fun sdeviation xs = sqrt (variance xs)
```

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

$$\begin{aligned}
\texttt{mean} &: \textsf{real}\ d\ \textsf{list} \to \textsf{real}\ d \\
\texttt{variance} &: \textsf{real}\ d\ \textsf{list} \to \textsf{real}\ d^2 \\
\texttt{sdeviation} &: \textsf{real}\ d\ \textsf{list} \to \textsf{real}\ d \\
\texttt{skewness} &: \textsf{real}\ d\ \textsf{list} \to \textsf{real}\ \mathbf{1} \\
\texttt{correlation} &: \textsf{real}\ d_1\ \textsf{list} \to \textsf{real}\ d_2\ \textsf{list} \to \textsf{real}\ \mathbf{1}
\end{aligned}$$

Differentiation and integration

We can write a function which differentiates another function numerically, using the formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

The ML code for this accepts a function `f` and increment `h` as arguments and returns a new function which is an approximation to the derivative of `f`:

```
fun diff(h,f) = fn x => (f(x+h) - f(x-h)) / (2.0 * h)
```

This has the type

$$\textsf{real}\ d_1 \times (\textsf{real}\ d_1 \to \textsf{real}\ d_2) \to (\textsf{real}\ d_1 \to \textsf{real}\ d_2 \cdot d_1^{-1}).$$

Unlike the statistical examples, the type of the result is related to the type of more than one argument. It also illustrates the expressiveness of higher-order functions.

We can likewise integrate a function, using one of the simplest methods: the trapezium rule [53, Section 4.1]. It is defined by the following formula, which gives an approximation to the area under the curve defined by f in the interval $a \leqslant x \leqslant b$ using $n+1$ values of $f(x)$:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left(f(a) + 2f(a+h) + \cdots + 2f(b-h) + f(b)\right), \quad h = \frac{b-a}{n}.$$

Its implementation in ML is shown below:

```
fun integrate(f, a, b, n) =
  let val h = (b-a) / real(n)
      fun iter(x, 0) = 0.0
        | iter(x, i) = f(x) + iter(x+h, i-1)
  in
    h * (f(a) / 2.0 + iter (a+h, n-1) + f(b) / 2.0)
  end
```

This has the type

$$(\textsf{real}\ d_1 \to \textsf{real}\ d_2) \times \textsf{real}\ d_1 \times \textsf{real}\ d_1 \times \textsf{int} \to \textsf{real}\ d_1 \cdot d_2.$$

Root finding

Below is a tiny implementation of the Newton-Raphson method for finding roots of equations, based on the iteration of

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

This method calculates a solution of $f(x) = 0$, making use of the derivative f'.

The ML code is as follows:

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

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

$$(\mathsf{real}\ d_1 \to \mathsf{real}\ d_2) \times (\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^{-1} \cdot d_2) \times \mathsf{real}\ d_1 \times \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ d_1.$$

Powers

To illustrate a more unusual type, here is a function of three arguments x, y and z:

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

This has the polymorphic type

$$\mathsf{real}\ d^{15} \times \mathsf{real}\ d^{6} \times \mathsf{real}\ d^{5} \to \mathsf{real}\ d^{30}.$$

The unusually high powers are forced by the addition of x^2 to y^5 to z^6, and whilst such a function is unlikely to turn up in a real program, it does illustrate the point that dimension inference must do 'more work' than conventional type inference, in this case calculating the lowest common multiple of 2, 5 and 6.

1.5 Applications

The examples just discussed typify the kind of programs that scientists and engineers write, and the application of dimension checking to these is obvious. However, there are many other applications which do not necessarily involve 'physical' quantities but would still benefit from dimensions.

Computer graphics is one such field. It depends heavily on algorithms from computational geometry; these involve the dimension *length* and its derivatives.

Three-dimensional visualisation applies the physics of light to obtain realistic images which simulate real-world scenes. The relevant dimension here is that of light *intensity*, and if the images were animated, then the dimension of *time* would be required too.

It is sometimes remarked that database systems would benefit from a dose of type theory, and much progress has been made recently on typed programming languages for database query and update [61]. Dimension types would be even better. The 'dimension' of commerce is *money*, so the salaries stored in a commercial database would have the dimension *money/time*. All the usual physical dimensions have their use too: how about electricity unit price (derived dimension: *money/energy*), price of coal (dimension: *money/mass*) and cost of land (dimension: $money/length^2$)?

Even systems programming, a pure 'computer science' application, might benefit from dimension checking. Operating system code deals routinely with values of disc capacity (units: bytes per sector), CPU time (units: seconds), and data transfer rate (units: bits per second).

To end this section, here is a real-life example of a bug which could have been picked up by a dimension type system that required consistent use of units.

> "Much to the surprise of Mission Control, the space shuttle Discovery flew upside-down over Maui on 19 June 1985 during an attempted test of a Star-Wars-type laser-beam missile defense experiment. The astronauts reported seeing the bright-blue low-power laser beam emanating from the top of Mona Kea, but the experiment failed because the shuttle's reflecting mirror was oriented upward! A statement issued by NASA said that the shuttle was to be repositioned so that the mirror was pointing (downward) at a spot *10,023 feet* above sea level on Mona Kea; that number was supplied to the crew in units of feet, and was correctly fed into the onboard guidance system – which unfortunately was expecting units in nautical miles, not feet. Thus the mirror wound up being pointed (upward) to a spot *10,023 nautical miles* above sea level…"†

Clearly this was really a problem with the human-computer interface, but it is not hard to imagine the same situation resulting from a pure programming error.

†Reproduced from ACM SIGSOFT Software Engineering Notes vol. 10 no. 3, July 1985, page 10

1.6 Outline of dissertation

The remainder of this dissertation consists of the following chapters.

Chapter 2: A dimension type system. The type system of a very small language with dimension types is presented. Two formulations of the typing rules are given: a set which includes explicit rules for generalisation and specialisation of type schemes, and a set of *syntax-directed* rules where the structure of the expression matches the structure of its typing derivations. The system differs from conventional ML-like type systems by the presence of *equations* between dimensions. The particular set of equations required—those of an Abelian group—lead to some interesting challenges in setting up a syntax-directed system, in particular the generalisation of a type with respect to a type assignment.

The two formulations are proved to be equivalent.

Chapter 3: Dimension type inference. Using the syntax-directed type system as a basis, this chapter presents a dimension type inference algorithm. As for ordinary ML, if an expression is typable at all then there is a single *most general* type which generates all possible types by substituting for type and dimension variables.

The algorithm differs from the standard one in its employment of *equational unification*: in this case, unification over Abelian groups. A unification algorithm and its proof of correctness is presented.

After discussing some related type systems, the chapter concludes with some refinements to the type system: the problem of finding a canonical form in which to display types, the possibility of providing polymorphism in recursive definitions, and some limitations of ML-style implicit polymorphism.

Chapter 4: Implementation. In this chapter we depart temporarily from formalism and discuss a working implementation of Standard ML extended with dimension types. The extension is *conservative* in the sense that existing ML programs will type-check and run unchanged, but may be given more refined types.

Chapter 5: Operational semantics. The formal semantics of the ML-like language of Chapters 2 and 3 is given by translating it into an explicitly-typed language in the style of System F. The semantics of *this* language is then given by a set of rules which define a big-step, evaluation relation.

The operational semantics is used to prove that “well-dimensioned programs do not go wrong” in the sense that dimension errors cannot occur at run-time.

Chapter 6: Denotational semantics. The operational semantics of the previous chapter is too clumsy to investigate more abstract properties of programs with dimension types, and to this end a denotational semantics

based on complete partial orders is developed. The agreement of the denotational and operational semantics is proved (an *adequacy* result).

Then a *logical relation* over this semantics is defined and used to prove a theorem analogous to parametricity results for polymorphically-typed languages. In essence, this theorem expresses the *dimensional invariance* of programs: that their behaviour is independent of the units of measure used.

Chapter 7: Dimensional invariance. One consequence of the dimensional invariance theorem is that for every dimension type there is an associated 'scaling theorem' valid for expressions of that type. Another application is in proving that there are certain types which are not possessed by any non-trivial term—for example, a dimensionally-polymorphic 'square root' function cannot be written in terms of the usual arithmetic primitives. Finally it is shown how the Pi Theorem from the theory of dimensional analysis may have an analogue in programs with dimension types.

Chapter 8: Conclusion. The dissertation is summarised, and directions for further research in dimension types and related type systems are suggested.

Chapter 2

A dimension type system

In this chapter we make formal the idea of a dimension type system by considering a small fragment of an ML-like language with dimension types. We call this language ML_δ. It differs from conventional ML in two ways. First, in addition to type polymorphism it supports dimension polymorphism, formalised by type schemes which can quantify over dimensions as well as types. Second, implicit equations hold between dimension expressions (and hence types) which are determined by the algebraic properties of dimensions.

We start by introducing the syntax of the language, and give some necessary definitions of concepts such as type equivalence, substitution and free variables. Then a set of typing rules is given, similar to those for ordinary ML but with additional rules to support dimension types.

In order to study an inference algorithm it is easier to work with a *syntax-directed* variant of the rules. Because of the equational theory imposed on dimensions and types, the setting up of such a system is far from trivial. The definitions of standard concepts (such as an ordering on type schemes, free variables in a type scheme, and the generalisation of a type with respect to a type assignment) are more subtle than usual. New concepts are required too, such as a special property of the free variables in a type assignment that is required for generalisation to work correctly. Having prepared the ground with a series of lemmas concerning the syntax-directed type system we prove that the two formulations are equivalent.

The system described in this chapter and its inference algorithm have previously appeared in a slightly different form [30].

2.1 Syntax

Dimensions

The syntax of *dimension expressions* (ranged over by δ) has the following grammar:

$$\begin{array}{rcll} \delta & ::= & d & \textit{dimension variables} \\ & | & \mathrm{B} & \textit{base dimensions} \\ & | & \mathbf{1} & \textit{unit dimension} \\ & | & \delta_1 \cdot \delta_2 & \textit{dimension product} \\ & | & \delta^{-1} & \textit{dimension inverse} \end{array}$$

Here d ranges over an infinite set of dimension variables *DimVars* and B ranges over a set of base dimensions *DimCons*. These sets are disjoint.

Dimensions satisfy certain algebraic properties, namely those of an Abelian group whose operation is dimension product. For example, the dimensions $\mathrm{M} \cdot \mathrm{L}$ and $\mathrm{L} \cdot \mathrm{M}$ are interchangeable, and $\mathrm{M} \cdot \mathrm{M}^{-1}$ is equivalent to the unit dimension $\mathbf{1}$. Hence we require a notion of dimension *equivalence*. Formally, we define $=_D$ to be a congruence on dimension expressions generated by the following equations:

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

Then the set of all dimension expressions quotiented by this equivalence forms a free Abelian group.

We now define exponentiation of dimensions δ^n for any integer n:

$$\delta^n = \begin{cases} \delta \cdot \ \cdots \ \cdot \delta & (n \text{ dimensions}) \text{ if } n > 0, \\ \mathbf{1} & \text{if } n = 0, \\ \delta^{-1} \cdot \ \cdots \ \cdot \delta^{-1} & (-n \text{ dimensions}) \text{ if } n < 0. \end{cases}$$

With this operation as ‘scalar multiplication’, the set of dimensions can be treated as a vector space over the integers, or more properly, a free $\mathbb{Z}$-*module* [1, 7]. This view of dimensions is discussed in more detail in Appendix B.

Next we introduce the notion of a *normal form* for dimension expressions. A dimension expression is *normalised* if it is in the following form:

$$d_1^{x_1} \cdot d_2^{x_2} \cdots d_m^{x_m} \cdot \mathrm{B}_1^{y_1} \cdot \mathrm{B}_2^{y_2} \cdots \mathrm{B}_n^{y_n}, \quad x_i, y_j \in \mathbb{Z} \setminus \{0\},\ d_i \text{ and } \mathrm{B}_j \text{ distinct.}$$

This assumes some total ordering on the sets used for dimension variables and base dimensions. When giving concrete examples we will not worry about this, but a real implementation such as the one discussed in Chapter 4 should be consistent in the way it displays dimensions to the user.

Alternatively, the normal form can be defined using a function which extracts the exponent of each variable or constant in a dimension expression. This is

defined inductively on the structure of the expression as follows:

$$\begin{array}{rcl}
\exp_\delta & : & \mathit{DimVars} \cup \mathit{DimCons} \to \mathbb{Z} \\
\exp_d(v) & = & \begin{cases} 1 & \text{if } v = d, \\ 0 & \text{if } v \neq d. \end{cases} \\
\exp_\mathrm{B}(v) & = & \begin{cases} 1 & \text{if } v = \mathrm{B}, \\ 0 & \text{if } v \neq \mathrm{B}. \end{cases} \\
\exp_\mathbf{1}(v) & = & 0 \\
\exp_{\delta_1 \cdot \delta_2}(v) & = & \exp_{\delta_1}(v) + \exp_{\delta_2}(v) \\
\exp_{\delta^{-1}}(v) & = & -\exp_\delta(v)
\end{array}$$

Fact. *Normal forms are unique:* $\delta_1 =_D \delta_2$ *if and only if* $\exp_{\delta_1} = \exp_{\delta_2}$.

We will use the notation $\mathrm{nf}(\delta)$ to denote the normal form of the dimension expression δ.

Types

The polymorphic types used in the introductory chapter were *implicitly quantified*, meaning that any type variable or dimension variable in the type of an expression was assumed to be universally quantified. In order to define a set of formal rules for type inference, it is necessary to distinguish between *simple types*, in which all type and dimension variables appear free, and *type schemes*, in which some variables are bound by explicit universal quantifiers. This is reflected in the syntax: simple types (ranged over by τ) and type schemes (ranged over by σ) are separate syntactic classes. We will use the word *type* to refer to both simple types and type schemes.

Simple types are defined by the following grammar:

$$\begin{array}{rcll}
\tau & ::= & t & \textit{type variables} \\
 & | & \mathsf{bool} & \textit{booleans} \\
 & | & \mathsf{real}\ \delta & \textit{dimensioned reals} \\
 & | & \tau_1 \to \tau_2 & \textit{function types}
\end{array}$$

Here t ranges over an infinite set of type variables *TyVars*. The types are deliberately limited to just booleans, dimensioned reals and functions. Even such a small fragment of a practical type system captures the essence of polymorphic dimension type inference, and its extension to other base types, tuples, lists and recursive data types would be straightforward. In Chapter 4 we describe the implementation of an extension to the Standard ML language using the ML Kit compiler.

Type schemes extend simple types with explicit quantification over type and dimension variables. Their syntax is defined below:

$$\begin{array}{rcll}
\sigma & ::= & \tau & \textit{types} \\
 & | & \forall t.\sigma & \textit{type quantification} \\
 & | & \forall d.\sigma & \textit{dimension quantification}
\end{array}$$

For conciseness we will use v to stand for a type or dimension variable, and write $\vec{v}$ to indicate a list of type and dimension variables. Then type schemes all have the general form $\forall\vec{v}.\tau$. Often we will abuse this notation and treat $\vec{v}$ as a set, making statements such as $v \in \vec{v}$ to mean that "the variable v is in the list of variables $\vec{v}$."

The equivalence relation introduced for dimension expressions is extended to simple types by the obvious congruence. For type schemes, we further allow renaming of bound type and dimension variables (alpha-conversion).

Substitution

A dimension substitution S is a map from dimension variables to dimension expressions. This extends to a homomorphic map between dimension expressions in the natural way, and to types which may have dimension components. Composition of substitutions is defined as usual, and the identity substitution is denoted by I. The *domain* of a dimension substitution is the set of variables which it modifies with respect to dimension equivalence:

$$\mathrm{dom}(S) \stackrel{\text{def}}{=} \{\, d \in \mathit{DimVars} \mid S(d) \neq_D d \,\}.$$

We will write $\{d_1 \mapsto \delta_1, \ldots, d_n \mapsto \delta_n\}$ to stand for a particular substitution whose domain is $\{d_1, \ldots, d_n\}$, or occasionally, the more concise $\{\vec{d} \mapsto \vec{\delta}\}$.

A type substitution S is a map from type variables to type expressions. Again, this extends to a map between type expressions, with composition of substitutions and domain of a substitution again defined in the standard way.

A general substitution S is a pair (S_d, S_t) consisting of a dimension substitution S_d and a type substitution S_t. Its interpretation as a function from types to types has the natural inductive definition, with S_d applied to the dimension δ in a type $\mathsf{real}\,\delta$, and S_t applied to type variables in the type. The domain of such a substitution is simply $\mathrm{dom}(S_d) \cup \mathrm{dom}(S_t)$; we assume that the sets *TyVars* and *DimVars* are disjoint.

Substitutions may also be applied to type schemes. Usually we will ensure that the substitution does not involve any of the bound variables of the scheme; that is, they are not present in its domain and are not present in any of the dimensions and types of its range. However, sometimes we will be slightly sloppy and let the substitution rename bound variables to avoid variable capture. In this case, the substitution is no longer a function but we can safely think of it as one because of the decision to identify type schemes up to alpha-conversion.

Two substitutions are equivalent under $=_D$ if they substitute equivalent dimension expressions on each variable:

$$S_1 =_D S_2 \text{ iff } S_1(v) =_D S_2(v) \text{ for all } v \in \mathit{DimVars} \cup \mathit{TyVars}.$$

An *invertible substitution*, usually denoted by U, has an inverse U^{-1} such that $U \circ U^{-1} =_D U^{-1} \circ U =_D I$.

Finally, we write $S|_{\mathcal{V}}$ to denote the *restriction* of a substitution S to the variables in a set $\mathcal{V}$:

$$S|_{\mathcal{V}}(v) = \begin{cases} S(v) & \text{if } v \in \mathcal{V}, \\ v & \text{otherwise.} \end{cases}$$

Free variables

The set of free dimension variables in a dimension expression δ has an obvious inductive definition which extends naturally to give the type and dimension variables in a type expression τ. However, this notion is of limited value because equivalent dimensions may have different free variables. For example, the dimension expression $d_1 \cdot (d_2 \cdot d_1^{-1})$ is equivalent to d_2 but the former has free variables d_1 and d_2 and the latter just d_2. An equational theory in which any pair of equivalent terms have the same free variables is called *regular*; our theory is not regular due to the axiom

$$\delta \cdot \delta^{-1} \quad =_D \quad \mathbf{1} \quad (\textit{inverses})$$

in which the variables on either side may be different.

When we present the typing rules for a language with dimension types we will identify types up to equivalence. The use of 'free variables' in the traditional syntactic sense would then be unsound, so instead we define a more semantic notion of variables in a dimension or type which *is* preserved under the equivalence relation. We are only really interested in the 'essential' dimension variables in a dimension expression δ: those variables on which a substitution can change the dimension under equivalence. Formally, a variable d in a dimension expression δ is essential if $\{d \mapsto \delta'\}\delta \neq_D \delta$ for some δ'. The normal form for a dimension expression provides an alternative, equivalent definition: dimension variables are essential if and only if they appear in the normal form. Formally,

$$\mathrm{fdv}(\delta) \stackrel{\mathrm{def}}{=} \{\, d \in \mathit{DimVars} \mid \exp_\delta(d) \neq 0 \,\}.$$

This is the method which a real implementation would employ, either maintaining dimensions in normal form or calculating $\exp_\delta$ whenever the free dimension variables are required.

A third view is that the essential variables in δ are those which appear in *all* dimensions equivalent to δ:

$$\mathrm{fdv}(\delta) = \bigcap \left\{\, \mathcal{V} \mid \delta =_D \delta',\ \mathcal{V} \text{ are the variables free in the expression } \delta' \,\right\}.$$

The following lemma proves that the three definitions are equivalent.

Lemma 2.1. *For any dimension expression δ and dimension variable d,*

$$\begin{aligned} & \{d \mapsto \delta'\}\delta \neq_D \delta \textit{ for some } \delta' && (1) \\ \Leftrightarrow\ & \exp_\delta(d) \neq 0 && (2) \\ \Leftrightarrow\ & d \textit{ is present in } \delta' \textit{ for any } \delta' \textit{ such that } \delta =_D \delta'. && (3) \end{aligned}$$

Proof. First observe that $\exp_\delta(d) \neq 0$ implies that d is present in the expression δ, and that $\exp_\delta(d) = 0$ implies that there is some $\delta' =_D \delta$ not containing the variable d (these are both easy to prove by induction on δ). Putting this together with the fact that $\exp_\delta = \exp_{\delta'}$ if and only if $\delta =_D \delta'$ gives the second equivalence: (2) $\Leftrightarrow$ (3).

For the equivalence of (1) and (2) we first prove by induction on δ that

$$\exp_{\{d \mapsto \delta'\}\delta}(d') = \begin{cases} \exp_\delta(d) \cdot \exp_{\delta'}(d') & \text{if } d = d', \\ \exp_\delta(d) \cdot \exp_{\delta'}(d') + \exp_\delta(d') & \text{otherwise.} \end{cases}$$

To show (1) $\Rightarrow$ (2) consider some dimension expression δ' such that $\{d \mapsto \delta'\}\delta \neq_D \delta$. Then $\exp_{\{d \mapsto \delta'\}\delta}(d') \neq \exp_\delta(d')$ for some d'. By considering the two cases $d = d'$ and $d \neq d'$ it follows from the result above that $\exp_\delta(d) \neq 0$ as required. For the converse (2) $\Rightarrow$ (1) we just pick $\delta' = \mathbf{1}$ and use the same result to show that $\exp_{\{d \mapsto \mathbf{1}\}\delta}(d) = 0$. Then $\exp_{\{d \mapsto \mathbf{1}\}\delta} \neq \exp_\delta$ and hence $\{d \mapsto \mathbf{1}\}\delta \neq_D \delta$. $\square$

With the final definition it is clear that $\delta_1 =_D \delta_2$ implies that $\text{fdv}(\delta_1) = \text{fdv}(\delta_2)$. The definition can be extended to simple types in the obvious way:

$$\begin{aligned} \text{fdv}(t) &= \emptyset \\ \text{fdv}(\mathsf{bool}) &= \emptyset \\ \text{fdv}(\mathsf{real}\ \delta) &= \text{fdv}(\delta) \\ \text{fdv}(\tau_1 \to \tau_2) &= \text{fdv}(\tau_1) \cup \text{fdv}(\tau_2) \end{aligned}$$

The set of free *type* variables in a simple type τ is denoted $\text{ftv}(\tau)$ and is defined conventionally:

$$\begin{aligned} \text{ftv}(t) &= \{t\} \\ \text{ftv}(\mathsf{bool}) &= \emptyset \\ \text{ftv}(\mathsf{real}\ \delta) &= \emptyset \\ \text{ftv}(\tau_1 \to \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \end{aligned}$$

We shall write $\text{fv}(\tau)$ to denote the set of type *and* dimension variables in a simple type:

$$\text{fv}(\tau) = \text{ftv}(\tau) \cup \text{fdv}(\tau)$$

These definitions seem innocuous enough, but the presence of the equivalence $=_D$ has some subtle consequences. In the case of a purely syntactic definition of free variables such as ftv, if a substitution S is applied to a type expression τ then the set of free variables in the resulting type has the property that

$$\text{ftv}(S(\tau)) = \bigcup_{t \in \text{ftv}(\tau)} \text{ftv}(S(t)).$$

But this does not hold for our definition of free dimension variables; we only have containment:

$$\mathrm{fdv}(S(\tau)) \subseteq \bigcup_{d \in \mathrm{fdv}(\tau)} \mathrm{fdv}(S(d)).$$

For a counter-example, consider the dimension type $\mathsf{real}\, d_1 \cdot d_2$ and the substitution $\{d_2 \mapsto d_1^{-1}\}$. In general, a substitution can cause variables to 'vanish' that are not in its domain; this happens only because the equational theory is not regular, and is the cause of most of the subtlety present in the dimension type system.

Expressions

The syntax of *expressions* in the language is given by the following grammar:

$$\begin{array}{lcll} e & ::= & x & \textit{variable} \\ & \mid & r & \textit{constant} \\ & \mid & e_1\, e_2 & \textit{application} \\ & \mid & \lambda x.\, e & \textit{abstraction} \\ & \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \textit{local definition} \\ & \mid & \mathsf{letrec}\ y(x) = e_1\ \mathsf{in}\ e_2 & \textit{recursive definition} \\ & \mid & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \textit{conditional} \end{array}$$

Here x and y range over an infinite set of value identifiers *Vars*, and r ranges over rational-valued constants such as `3.14`. A complete language is more likely to allow floating-point values. The notation real was chosen merely to match up with actual programming languages such as ML; no language actually caters for real numbers in the sense understood by a mathematician. In any case, we will refer loosely to values of type $\mathsf{real}\,\delta$ as 'reals'.

In order to write practical programs in the language, we assume the existence of some built-in arithmetic functions including at least the following:

$$\begin{array}{rcl} \mathtt{+}, \mathtt{-} & : & \forall d.\mathsf{real}\ d \to \mathsf{real}\ d \to \mathsf{real}\ d \\ \mathtt{*} & : & \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2 \\ \mathtt{/} & : & \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2^{-1} \\ \mathtt{<} & : & \forall d.\mathsf{real}\ d \to \mathsf{real}\ d \to \mathsf{bool} \end{array}$$

An alternative view

As mentioned at the start of this chapter, dimensions can be viewed as elements of a vector space over the integers, or $\mathbb{Z}$-module. It turns out that dimension types, substitutions and type schemes also fit very nicely into this algebraic setting. Moreover, when all dimension variables are drawn from a *finite* set then dimensions, dimension types and substitutions can be interpreted more concretely by vectors and matrices of integers. This viewpoint is described in Appendix B. None of the results of this chapter or the next use it directly, but it inspired some of the constructions used in the proofs.

2.2 Typing rules

In this section we define the type system of ML_δ through a set of typing rules. To best explain the system we give two distinct but equivalent formulations. The first of these is the more intuitive, with special typing rules for instantiation and generalisation of type schemes. To motivate and prove the correctness of an inference algorithm, we reformulate the system as a set of *syntax-directed* rules where the structure of an expression uniquely determines the structure of its typing derivations. It turns out to be surprisingly difficult to obtain a sound syntax-directed system which leads naturally to an inference algorithm. This is one reason for presenting the non-syntax-directed variant of the rules at all. Another is that we want to make clear the similarity between these rules and those for an explicitly-typed language which is presented in Chapter 5.

Non-syntax-directed rules

A *type assignment* Γ is a finite map between (value) variables and type schemes. Notions of equivalence and substitution carry over from type schemes to type assignments in the obvious way. Then a *typing judgment*

$$\Gamma \vdash e : \sigma$$

means

"In the context of type assignment Γ, the expression e has type σ."

The rules are shown in Figure 2.1. The notation $\Gamma[x : \tau]$ indicates the extension or update of the type assignment Γ with a type τ for variable x. We write $\Gamma \vdash_{\mathrm{nsd}} e : \sigma$ to mean that there is a typing derivation with conclusion $\Gamma \vdash e : \sigma$ in this *non-syntax-directed* system. The rules are almost the same as for a fragment of conventional ML, for example those presented by Cardelli [9]. Apart from the rules for constants, only two new rules are required—generalisation and specialisation for dimensions. To incorporate the equational theory of dimensions and types we could add a rule such as the following:

$$(\text{deq}) \quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \quad \tau_1 =_D \tau_2$$

This is rather like the *subsumption* rule found in type systems with subtyping, except that the relation is an equivalence instead of just an ordering. Instead of using an explicit rule such as this, we *identify* dimensions, types and type assignments up to equivalence, which greatly simplifies matters. This identification is sound because all the rules preserve the equivalence; in particular notice that our notion of *essential* free variables is necessary for rule (dgen) to be well-defined.

There are two axioms for constant reals: one for zero, which is polymorphic in its dimension, and one for all others, which are dimensionless. Observe that none of the rules explicitly introduce types involving base dimensions. We assume that in addition to the arithmetic operators already mentioned, there is a set of values

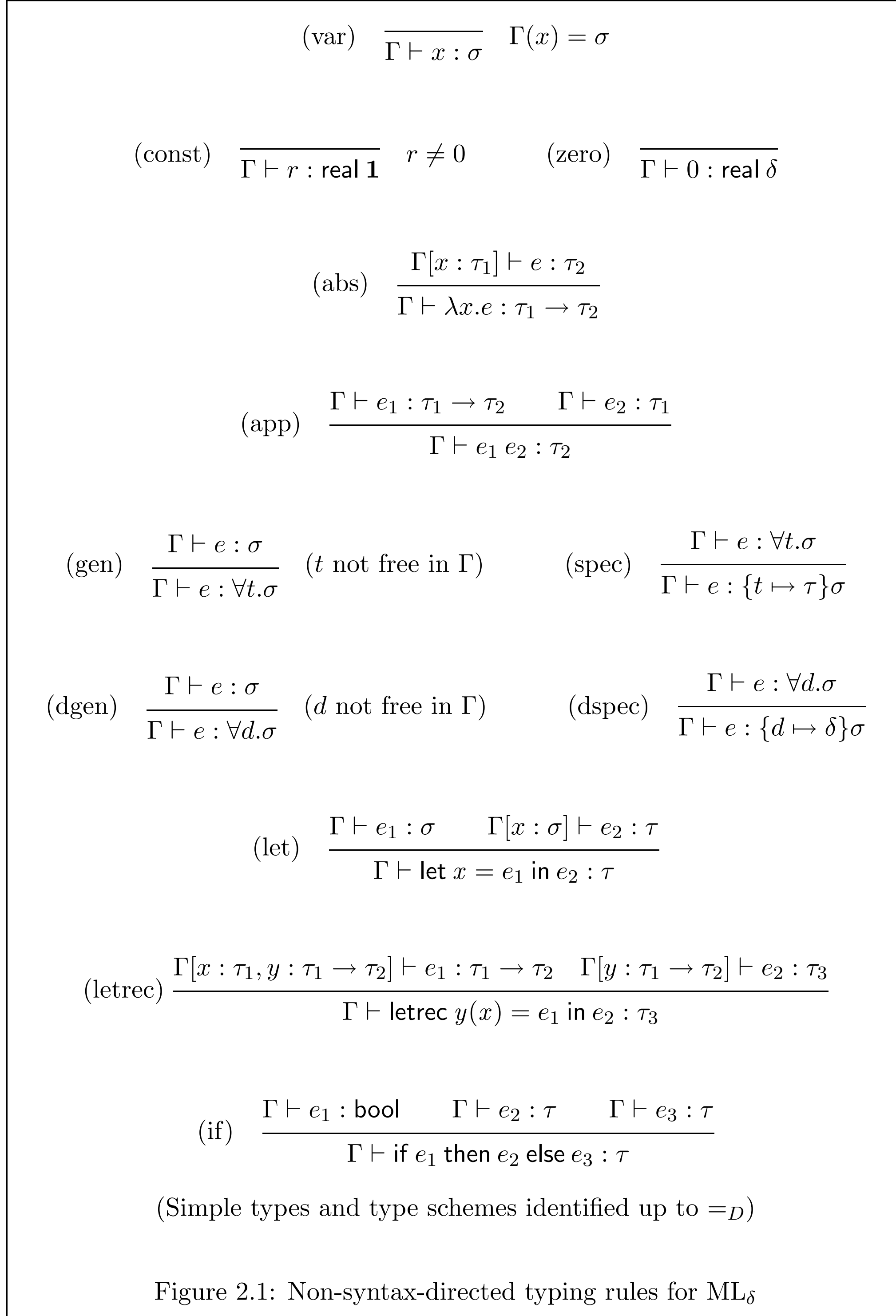


Figure 2.1: Non-syntax-directed typing rules for ML_δ

representing the default unit of measure for each base dimension. For example, in a system with mass, length and time we would expect three unit values:

$$\begin{aligned} kg &: \mathsf{real}\,\mathrm{M} \\ m &: \mathsf{real}\,\mathrm{L} \\ s &: \mathsf{real}\,\mathrm{T} \end{aligned}$$

In a full language there would be a construct for declaring base dimensions along with a default unit, such as the one described in Chapter 4. Derived units can be obtained as multiples of the default units, for example in the fragment

$$\mathsf{let}\ lb = 0.454{*}kg\ \mathsf{in}\ \ldots$$

Example. A derivation which illustrates most of the rules is shown in Figure A.1 in Appendix A.

Syntax-directed rules

The typing rules shown in Figure 2.1 are not syntax-directed because the rules (gen), (dgen), (spec) and (dspec) can be applied to any expression. However, observe that (spec) and (dspec) may only usefully be applied at the leaves of the derivation tree just after an application of the axiom (var). Also, (gen) and (dgen) may only usefully be applied to entail the first premise of an application of (let). Hence the syntax-directed formulation shown in Figure 2.2 combines (spec) and (dspec) with the (var) axiom to give a new rule (var′), and combines (gen) and (dgen) with the (let) rule to give (let′). This is done using notions of specialisation ($\preccurlyeq_D$) and generalisation ($\mathrm{Gen}(\Gamma, \tau)$) which are more general than the single-step rules used up to now. These are discussed in the next two sections.

In the new rules notice that all typing judgments have the form $\Gamma \vdash e : \tau$ for a *simple type* τ. We shall write $\Gamma \vdash_{\mathrm{sd}} e : \tau$ to mean that there is a typing derivation with conclusion $\Gamma \vdash e : \tau$ in this system. Figure A.2 in Appendix A shows a derivation of the same example used earlier.

2.3 Specialisation of type schemes

The (var′) rule rolls together an arbitrary number of applications of the (spec) and (dspec) rule together with the (var) rule which picks out a type scheme from a type assignment. To do this it uses a relation $\preccurlyeq_D$ between type schemes and simple types which we call *specialisation.* We say that a type scheme $\sigma = \forall \vec{v}.\tau$ *specialises to* a simple type τ', and write $\sigma \preccurlyeq_D \tau'$, if there is a substitution R such that

$$\begin{aligned} &\mathrm{dom}(R) \subseteq \vec{v} \\ \text{and } &R(\tau) =_D \tau'. \end{aligned}$$

$$\text{(var}'\text{)} \quad \frac{}{\Gamma \vdash x : \tau} \quad \Gamma(x) \preccurlyeq_D \tau$$

$$\text{(const)} \quad \frac{}{\Gamma \vdash r : \mathsf{real}\ \mathbf{1}} \quad r \neq 0 \qquad \text{(zero)} \quad \frac{}{\Gamma \vdash 0 : \mathsf{real}\ \delta}$$

$$\text{(abs)} \quad \frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{(app)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\text{(let}'\text{)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \mathrm{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\text{(letrec)} \quad \frac{\Gamma[x : \tau_1, y : \tau_1 \to \tau_2] \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma[y : \tau_1 \to \tau_2] \vdash e_2 : \tau_3}{\Gamma \vdash \mathsf{letrec}\ y(x) = e_1\ \mathsf{in}\ e_2 : \tau_3}$$

$$\text{(if)} \quad \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

(Simple types identified up to $=_D$, type schemes up to $\cong_D$)

Figure 2.2: Syntax-directed typing rules for ML_δ

This is not standard notation, and unfortunately many authors use the opposite inequality. In particular, the Definition of Standard ML [43] uses the symbol $\succ$. However, there seems to be agreement amongst authors of papers on subtyping that $\tau_1 \leqslant \tau_2$ means "τ_1 is a subtype of τ_2", and its interpretation as "a value of type τ_1 may be used in place of a value of type τ_2" is analogous to our interpretation of specialisation. Indeed, recent work on combining Hindley-Milner type inference with System F types [49] uses the same notation as here.

Our decision to identify type schemes up to renaming of bound variables is usefully employed in the following lemma, which states that the specialisation relation is preserved under arbitrary substitutions.

Lemma 2.2 (Substitutions preserve specialisation). *For any substitution S, if $\sigma \preccurlyeq_D \tau$ then $S(\sigma) \preccurlyeq_D S(\tau)$.*

Proof. Let $\sigma = \forall \vec{v}.\tau$, and choose $\vec{v}$ so that they are not involved in S. Then $\sigma \preccurlyeq_D \tau'$ if and only if there is a substitution R with domain $\vec{v}$ such that $R(\tau) =_D \tau'$. Now define another substitution R' by the following:

$$R'(v) = \begin{cases} S(R(v)) & \text{if } v \in \vec{v}, \\ v & \text{otherwise.} \end{cases}$$

So R' is the restriction of $S \circ R$ to $\vec{v}$. We now show that $R' \circ S =_D S \circ R$. Consider some type or dimension variable v.

- If $v \in \vec{v}$ then $R'(S(v)) = R'(v) = S(R(v))$ as required, because the domain of S does contain any of $\vec{v}$.
- If $v \notin \vec{v}$ then $S(v)$ is a type or dimension which does not involve any of $\vec{v}$. Hence $R'(S(v)) = S(v) = S(R(v))$ as required because $v \notin \text{dom}(R)$.

Thus we have that $R'(S(\tau)) =_D S(R(\tau)) =_D S(\tau')$, so $S(\sigma) \preccurlyeq_D S(\tau')$ as required. □

Observe that this result comes from general properties of substitutions and does not rely on the equivalence relation $=_D$.

Type scheme ordering

The specialisation relation can be extended to a partial order on type schemes by the following definition. A type scheme σ specialises to a type scheme σ', written $\sigma \preccurlyeq_D \sigma'$, if for all τ such that $\sigma' \preccurlyeq_D \tau$ it is the case that $\sigma \preccurlyeq_D \tau$. The following lemma provides a syntactic characterisation of this ordering analogous to the definition of the specialisation relation.

Lemma 2.3 (Type scheme ordering). *Let $\vec{v}_1$ and $\vec{v}_2$ be disjoint lists of type and dimension variables. Then $\forall \vec{v}_1.\tau_1 \preccurlyeq_D \forall \vec{v}_2.\tau_2$ if and only if there is a substitution R such that*

$$\text{dom}(R) \subseteq \vec{v}_1$$
$$\textit{and } R(\tau_1) =_D \tau_2.$$

Proof.

($\Rightarrow$). Clearly $\forall \vec{v}_2.\tau_2 \preccurlyeq_D \tau_2$ by the identity substitution. Then $\forall \vec{v}_1.\tau_1 \preccurlyeq_D \tau_2$ from the definition of the ordering on type schemes given above, and from the definition of the specialisation relation there must be some substitution R with the properties required.

($\Leftarrow$). Consider some type τ such that $\forall \vec{v}_2.\tau_2 \preccurlyeq_D \tau$. Then there is some substitution R' such that $R'(\tau_2) =_D \tau$ and $\mathrm{dom}(R') \subseteq \vec{v}_2$. Now consider $R'' = (R' \circ R)|_{\vec{v}_1}$. By the definition of specialisation this fulfils the requirements for $\forall \vec{v}_1.\tau_1 \preccurlyeq_D \tau$. □

Using this result, Lemma 2.2 can be extended to show that the ordering on type schemes is preserved under substitution also.

Type scheme equivalence

We make one final definition: if $\sigma_1 \preccurlyeq_D \sigma_2$ and $\sigma_2 \preccurlyeq_D \sigma_1$ then they specialise to exactly the same types, and we write $\sigma_1 \cong_D \sigma_2$. For ordinary ML types, this is *almost* plain renaming, except that a scheme may quantify over superfluous type variables which do not occur in its body. For dimension types, the equivalences can be more subtle, as the following three cases demonstrate. For each example we present substitutions which prove the equivalence valid by the lemma just presented together with an appropriate renaming of bound variables to ensure that they are disjoint. As one would expect, it is always possible to find substitutions which are *inverses* of each other, a fact which is proved formally by Lemma 3.10 on page 59.

Example (a).

$$\forall d_1.\forall d_2.\mathsf{real}\ d_1 \cdot d_2 \to \mathsf{real}\ d_1 \cdot d_2 \ \cong_D\ \forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1.$$

This equivalence is justified using Lemma 2.3 by the substitutions

$$\{\, d_1 \mapsto d_1 \cdot d_2^{-1} \,\}$$
$$\text{and } \{\, d_1 \mapsto d_1 \cdot d_2 \,\}.$$

Notice how one type scheme has two bound variables but the other has only one. In contrast to ordinary type polymorphism, the number of bound variables is not an immediate indication of the 'degree of polymorphism' in a type scheme. We will return to this point in Section 3.3 in the next chapter.

Example (b).

$$\begin{aligned} &\forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2 \\ &\quad \cong_D\ \forall d_1.\forall d_2.\mathsf{real}\ d_1 \cdot d_2^{-1} \to \mathsf{real}\ d_1^{-2} \cdot d_2 \to \mathsf{real}\ d_1^{-1}. \end{aligned}$$

This equivalence is validated by the substitutions

$$\{\, d_1 \mapsto d_1 \cdot d_2^{-1},\ d_2 \mapsto d_2 \cdot d_1^{-2} \,\}$$
$$\text{and } \{\, d_1 \mapsto d_1^{-1} \cdot d_2^{-1},\ d_2 \mapsto d_2^{-1} \cdot d_1^{-2} \,\}.$$

Unlike the first example, both type schemes have the same number of bound variables, and there is no other equivalent scheme with fewer.

Example (c).

$$\forall d_1.\mathsf{real}\ d_1 \cdot d_2^{-1} \cdot d_3 \to \mathsf{real}\ d_1^{-1} \cdot d_2 \ \cong_D\ \forall d_1.\mathsf{real}\ d_1^{-1} \cdot d_3 \to \mathsf{real}\ d_1 .$$

The following substitutions support this equivalence:

$$\{\ d_1 \mapsto d_1^{-1} \cdot d_2\ \}$$
$$\text{and } \{\ d_1 \mapsto d_1^{-1} \cdot d_2\ \}.$$

This example is particularly unusual, as the 'free' dimension variables in the equivalent type schemes do not even coincide, if the free variables in a scheme $\forall \vec{v}.\tau$ are defined as those variables which are present in $\mathrm{fv}(\tau)$ but not in $\vec{v}$. Although by this definition the free variables are preserved under $=_D$, they are *not* preserved by equivalence up to specialisation.

Free variables in a type scheme

We require some notion of 'essential' free variables again. For simple types we gave three alternative views: those variables on which a substitution could change the type, those variables present in the normal form of the type, and those variables present in all equivalent types. We can do the same for type schemes as follows.

The first says that a variable v is essential in a scheme σ if $S(\sigma) \not\cong_D \sigma$ for some substitution S whose domain is $\{v\}$. If this is not the case, and no substitution for the variable can change the type scheme, then the non-essential variable is 'dependent' on one of the bound variables in the scheme. In Example (c) above, the free variable d_2 is linked to the bound variable d_1 because whenever d_1 appears, d_2^{-1} appears too.

In the next chapter a simplification procedure is presented which calculates for any type scheme σ a unique representative for its equivalence class under $\cong_D$. One property of this canonical form for type schemes is that it contains only the essential free variables, so by making sure that type schemes remain in this form we can use the naive method to find its free variables. For one thing it is easy to see that the application of a capture-avoiding substitution to a type scheme cannot *introduce* non-essential free variables.

Finally, the essential free variables can be viewed as those which appear in every type to which the type scheme specialises:

$$\mathrm{fv}(\sigma) = \bigcap \{\ \mathrm{fv}(\tau)\ \mid\ \sigma \preccurlyeq_D \tau\ \}.$$

Notice how this formulation neatly avoids mentioning the bound variables in the type scheme explicitly. The proof that the three definitions are equivalent is deferred until the next chapter.

Using this final definition straightforward set-theoretic reasoning leads immediately to

$$\sigma_1 \preccurlyeq_D \sigma_2 \ \Rightarrow\ \mathrm{fv}(\sigma_1) \subseteq \mathrm{fv}(\sigma_2)$$

and its corollary

$$\sigma_1 \cong_D \sigma_2 \Rightarrow \mathrm{fv}(\sigma_1) = \mathrm{fv}(\sigma_2)$$

as we required.

Extension to type assignments

The definitions of type scheme ordering and type scheme equivalence extend to type assignments in a pointwise way. Formally, $\Gamma_1 \preccurlyeq_D \Gamma_2$ if Γ_1 and Γ_2 share domains and for all $x \in \mathrm{dom}(\Gamma_1)$ it is the case that $\Gamma_1(x) \preccurlyeq_D \Gamma_2(x)$. The definition for equivalence is similar.

Likewise we define the free variables in a type assignment Γ, denoted $\mathrm{fv}(\Gamma)$, as the union of all free variables in the type schemes present in the assignment:

$$\mathrm{fv}(\Gamma) = \bigcup \{ \mathrm{fv}(\Gamma(x)) \mid x \in \mathrm{dom}(\Gamma) \}.$$

2.4 Generalisation with respect to a type assignment

The (let′) rule combines several uses of (gen) and (dgen) together with the (let) rule which introduces a type scheme into the type assignment. Given a type τ we would like to obtain a type scheme which is as general as possible. For conventional ML, this means applying the rule (gen) for every type variable in τ which does not violate the side-condition. This process is sometimes called the *closure* of a type with respect to a type assignment and is defined by

$$\mathrm{Gen}(\Gamma, \tau) \stackrel{\mathrm{def}}{=} \forall \vec{t}.\tau, \quad \text{where } \vec{t} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma).$$

In the sections which follow we show how things are not as simple when dimensions are added to the type system.

The wrong way to generalise

The natural extension of the above definition to a dimension type system would be the following:

$$\mathrm{NGen}(\Gamma, \tau) \stackrel{\mathrm{def}}{=} \forall \vec{v}.\tau, \quad \text{where } \vec{v} = \mathrm{fv}(\tau) \setminus \mathrm{fv}(\Gamma).$$

Unfortunately this obvious approach has some subtle problems (NGen stands for “naive generalisation”), noted also by Rittri in his article on polymorphic recursion for dimension types [58]. Consider the assignment $\Gamma = \{x : \mathsf{real}\, d_1 \cdot d_2\}$ and the type $\tau = \mathsf{real}\, d_1 \to \mathsf{real}\, d_2$. Then by the rule given above,

$$\mathrm{NGen}(\Gamma, \tau) = \mathsf{real}\, d_1 \to \mathsf{real}\, d_2$$

which introduces no polymorphism. Intuitively, though, there is something strange about Γ: although it contains two free dimension variables, it has only

one *degree of freedom*, as any substitution made for d_1 can be undone by a substitution made for d_2. In some sense, the variables d_1 and d_2 are not independent. The type τ, on the other hand, has two degrees of freedom because substitutions made for the free variables d_1 and d_2 *are* independent. We shall see shortly that the non-independence of d_1 and d_2 in the type assignment is 'hiding' some polymorphism which can be extracted by a more sophisticated procedure than NGen.

Free variable reduced form

We make the ideas above more precise by the following definition. We say that a type scheme σ is in *free variable reduced form* if for any two substitutions S_1 and S_2,

$$S_1(\sigma) \cong_D S_2(\sigma) \iff S_1(v) =_D S_2(v) \text{ for all } v \in \text{fv}(\sigma).$$

A similar definition can be made for type assignments. Its essence is that a substitution on the free variables of a type scheme in free variable reduced form is uniquely determined by its effect when applied to the type scheme. This is automatically the case for free type variables, but in general there may be several distinct dimension substitutions which have the same effect on a type scheme.

Example. The type $\mathsf{real}\, d_1 \cdot d_2$ is not in free variable reduced form because there are substitutions which have the same effect on $\mathsf{real}\, d_1 \cdot d_2$ but different effects on d_1 or d_2. Consider, for example, the identity substitution and the substitution $\{ d_1 \mapsto d_2^2, d_2 \mapsto d_1 \cdot d_2^{-1} \}$.

The importance of this notion is that the naive generalisation procedure $\text{NGen}(\Gamma, \tau)$ is sound if the type assignment Γ is in free variable reduced form. It is possible to transform an arbitrary type scheme or type assignment into free variable reduced form by means of an invertible substitution. Algebraically, this substitution is a *change of basis.*

Proposition 2.4 (Change of basis). *For any type assignment Γ there is an invertible substitution U so that $U(\Gamma)$ is in free variable reduced form.*

Proof. The proof is by construction and is postponed until the next chapter when a simplification procedure is presented which calculates U. □

The right way to generalise

The discussion above suggests the following method of calculating Gen:

1. Let U be an invertible substitution so that $U(\Gamma)$ is in free variable reduced form.
2. Calculate $\text{NGen}(U(\Gamma), U(\tau)) = \forall \vec{v}. U(\tau)$ where $\vec{v} = \text{fv}(U(\tau)) \setminus \text{fv}(U(\Gamma))$.
3. Then $\text{Gen}(\Gamma, \tau)$ is given by $U^{-1}(\forall \vec{v}. U(\tau))$. This may involve renaming the variables in $\vec{v}$ to avoid clashes with the substitution U^{-1}.

Reusing the earlier example in which $\Gamma = \{x : \mathsf{real}\, d_1 \cdot d_2\}$ and $\tau = \mathsf{real}\, d_1 \rightarrow \mathsf{real}\, d_2$, a suitable substitution U is

$$\begin{aligned} U &= \{d_1 \mapsto d_1 \cdot d_2^{-1}\} \\ \text{so } U^{-1} &= \{d_1 \mapsto d_1 \cdot d_2\}\,. \end{aligned}$$

Then

$$\begin{aligned} \mathrm{Gen}(\Gamma, \tau) &= U^{-1}(\mathrm{NGen}(U(\{x : \mathsf{real}\, d_1 \cdot d_2\}), U(\mathsf{real}\, d_1 \rightarrow \mathsf{real}\, d_2))) \\ &= U^{-1}(\mathrm{NGen}(\{x : \mathsf{real}\, d_1\}\,, \mathsf{real}\, d_1 \cdot d_2^{-1} \rightarrow \mathsf{real}\, d_2)) \\ &= U^{-1}(\forall d_2.\mathsf{real}\, d_1 \cdot d_2^{-1} \rightarrow \mathsf{real}\, d_2) \\ &= U^{-1}(\forall d_3.\mathsf{real}\, d_1 \cdot d_3^{-1} \rightarrow \mathsf{real}\, d_3) \\ &= \forall d_3.\mathsf{real}\, d_1 \cdot d_2 \cdot d_3^{-1} \rightarrow \mathsf{real}\, d_3. \end{aligned}$$

This is correct: it confirms our intuition that although Γ and τ have the same free variables, the type assignment Γ has one degree of freedom less than the type τ and so generalisation should be able to 'uncover' this difference to produce one bound variable in the type scheme deduced.

In summary, here is the correct definition of Gen.

Definition 2.5.

$$\mathrm{Gen}(\Gamma, \tau) \stackrel{\text{def}}{=} U^{-1}(\mathrm{NGen}(U(\Gamma), U(\tau)))$$

where U is invertible and $U(\Gamma)$ is in free variable reduced form.

An important property of Gen

In this section we prove an important property of generalisation from which all other results we require can be derived without recourse to intricate reasoning about the syntax of type schemes and types.

First we show that it holds in the restricted case when Γ is already in free variable reduced form: then the naive procedure $\mathrm{NGen}(\Gamma, \tau)$ is sufficient.

Lemma 2.6. *Let Γ be a type assignment in free variable reduced form, let τ be any type, and let $\sigma = \mathrm{NGen}(\Gamma, \tau)$. Then for any substitution S,*

$$S(\sigma) \preccurlyeq_D \tau' \textit{ iff there is some } R \textit{ such that } R(\Gamma) \cong_D S(\Gamma) \textit{ and } R(\tau) =_D \tau'.$$

Proof. By renaming we can assume that S does not involve the bound variables in σ, so $S(\sigma) = \forall \vec{v}.S(\tau)$ where $\vec{v} = \mathrm{fv}(\tau) \setminus \mathrm{fv}(\Gamma)$. It is easy to see that $\mathrm{fv}(S(\Gamma)) \cap \vec{v} = \emptyset$. Consider some $v \in \mathrm{fv}(\Gamma)$, so $v \notin \vec{v}$. The assumption on S then ensures that $S(v)$ contains none of the variables in $\vec{v}$.

The two directions of the equivalence are proved as follows.

($\Rightarrow$). From the definition of the specialisation relation, $\forall \vec{v}.S(\tau) \preccurlyeq_D \tau'$ if and only if there is some substitution R' such that $R'(S(\tau)) =_D \tau'$ with $\mathrm{dom}(R') \subseteq \vec{v}$. Then $R = R' \circ S$ has the properties required to obtain the conclusion: $R'(S(\Gamma)) \cong_D S(\Gamma)$ because $\mathrm{dom}(R') \cap \mathrm{fv}(S(\Gamma)) = \emptyset$.

($\Leftarrow$). Let $R' = R|_{\vec{v}}$. To show that $R'(S(\tau)) =_D \tau'$ we show that for all $v \in \mathrm{fv}(\tau)$ it is the case that $R'(S(v)) =_D R(v)$. Consider two possibilities:

- If $v \in \text{fv}(\Gamma)$ then because $S(v)$ does not involve $\vec{v}$ it must be true that $R'(S(v)) =_D S(v)$. Now because Γ is in free variable reduced form, the fact that $R(\Gamma) \cong_D S(\Gamma)$ implies that for all $v \in \text{fv}(\Gamma)$ we have $R(v) =_D S(v)$. Hence $R'(S(v)) =_D R(v)$ as required.
- If $v \notin \text{fv}(\Gamma)$ then $v \in \vec{v}$ so $S(v) =_D v$. Hence $R'(S(v)) =_D R'(v) =_D R(v)$ by definition of R'.

□

The result for the general case then follows straightforwardly from Definition 2.5.

Proposition 2.7 (Gen)**.** *Let Γ be a type assignment, let τ be any type, and let $\sigma = \text{Gen}(\Gamma, \tau)$. Then for any substitution S,*

$$S(\sigma) \preccurlyeq_D \tau' \textit{ iff there is some } R \textit{ such that } R(\Gamma) \cong_D S(\Gamma) \textit{ and } R(\tau) =_D \tau'.$$

Proof. From the definition, $\text{Gen}(\Gamma, \tau) = U^{-1}(\sigma')$ where $\sigma' = \text{NGen}(U(\Gamma), U(\tau))$ and U is an invertible substitution such that $U(\Gamma)$ is in free variable reduced form. By the previous lemma, $S(U^{-1}(\sigma')) \preccurlyeq_D \tau'$ if and only if there is some substitution R' such that

$$R'(U(\Gamma)) \cong_D (S \circ U^{-1})(U(\Gamma)) \text{ and } R'(U(\tau)) =_D \tau'.$$

Since $(S \circ U^{-1})(U(\Gamma)) \cong_D S(\Gamma)$ a suitable substitution is just $R = R' \circ U$. □

Rittri's solution to the problem of generalisation is slightly different to ours: he finds a substitution R which is applied to the type τ before performing naive generalisation [58]. This substitution leaves the type assignment alone, so $R(\Gamma) \cong_D \Gamma$, but 'reveals' the additional polymorphism present in τ by a renaming of variables. The following lemma shows how the two methods are related by determining such a substitution using Definition 2.5.

Lemma 2.8. *For any type assignment Γ and type τ,*

$$\text{Gen}(\Gamma, \tau) \cong_D \text{NGen}(\Gamma, R(\tau))$$

for some substitution R such that $R(\Gamma) \cong_D \Gamma$.

Proof. From definition, we have that

$$\text{Gen}(\Gamma, \tau) = U^{-1}(\forall \vec{v}. U(\tau))$$

where U is an invertible substitution such that $U(\Gamma)$ is in free variable reduced form, and $\vec{v} = \text{fv}(U(\tau)) \setminus \text{fv}(U(\Gamma))$. Now let S be an invertible renaming of the variables in $\vec{v}$ given by

$$S = \{\vec{v} \leftrightarrow \vec{v}_0\}$$

where $\vec{v}_0$ are fresh. Then we can push U^{-1} inside the type scheme, as follows:

$$U^{-1}(\forall \vec{v}. U(\tau)) \cong_D \forall \vec{v}_0. U^{-1}(S(U(\tau))).$$

The substitution required for the result is then just $R = U^{-1} \circ S \circ U$. Clearly $R(\Gamma) \cong_D \Gamma$ because $\mathrm{dom}(S) \cap \mathrm{fv}(U(\Gamma)) = \emptyset$. Now consider the free variables in $U(\tau)$:

$$\mathrm{fv}(U(\tau)) = (\mathrm{fv}(U(\Gamma)) \cap \mathrm{fv}(U(\tau))) \cup \vec{v}.$$

Then

$$\mathrm{fv}(S(U(\tau))) = (\mathrm{fv}(U(\Gamma)) \cap \mathrm{fv}(U(\tau))) \cup \vec{v}_0$$

and so

$$\mathrm{fv}(U^{-1}(S(U(\tau)))) = (\mathrm{fv}(\Gamma) \cap \mathrm{fv}(\tau)) \cup \vec{v}_0.$$

Hence $\vec{v}_0 = \mathrm{fv}(R(\tau)) \setminus \mathrm{fv}(\Gamma)$ as required by the definition of NGen. □

Some lemmas concerning Gen

We now use Proposition 2.7 to prove a series of lemmas on which the correctness of the type system depends.

First we show how substitutions interact with generalisation. In general, the type scheme resulting from generalisation is less general if a substitution is applied to the type and type assignment first. For the special case of an invertible substitution, or change of basis, an equivalence holds: generalisation commutes with change of basis, as one would expect.

Lemma 2.9. *For any substitution S, type assignment Γ and type τ,*

$$S(\mathrm{Gen}(\Gamma, \tau)) \preccurlyeq_D \mathrm{Gen}(S(\Gamma), S(\tau)).$$

Moreover, for an invertible substitution U it is the case that

$$U(\mathrm{Gen}(\Gamma, \tau)) \cong_D \mathrm{Gen}(U(\Gamma), U(\tau)).$$

Proof.

For the first part we must show that $\mathrm{Gen}(S(\Gamma), S(\tau)) \preccurlyeq_D \tau'$ implies that $S(\mathrm{Gen}(\Gamma, \tau)) \preccurlyeq_D \tau'$. If the premise holds then by Proposition 2.7 there is some substitution R such that $R(S(\Gamma)) \cong_D S(\Gamma)$ and $R(S(\tau)) =_D \tau'$. Now let $R' = R \circ S$. Then $R'(\Gamma) \cong_D S(\Gamma)$ and $R'(\tau) =_D \tau'$ which by Proposition 2.7 gives $S(\mathrm{Gen}(\Gamma, \tau)) \preccurlyeq_D \tau'$ as required.

For the second part consider some τ' such that $U(\mathrm{Gen}(\Gamma, \tau)) \preccurlyeq_D \tau'$. Then there is a substitution R such that $R(\Gamma) \cong_D U(\Gamma)$ and $R(\tau) =_D \tau'$. Now let $R' = R \circ U^{-1}$. Then $R'(U(\Gamma)) \cong_D R(\Gamma) \cong_D U(\Gamma)$ and $R'(U(\tau)) =_D R(\tau) =_D \tau'$ so by Proposition 2.7 $\mathrm{Gen}(U(\Gamma), U(\tau)) \preccurlyeq_D \tau'$ as required. □

In the general case when S is not necessarily invertible we have the following equivalence result. Before presenting it we introduce one new concept. The *restriction* of a substitution S to a type assignment Γ is a substitution S_0 such that

- $S_0(\Gamma) \cong_D S(\Gamma)$.
- For any other substitution S' such that $S'(\Gamma) \cong_D S(\Gamma)$ there is some substitution R such that $S' =_D R \circ S_0$.

In the terminology of unification theory, S_0 is a *most general matcher* of Γ to $S(\Gamma)$. In the absence of the equivalence $=_D$, this is just a complicated way of defining the restriction of a substitution S to the *variables* in a type assignment Γ, that is $S|_{\mathrm{fdv}(\Gamma)}$. In the presence of $=_D$, this is only the case if Γ is in free variable reduced form, as can be verified easily from the definition on page 30. It is also easy to see that if the restriction of $S \circ U^{-1}$ to $U(\Gamma)$ is R for an invertible substitution U, then the restriction on S to Γ is $S_0 = R \circ U$. Hence using Proposition 2.4 to obtain a suitable substitution U we can always find S_0.

Lemma 2.10. *For any substitution S, type assignment Γ and type τ, there is some other substitution S_0 such that $S(\Gamma) \cong_D S_0(\Gamma)$ and*

$$S(\mathrm{Gen}(\Gamma, \tau)) \cong_D \mathrm{Gen}(S_0(\Gamma), S_0(\tau)).$$

Proof. Let S_0 be the restriction of S to Γ. Then the two directions of the equivalence can be proved as follows.

($\preccurlyeq_D$). From Proposition 2.7, if $\mathrm{Gen}(S_0(\Gamma), S_0(\tau)) \preccurlyeq_D \tau'$ then there is a substitution R such that $R(S_0(\Gamma)) \cong_D S_0(\Gamma)$ and $R(S_0(\tau)) =_D \tau'$. Let $R' = R \circ S_0$. Then $R'(\Gamma) \cong_D S(\Gamma)$ and $R'(\tau) =_D \tau'$ so $S(\mathrm{Gen}(\Gamma, \tau)) \preccurlyeq_D \tau'$ as required.

($\succcurlyeq_D$). From Proposition 2.7, if $S(\mathrm{Gen}(\Gamma, \tau)) \preccurlyeq_D \tau'$ then there is some substitution R such that $R(\Gamma) \cong_D S(\Gamma)$ and $R(\tau) =_D \tau'$. Then from the definition of S_0 we know that $R =_D R' \circ S_0$ for some R'. Hence $R'(S_0(\Gamma)) \cong_D S_0(\Gamma)$ and $R'(S_0(\tau)) =_D \tau'$ which entails $\mathrm{Gen}(S_0(\Gamma), S_0(\tau)) \preccurlyeq_D \tau'$ as required. $\square$

If a substitution leaves a type scheme fixed, then it will do the same for a more general type scheme. The same is true for type assignments.

Lemma 2.11. *If $S(\sigma_2) \cong_D \sigma_2$ and $\sigma_1 \preccurlyeq_D \sigma_2$ then $S(\sigma_1) \cong_D \sigma_1$. Also, if $S(\Gamma_2) \cong_D \Gamma_2$ and $\Gamma_1 \preccurlyeq_D \Gamma_2$ then $S(\Gamma_1) \cong_D \Gamma_1$.*

Proof. We show first that the result holds if σ_2 is in free variable reduced form. In this case, if $S(\sigma_2) \cong_D \sigma_2$ then $S(v) =_D v$ for all $v \in \mathrm{fv}(\sigma_2)$. Also, $\sigma_1 \preccurlyeq_D \sigma_2$ implies that $\mathrm{fv}(\sigma_1) \subseteq \mathrm{fv}(\sigma_2)$ (by our definition of *essential* free variables), so clearly it must be the case that $S(\sigma_1) \cong_D \sigma_1$ also.

For the general case, let U be an invertible substitution so that $U(\sigma_2)$ is in free variable reduced form. Then we can write $S(\sigma_2) \cong_D \sigma_2$ as $S(U^{-1}(U(\sigma_2))) \cong_D \sigma_2$, and apply U to both sides to obtain

$$(U \circ S \circ U^{-1})(U(\sigma_2)) \cong_D U(\sigma_2).$$

We also have that $U(\sigma_1) \preccurlyeq_D U(\sigma_2)$, so using the result for the special case proved above we obtain

$$(U \circ S \circ U^{-1})(U(\sigma_1)) \cong_D U(\sigma_1).$$

Then by applying U^{-1} to both sides we get $S(\sigma_1) \cong_D \sigma_1$ as required.

The same result for type assignments follows by a simple induction on the size of the type assignment. □

This lemma is then used to prove that generalisation of a fixed type τ preserves the specialisation ordering.

Lemma 2.12. *If* $\Gamma_1 \preccurlyeq_D \Gamma_2$ *then* $\mathrm{Gen}(\Gamma_1, \tau) \preccurlyeq_D \mathrm{Gen}(\Gamma_2, \tau)$.

Proof. We must show for any τ' that $\mathrm{Gen}(\Gamma_2, \tau) \preccurlyeq_D \tau'$ implies $\mathrm{Gen}(\Gamma_1, \tau) \preccurlyeq_D \tau'$. From Proposition 2.7 there must be a substitution R such that $R(\Gamma_2) \cong_D \Gamma_2$ and $R(\tau) =_D \tau'$. Then from Lemma 2.11 $R(\Gamma_1) \cong_D \Gamma_1$ also. Hence $\mathrm{Gen}(\Gamma_1, \tau) \preccurlyeq_D \tau'$ as required. □

2.5 Properties of syntax-directed derivations

We have now discussed in detail the notions of specialisation and generalisation required by the syntax-directed typing rules of Figure 2.2. To simplify the remaining proofs of this chapter and the proofs in the next, we assume that type schemes and type assignments in these rules are identified up to the equivalence $\cong_D$. This identification is sound for the following reasons:

- Specialisation in (var′) preserves the equivalence by definition, that is, if $\sigma_1 \preccurlyeq_D \tau$ and $\sigma_1 \cong_D \sigma_2$ then $\sigma_2 \preccurlyeq_D \tau$.
- Generalisation in (let′) preserves equivalence by a corollary of Lemma 2.12: if $\Gamma_1 \cong_D \Gamma_2$ then $\mathrm{Gen}(\Gamma_1, \tau) \cong_D \mathrm{Gen}(\Gamma_2, \tau)$.
- Substitutions preserve equivalence, that is, if $\sigma_1 \cong_D \sigma_2$ then $S(\sigma_1) \cong_D S(\sigma_2)$ and likewise for type assignments.

We now prove two lemmas concerning syntax-directed typing derivations. The first shows that if a type for e can be obtained in the context of a type assignment Γ_2, then the same type can be obtained under a more general type assignment Γ_1.

Lemma 2.13 (Context generalisation). *If* $\Gamma_2 \vdash_{\mathrm{sd}} e : \tau$ *and* $\Gamma_1 \preccurlyeq_D \Gamma_2$ *then* $\Gamma_1 \vdash_{\mathrm{sd}} e : \tau$.

Proof. By induction on the structure of e. We present only the cases for variables and let constructs; the rest are straightforward.

- For a variable x we have the derivation

  $$\frac{}{\Gamma_2 \vdash x : \tau}\,(\mathrm{var}')$$

  under the side-condition that $\Gamma_2(x) \preccurlyeq_D \tau$. Now $\Gamma_1(x) \preccurlyeq_D \Gamma_2(x)$ so the required derivation follows directly from the definition of $\preccurlyeq_D$ as a binary relation on schemes.

- A let construct must have the following typing derivation:

$$\frac{\Gamma_2 \vdash e_1 : \tau_1 \qquad \Gamma_2[x : \mathrm{Gen}(\Gamma_2, \tau_1)] \vdash e_2 : \tau_2}{\Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\mathrm{let}')$$

By the induction hypothesis there must be a derivation of $\Gamma_1 \vdash e_1 : \tau_1$. Then from Lemma 2.12 we can infer that $\mathrm{Gen}(\Gamma_1, \tau_1) \preccurlyeq_D \mathrm{Gen}(\Gamma_2, \tau_1)$ and hence apply the induction hypothesis to the second premise to get a derivation of

$$\Gamma_1[x : \mathrm{Gen}(\Gamma_1, \tau_1)] \vdash e_2 : \tau_2.$$

The result then follows by an application of the (let′) rule.

□

If a typing derivation $\Gamma \vdash_{\mathrm{sd}} e : \tau$ contains free type or dimension variables, then it represents a kind of derivation *scheme* from which other less general derivations may be obtained by substitution for those free variables. This is proved by the following lemma.

Lemma 2.14 (Substitution on a derivation). *For any derivation $\Gamma \vdash_{\mathrm{sd}} e : \tau$ and substitution S there is a derivation $S(\Gamma) \vdash_{\mathrm{sd}} e : S(\tau)$.*

Proof. First we assume that no bound variables in Γ are in $\mathrm{dom}(S)$ or are introduced by S. This can be ensured by renaming all bound variables. Then we proceed by induction on the structure of e. Only the cases for variables and let constructs are of interest.

- A variable x must have the derivation

$$\frac{}{\Gamma \vdash x : \tau}\ (\mathrm{var}')$$

and from the side-condition we know that $\Gamma(x) \preccurlyeq_D \tau$. Then $S(\Gamma)(x) \preccurlyeq_D S(\tau)$ because by Lemma 2.2 the specialisation relation is preserved under substitutions. The result follows immediately.

- A let construct has the derivation

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \mathrm{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\mathrm{let}')$$

Applying the induction hypothesis to the second premise gives a derivation of

$$S(\Gamma)[x : S(\mathrm{Gen}(\Gamma, \tau_1))] \vdash e_2 : S(\tau_2).$$

Then from Lemma 2.10 we know that there is a substitution R such that $R(\Gamma) \cong_D S(\Gamma)$ and $\mathrm{Gen}(R(\Gamma), R(\tau_1)) \cong_D S(\mathrm{Gen}(\Gamma, \tau_1))$. Substituting these into the derivation gives

$$R(\Gamma)[x : \mathrm{Gen}(R(\Gamma), R(\tau_1))] \vdash e_2 : S(\tau_2)$$

and applying the induction hypothesis to the first premise produces a derivation of

$$R(\Gamma) \vdash e_1 : R(\tau_1).$$

These two derivations can be combined using the (let′) rule to give

$$\frac{R(\Gamma) \vdash e_1 : R(\tau_1) \qquad R(\Gamma)[x : \mathrm{Gen}(R(\Gamma), R(\tau_1))] \vdash e_2 : S(\tau_2)}{R(\Gamma) \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : S(\tau_2)}\ (\mathrm{let}').$$

This is the derivation we require because $R(\Gamma) \cong_D S(\Gamma)$.

□

2.6 Equivalence of the two formulations

We now prove that the syntax-directed and non-syntax-directed rules permit equivalent typings. There is a similar proof in Henglein's paper on polymorphic recursion [23]. The obvious statement (that $\Gamma \vdash_{\mathrm{nsd}} e : \tau$ if and only if $\Gamma \vdash_{\mathrm{sd}} e : \tau$) comes about as a special case of the following more general result.

Proposition 2.15 (Equivalence of formulations). *For any type assignment Γ and expression e it is the case that*

$$\Gamma \vdash_{\mathrm{nsd}} e : \sigma \quad \textit{if and only if for all } \tau \textit{ such that } \sigma \preccurlyeq_D \tau, \quad \Gamma \vdash_{\mathrm{sd}} e : \tau.$$

Proof. By renaming we can ensure that all bound variables in the type assignment Γ are distinct from free variables in Γ and σ. Then the proof proceeds as follows.

($\Rightarrow$). By induction on the structure of the derivation. We give the more interesting cases.

- The derivation is a single application of (var).

$$\frac{}{\Gamma \vdash x : \sigma}\ (\mathrm{var})$$

From the side-condition we know that $\Gamma(x) = \sigma$ so we can immediately construct the required derivation using (var′) for any τ such that $\sigma \preccurlyeq_D \tau$.

- The last rule was (dgen).

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall d.\sigma}\ (\mathrm{dgen}) \quad d \text{ not free in } \Gamma$$

By the induction hypothesis, for any τ' such that $\sigma \preccurlyeq_D \tau'$ there is a derivation $\Gamma \vdash_{\mathrm{sd}} e : \tau'$. Then by Lemma 2.14 (substitution on a derivation), and making use of the side-condition, for any dimension δ there is a derivation $\Gamma \vdash_{\mathrm{sd}} e : \{d \mapsto \delta\}\tau'$. Now because $\forall d.\sigma \preccurlyeq_D \tau$ if and only if $\sigma \preccurlyeq_D \tau'$ where $\tau = \{d \mapsto \delta\}\tau'$ for some δ, we have the result as required.

The case for (gen) is similar.

- The last rule was (dspec).

$$\frac{\Gamma \vdash e : \forall d.\sigma}{\Gamma \vdash e : \{d \mapsto \delta\}\sigma} \text{(dspec)}$$

  If $\{d \mapsto \delta\}\sigma \preccurlyeq_D \tau$ then clearly $\forall d.\sigma \preccurlyeq_D \tau$. Then applying the induction hypothesis to the premise gives a derivation $\Gamma \vdash_{\text{sd}} e : \tau$ as required.

  The case for (spec) is similar.

- The last rule was (let).

$$\frac{\Gamma \vdash e_1 : \forall \vec{v}.\tau_1 \qquad \Gamma[x : \forall \vec{v}.\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \text{(let)}$$

  By the induction hypothesis, for any τ such that $\forall \vec{v}.\tau_1 \preccurlyeq_D \tau$ there is a derivation

$$\Gamma \vdash_{\text{sd}} e_1 : \tau. \tag{1}$$

  Similarly, by applying the induction hypothesis we can obtain a syntax-directed derivation of the second premise. Now it is clear from Proposition 2.7 that $\text{Gen}(\Gamma, \tau_1) \preccurlyeq_D \forall \vec{v}.\tau_1$ if $\text{fv}(\Gamma) \cap \vec{v} = \emptyset$, which is ensured by our assumption about bound variables. Then by Lemma 2.13 there is a derivation of

$$\Gamma[x : \text{Gen}(\Gamma, \tau_1)] \vdash_{\text{sd}} e_2 : \tau_2 \tag{2}$$

  and putting (1) and (2) together gives the following derivation, as required.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \text{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \text{(let}'\text{)}$$

($\Leftarrow$). By induction on the structure of e. Again we consider only the interesting cases.

- The expression is a variable. Then we have the following syntax-directed derivation:

$$\frac{}{\Gamma \vdash x : \tau} \text{(var}'\text{)} \quad \Gamma(x) \preccurlyeq_D \tau$$

  Let $\Gamma(x) = \forall v_1 \ldots v_n.\tau'$. Then from the definition of specialisation there is a substitution R with $\text{dom}(R) \subseteq \{v_1, \ldots, v_n\}$ such that $R(\tau') =_D \tau$. Moreover, provided that the range of R does not involve the variables in its domain (which was ensured by renaming), then R can be written as the composition of n substitutions

$$R_n \circ \cdots \circ R_1$$

where each R_i is a single type or dimension substitution having the form $\{v_i \mapsto \tau_i\}$ or $\{v_i \mapsto \delta_i\}$. Then the equivalent derivation in the non-syntax-directed system is the following:

$$\dfrac{\dfrac{}{\Gamma \vdash x : \forall v_1 \ldots v_n.\tau'}\ (\text{var})}{\Gamma \vdash x : (R_n \circ \cdots \circ R_1)(\tau')}\ (\text{spec and dspec}).$$

- The expression is a let construct with a derivation as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x : \mathrm{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

By Lemma 2.8 there must be some substitution R such that $R(\Gamma) \cong_D \Gamma$ and $\mathrm{Gen}(\Gamma, \tau_1) \cong_D \mathrm{NGen}(\Gamma, R(\tau_1))$. From definition $\mathrm{NGen}(\Gamma, R(\tau_1)) = \forall \vec{v}.R(\tau_1)$ where $\vec{v} = \mathrm{fv}(R(\tau_1)) \setminus \mathrm{fv}(\Gamma)$. Then by Lemma 2.14 we can apply R to obtain syntax-directed derivations of

$$\Gamma \vdash e_1 : R(\tau_1)$$
$$\text{and } \Gamma[x : \forall \vec{v}.R(\tau_1)] \vdash e_2 : \tau_2.$$

Next we apply the induction hypothesis to get derivations of both of these in the non-syntax-directed system, and use rule (gen) or rule (dgen) for each variable in $\vec{v}$ to form the required derivation as follows:

$$\dfrac{\dfrac{\Gamma \vdash e_1 : R(\tau_1)}{\Gamma \vdash e_1 : \forall \vec{v}.R(\tau_1)}\ (\text{dgen, gen}) \qquad \Gamma[x : \forall \vec{v}.R(\tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\text{let})$$

□

2.7 Normalising a typing derivation

The case for (let′) in the above proof of equivalence suggests a method of transforming any syntax-directed derivation into one which uses only the naive generalisation procedure NGen. The following lemma proves that this is possible.

Lemma 2.16. *If* $\Gamma \vdash_{\mathrm{sd}} e : \tau$ *then there is another derivation in which every* (let′) *rule uses just* NGen*.*

Proof. By induction on the structure of the derivation. The only case of interest is that for (let′):

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \mathrm{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\text{let}')$$

From Lemma 2.8 there must be some substitution R such that $R(\Gamma) \cong_D \Gamma$ and $\mathrm{Gen}(\Gamma, \tau_1) \cong_D \mathrm{NGen}(\Gamma, R(\tau_1))$. From definition $\mathrm{NGen}(\Gamma, R(\tau_1)) = \forall \vec{v}.R(\tau_1)$

where $\vec{v} = \text{fv}(R(\tau_1)) \setminus \text{fv}(\Gamma)$. Then by Lemma 2.14 we can apply R to the derivations to obtain

$$\begin{gathered} \Gamma \vdash e_1 : R(\tau_1) \\ \text{and } \Gamma[x : \forall \vec{v}.R(\tau_1)] \vdash e_2 : \tau_2. \end{gathered}$$

Next we apply the induction hypothesis to get suitable derivations of both of these, and hence apply the (let′) rule using only naive generalisation:

$$\frac{\Gamma \vdash e_1 : \forall \vec{v}.R(\tau_1) \qquad \Gamma[x : \forall \vec{v}.R(\tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\text{let}')$$

□

This normalisation process is used in Chapter 5 to facilitate a translation from the implicitly-typed language ML_δ described here into an explicitly-typed language Λ_δ.

Chapter 3

Dimension type inference

Type inference is the process of finding a valid type τ for an expression e which contains no explicit type information. For ML there is a well-known type inference algorithm, due to Damas and Milner [13], which deduces a *most general* type τ in the sense that any other valid type can be obtained from τ by substituting for its free type variables. In this chapter we show how this algorithm can be adapted to find a most general *dimension type* whose valid instances are obtained by substituting for its free type and dimension variables.

A process central to type inference is the *unification* of types. In conventional type inference, syntactic unification is used, but to infer dimension types it is necessary to unify types with respect to the equivalence relation $=_D$ defined in the last chapter. In Section 3.1 we study this problem, beginning with a general overview of equational unification. For the particular case of dimension types, it turns out that the unification problem shares with syntactic unification the important property that for any two types which are unifiable there is a unique most general unifier. Armed with this fact, in Section 3.2 we show that the inference algorithm finds the most general type of any typable expression, or reports failure if no type exists. Then in Section 3.3 we look at the problem of finding a unique form for displaying principal types, which is an important feature of a real implementation. It turns out that the simplification algorithm used for this purpose can also be used to calculate the invertible substitution required by the generalisation procedure during type inference. Related type systems are described in Section 3.4. Finally we discuss some refinements to the system which would allow more expressions to be typed: the possibility of providing polymorphism in recursive definitions, the extra power that dependent types would give, and the increase in expressiveness permitted by polymorphism in function arguments and results. These motivate the introduction of a more richly-typed language in Chapter 5.

3.1 Unification

In order to construct a typing derivation working backwards from the conclusion to the premises it is necessary to guess some of the types of the premises, by

generating fresh type and dimension variables, and then *unify* appropriate types to obtain a valid typing. This process of unification is at the heart of ML type inference.

The dimension type system differs from ML in one important respect: types are identified *up to equivalence under* $=_D$. Therefore the inference algorithm must unify types up to this equivalence, in contrast to the free unification employed in conventional ML type inference. This use of *equational* unification in a type inference algorithm is unusual; another example is in Thatte's automatic generation of coercions between data representations [62].

Syntactic *vs* semantic unification

The problem of syntactic, or *free*, unification is the following. Given two terms τ_1 and τ_2, find a substitution S such that $S(\tau_1) = S(\tau_2)$. This substitution is called a *unifier* of τ_1 and τ_2. Syntactic unification has the property that if any unifier exists at all, then there is a unique *most general unifier* S from which any other unifier can be derived. Formally, for any unifier S' there is a substitution R such that $S' = R \circ S$.

Syntactic unification is decidable; the original algorithm is due to Robinson [59] and there have been many more efficient algorithms designed since.

Given an equational theory E such as the one described here for dimension expressions, one can pose a *semantic* or *equational* unification problem by replacing ordinary syntactic equality with the equivalence relation induced by the theory (written $=_E$). In the general case, we are given a set of *E-unification problems* each of which is a pair of terms τ_1 and τ_2. Then a substitution S is an *E-unifier* of this set if $S(\tau_1) =_E S(\tau_2)$ for every pair (τ_1, τ_2) in the set.

Sets of most general unifiers

In contrast to the free theory, for many equational theories unique most general unifiers do not exist. An example is the equational theory C (standing for *commutativity*) induced by the equation

$$\tau_1 + \tau_2 =_C \tau_2 + \tau_1.$$

Consider the unification problem

$$x_1 + x_2 \stackrel{?}{=}_C c_1 + c_2$$

where x_1 and x_2 are variables, and c_1 and c_2 are constants. This has two unifiers, $\{x_1 \mapsto c_1, x_2 \mapsto c_2\}$ and $\{x_1 \mapsto c_2, x_2 \mapsto c_1\}$, neither of which is more general than the other.

Instead of a single most general unifier, for many E-unification problems there is a *set* of most general unifiers from which any other unifier can be derived [27]. To formalise this notion we make some preliminary definitions. We say that $S_1 \leqslant_E S_2$ (substitution S_1 "is more general than" substitution S_2) if there is a substitution R such that $R \circ S_1 =_E S_2$. The notation $S_1 \equiv_E S_2$ means that $S_1 \leqslant_E S_2$ *and* $S_2 \leqslant_E S_1$.

Then a set of unifiers $\mathcal{U}$ is *complete* if for any unifier S' there is some $S \in \mathcal{U}$ such that $S \leqslant_E S'$. This just says that the elements of $\mathcal{U}$ generate all unifiers by instantiation.

Finally, $\mathcal{U}$ is said to be a complete set of *most general unifiers* if in addition to completeness it has the property that for any two unifiers $S_1, S_2 \in \mathcal{U}$ such that $S_1 \leqslant_E S_2$ it is the case that $S_1 = S_2$.

Classification of theories

The notion of a complete set of most general unifiers $\mathcal{U}$ can be used to classify equational theories in the following way.

- If $\mathcal{U}$ always has at most one element then the theory is *unitary*.
- If $\mathcal{U}$ is always finite then the theory is *finitary*.
- If $\mathcal{U}$ is possibly infinite then the theory is *infinitary*.

By this classification, free unification is unitary.

The presence of free constant symbols (such as the base dimensions in a dimension δ) and free function symbols (such as arrows in a type τ) complicates matters. It can affect the classification of a theory: the equational theory $AC1$ containing axioms for associativity, commutativity and identity is unitary in the absence of free constants but becomes finitary when constants are introduced. A unification problem is called *pure* if it does not contain constants; if constants are present then it is called *applied*.

The equational theory for dimension types

The equations defining $=_D$ over dimensions are precisely the axioms of Abelian groups. We are very fortunate in that unification is *unitary* for Abelian groups with free nullary constants (our base dimensions). So if a unifier exists at all then there is a unique most general unifier from which all others can be derived by instantiation. Furthermore, there is an algorithm which will find this most general unifier if it exists, or report failure if it does not. Before describing this algorithm, it is worth mentioning that some survey papers on unification classify Abelian group unification as *finitary* [60]. However, this refers to unification in the presence of *arbitrary* free constants, which may include function symbols. Consider the unification problem

$$x \cdot (y \oplus z) \stackrel{?}{=}_D w \cdot (a \oplus b)$$

where w, x, y and z are variables, a and b are free constants, and $\oplus$ is a free binary function symbol. Then the unifiers

$$\{x \mapsto w,\ y \mapsto a,\ z \mapsto b\}$$

and

$$\{x \mapsto a \oplus b,\ w \mapsto y \oplus z\}$$

are incomparable and there is no unifier which is more general than both of them.

Take away the free function symbols, and the problem becomes unitary [2, 8]. In type expressions such as $\mathsf{real}\, d_1 \to \mathsf{real}\, d_2$ we *do* have free function symbols: the unary real and binary $\to$. However, the stratification of the syntax into dimensions and types means that these cannot occur inside a dimension expression such as that in the type $\mathsf{real}\,(d_1 \to d_2)$. This ensures that the problem remains unitary even at the level of types.

Unification of dimensions

First we note that the unification of two dimensions δ_1 and δ_2 can be reduced to the unification of $\delta_1 \cdot \delta_2^{-1}$ and $\mathbf{1}$. Therefore we will refer to the "unification of δ" to mean finding a substitution S such that $S(\delta) =_D \mathbf{1}$. Suppose that the normal form of δ is

$$d_1^{x_1} \cdots d_m^{x_m} \cdot \mathrm{B}_1^{y_1} \cdots \mathrm{B}_n^{y_n}.$$

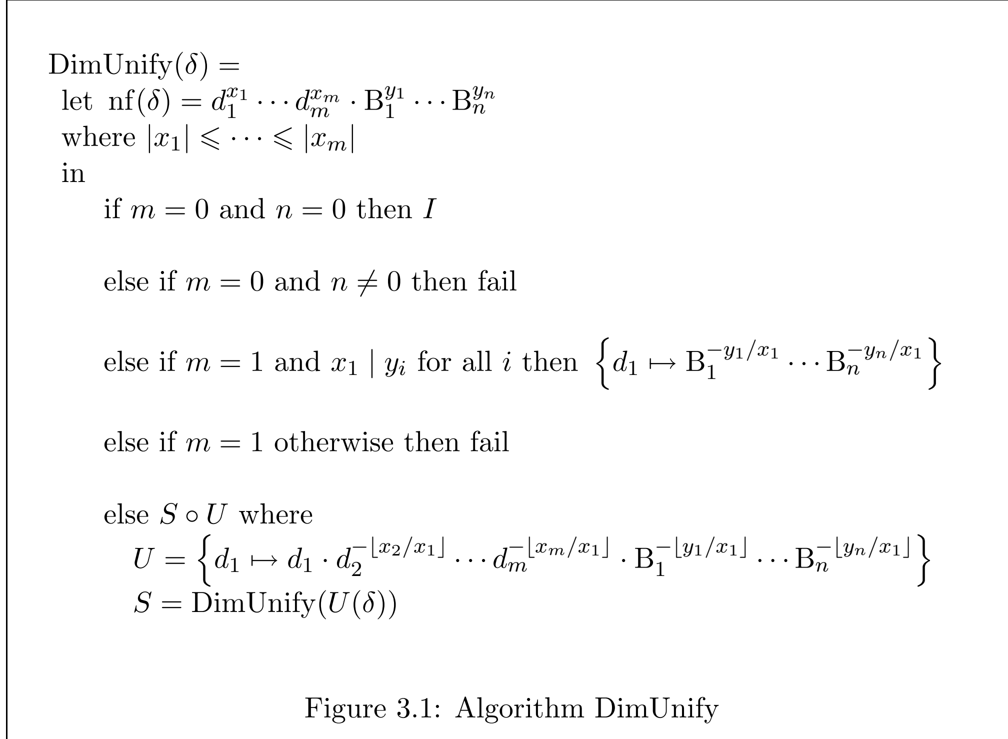
In fact the constant part can be reduced to B^g, where $g = \gcd(y_1, \ldots, y_n)$. Then the problem of unification can be seen to be equivalent to the problem of finding all solutions in integers $z_1, \ldots, z_m$ to the linear equation

$$x_1 z_1 + \cdots + x_m z_m + g = 0.$$

There are standard algorithms to solve this problem [48, 32]; we use Knuth's, which is essentially an adaptation of Euclid's greatest common divisor algorithm. Lankford, Butler and Brady were the first to apply such algorithms to unification in Abelian groups [36], but they did not consider any particular algorithm in detail. We take a more direct approach and present an algorithm and its proof of correctness.

The unification algorithm DimUnify is shown in Figure 3.1. Although it is presented as a recursively-defined function, it is easier to think of it iteratively. A substitution is built up incrementally, on each iteration reducing the size of the smallest exponent in the dimension expression by means of an invertible substitution. When only one variable remains ($m = 1$), the algorithm either completes the unifier by substituting an appropriate constant dimension or reports failure if no such substitution exists. Take care to note that the form of δ changes for each iteration: our identifiers $d_1, \ldots, d_m$ may refer to different actual dimension variables on different iterations.

The correctness of the algorithm divides into two. First we show that when the algorithm terminates with a substitution S then this is indeed a unifier (*soundness*). Then we show that whenever a unifier exists then this is an instance of the most general unifier returned by the algorithm (*completeness*).



DimUnify$(\delta) =$
let $\mathrm{nf}(\delta) = d_1^{x_1} \cdots d_m^{x_m} \cdot \mathrm{B}_1^{y_1} \cdots \mathrm{B}_n^{y_n}$
where $|x_1| \leqslant \cdots \leqslant |x_m|$
in
if $m = 0$ and $n = 0$ then I

else if $m = 0$ and $n \neq 0$ then fail

else if $m = 1$ and $x_1 \mid y_i$ for all i then $\left\{d_1 \mapsto \mathrm{B}_1^{-y_1/x_1} \cdots \mathrm{B}_n^{-y_n/x_1}\right\}$

else if $m = 1$ otherwise then fail

else $S \circ U$ where
$U = \left\{d_1 \mapsto d_1 \cdot d_2^{-\lfloor x_2/x_1 \rfloor} \cdots d_m^{-\lfloor x_m/x_1 \rfloor} \cdot \mathrm{B}_1^{-\lfloor y_1/x_1 \rfloor} \cdots \mathrm{B}_n^{-\lfloor y_n/x_1 \rfloor}\right\}$
$S = \mathrm{DimUnify}(U(\delta))$

Figure 3.1: Algorithm DimUnify

Theorem 3.1 (Soundness and completeness of DimUnify**).** *For any δ the algorithm* DimUnify(δ) *terminates with failure or with a substitution S, and:*

- *If* DimUnify$(\delta) = S$ *then* $S(\delta) =_D \mathbf{1}$.
- *If* $S'(\delta) = \mathbf{1}$ *then* DimUnify$(\delta) = S$ *such that* $S' =_D R \circ S$ *for some substitution R.*

Proof. By induction on $\mu(\delta)$, where μ is the following function (with m and x_1 in δ defined as in Figure 3.1):

$$\mu(\delta) = \begin{cases} 0 & \text{if } m = 0 \text{ or } m = 1, \\ |x_1| & \text{otherwise.} \end{cases}$$

- If $m = 0$ then the result follows immediately.
- If $m = 1$ and x_1 divides all of $\{y_1, \ldots, y_n\}$ then clearly the substitution

  $$S = \left\{d_1 \mapsto \mathrm{B}_1^{-y_1/x_1} \cdots \mathrm{B}_n^{-y_n/x_1}\right\}$$

  is a unifier as required. For the second part of the result, the arbitrary unifier S' must satisfy

  $$S'(d_1^{x_1}) =_D \mathrm{B}_1^{-y_1} \cdots \mathrm{B}_n^{-y_n}.$$

  Clearly this can only be the case if x_1 divides all of $\{y_1, \ldots, y_n\}$. Now let R be a substitution equal to S' except at d_1 where $R(d_1) = d_1$. Then

$$\text{Unify}(\mathsf{bool}, \mathsf{bool}) = \text{Unify}(t, t) \quad = \quad I$$

$$\text{Unify}(t, \tau) = \text{Unify}(\tau, t) \quad = \quad \begin{cases} \text{fail} & \text{if } t \in \text{ftv}(\tau), \\ \{t \mapsto \tau\} & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \text{Unify}(\tau_1 \to \tau_2,\, \tau_3 \to \tau_4) \quad = \quad & S_2 \circ S_1 \\ & \text{where } S_1 = \text{Unify}(\tau_1, \tau_3) \\ & \text{and } S_2 = \text{Unify}(S_1(\tau_2), S_1(\tau_4)) \end{aligned}$$

$$\text{Unify}(\mathsf{real}\,\delta_1, \mathsf{real}\,\delta_2) \quad = \quad \text{DimUnify}(\delta_1 \cdot \delta_2^{-1})$$

$$\text{Unify}(?, ?) \quad = \quad \text{fail}$$

Figure 3.2: Algorithm Unify

$S' =_D R \circ S$ where S is the substitution returned by the algorithm, shown above.

- If $m > 1$ then by a simple calculation we obtain

$$U(\delta) =_D d_1^{x_1} \cdot d_2^{x_2 \bmod x_1} \cdots d_m^{x_m \bmod x_1} \cdot \mathrm{B}_1^{y_1 \bmod x_1} \cdots \mathrm{B}_n^{y_n \bmod x_1}.$$

Now if x_1 divides all of $\{x_2, \ldots, x_m\}$ then $\mu(U(\delta)) = 0$. Otherwise there is some non-zero ($x_i \bmod x_1$) smaller in magnitude than x_1 so $\mu(U(\delta)) < \mu(\delta)$. Hence we can apply the induction hypothesis and deduce that $S(U(\delta)) =_D \mathbf{1}$. For completeness we use the fact that invertible substitutions preserve most general unifiers. Observe that U is invertible, as any substitution of the form $\{d \mapsto d \cdot \delta\}$ with $d \notin \text{fdv}(\delta)$ has inverse $\{d \mapsto d \cdot \delta^{-1}\}$. Now consider an arbitrary unifier S' of δ. Then $S' \circ U^{-1}$ is a unifier of $U(\delta)$. By the induction hypothesis there must be some substitution R such that $S' \circ U^{-1} =_D R \circ S$. Hence $S' =_D R \circ S \circ U$ as required.

□

Unification of types

Any standard algorithm for syntactic unification can be extended to support dimension unification simply by adding a new clause which unifies δ_1 and δ_2 whenever it is required to unify $\mathsf{real}\,\delta_1$ and $\mathsf{real}\,\delta_2$. A naive but concise algorithm is shown in Figure 3.2. It is straightforward to prove that this algorithm does compute the most general unifier of two types if one exists, making use of Theorem 3.1 in the base case.

Theorem 3.2 (Soundness and completeness of Unify**).** *For any two type expressions* τ_1 *and* τ_2 *the algorithm* $\text{Unify}(\tau_1, \tau_2)$ *terminates with failure or with a substitution* S*, and:*

- *If* $\text{Unify}(\tau_1, \tau_2) = S$ *then* $S(\tau_1) =_D S(\tau_2)$*.*
- *If* $S'(\tau_1) =_D S'(\tau_2)$ *then* $\text{Unify}(\tau_1, \tau_2) = S$ *such that* $S' =_D R \circ S$ *for some substitution* R*.*

Proof. By induction on the structure of τ_1. □

3.2 Inference

Figure 3.3 shows the inference algorithm. In essence it is the same as Damas and Milner's original algorithm [13]. We have omitted the clauses for conditionals and for recursion, as these can be expressed easily by extending the type assignment with constants *cond* and *fix* of appropriate type.

Usually the generation of fresh type variables is left implicit, and would be implemented most naturally using imperative features of a programming language. Unfortunately this makes a thorough proof of completeness rather intricate in its treatment of free variables and substitutions. Instead, the algorithm presented here is purely functional, maintaining a set of 'used' type and dimension variables so that fresh variables are generated away from this set.

The algorithm operates as follows. Given a type assignment Γ, an expression e and a set of type and dimension variables $\mathcal{V}$ with $\text{fv}(\Gamma) \subseteq \mathcal{V}$, then $\text{Infer}(\mathcal{V}, \Gamma, e)$ either indicates failure or returns a triple $(\mathcal{V}', S, \tau)$ consisting of an updated set of used variables $\mathcal{V}' \supseteq \mathcal{V}$, a substitution S and a type τ. The following *soundness* result states that a successful outcome $(\mathcal{V}', S, \tau)$ implies that there is a valid typing derivation $S(\Gamma) \vdash_{\text{sd}} e : \tau$.

Theorem 3.3 (Soundness of Infer**).** *If* $\text{Infer}(\mathcal{V}, \Gamma, e) = (\mathcal{V}', S, \tau)$ *then there is a typing derivation* $S(\Gamma) \vdash_{\text{sd}} e : \tau$*.*

Proof. See Appendix C. □

The *completeness* result states that if a valid typing exists then it is an instance of the typing returned by the algorithm. To be precise, for any valid typing $S(\Gamma) \vdash_{\text{sd}} e : \tau$ with $\text{fv}(\Gamma) \subseteq \mathcal{V}$ the algorithm will return a triple $(\mathcal{V}', S_0, \tau_0)$ such that $S =_D R \circ S_0$ and $\tau =_D R(\tau_0)$ for some substitution R.

Theorem 3.4 (Completeness of Infer**).** *If* $S(\Gamma) \vdash_{\text{sd}} e : \tau$ *and* $\text{fv}(\Gamma) \subseteq \mathcal{V}$ *then* $\text{Infer}(\mathcal{V}, \Gamma, e)$ *succeeds with result* $(\mathcal{V}', S_0, \tau_0)$ *such that*

$$S =_D R \circ S_0 \text{ and } \tau =_D R(\tau_0)$$

for some substitution R*.*

Proof. See Appendix C. □

$\text{Infer}(\mathcal{V}, \Gamma, x) = (\mathcal{V} \cup \vec{v'}, I, \{\vec{v} \mapsto \vec{v'}\}\tau)$
where
$\Gamma(x)$ is $\forall \vec{v}.\tau$
$\vec{v'}$ are type and dimension variables not in $\mathcal{V}$

$\text{Infer}(\mathcal{V}, \Gamma, 0) = (\mathcal{V} \cup \{d\}, I, \mathsf{real}\, d)$
where
d is a dimension variable not in $\mathcal{V}$

$\text{Infer}(\mathcal{V}, \Gamma, r) = (\mathcal{V}, I, \mathsf{real}\, \mathbf{1})$

$\text{Infer}(\mathcal{V}, \Gamma, e_1\, e_2) = (\mathcal{V}_2 \cup \{t\}, (S_3 \circ S_2 \circ S_1)|_{\mathcal{V}}, S_3(t))$
where
$(\mathcal{V}_1, S_1, \tau_1) = \text{Infer}(\mathcal{V}, \Gamma, e_1)$
$(\mathcal{V}_2, S_2, \tau_2) = \text{Infer}(\mathcal{V}_1, S_1(\Gamma), e_2)$
$S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \to t)$
t is a type variable not in $\mathcal{V}_2$

$\text{Infer}(\mathcal{V}, \Gamma, \lambda x.e) = (\mathcal{V}', S|_{\mathcal{V}}, S(t) \to \tau)$
where
$(\mathcal{V}', S, \tau) = \text{Infer}(\mathcal{V} \cup \{t\}, \Gamma[x : t], e)$
t is a type variable not in $\mathcal{V}$

$\text{Infer}(\mathcal{V}, \Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = (\mathcal{V}_2, S_2 \circ S_1, \tau_2)$
where
$(\mathcal{V}_1, S_1, \tau_1) = \text{Infer}(\mathcal{V}, \Gamma, e_1)$
$(\mathcal{V}_2, S_2, \tau_2) = \text{Infer}(\mathcal{V}_1, S_1(\Gamma)[x : \sigma], e_2)$
$\sigma = \text{Gen}(S_1(\Gamma), \tau_1)$

$\text{InferScheme}(\Gamma, e) = (S_0, \sigma_0)$
where
$(-, S_0, \tau_0) = \text{Infer}(\text{fv}(\Gamma), \Gamma, e)$
$\sigma_0 = \text{Gen}(S_0(\Gamma), \tau_0)$

Figure 3.3: Algorithm Infer

These results can also be expressed in terms of type schemes. The function InferScheme shown in Figure 3.3 applies the inference algorithm and then uses Gen to obtain a type scheme. We introduce some terminology: σ is a *principal type scheme* of e under Γ if

$$\Gamma \vdash e : \tau \text{ if and only if } \sigma \preccurlyeq_D \tau.$$

Then the soundness and completeness results can be re-expressed as follows.

Corollary 3.5 (Soundness of InferScheme**).** *If* $\text{InferScheme}(\Gamma, e) = (S_0, \sigma_0)$ *then* σ_0 *is a principal type scheme of* e *under* $S_0(\Gamma)$*.*

Proof. Let $\text{Infer}(\text{fv}(\Gamma), \Gamma, e) = (S_0, \tau_0)$ and $\sigma_0 = \text{Gen}(S_0(\Gamma), \tau_0)$. Then both directions of the equivalence are proved as follows.

($\Rightarrow$). If $S_0(\Gamma) \vdash e : \tau$ then from Theorem 3.4 (completeness of Infer) there is a substitution R such that $S_0 =_D R \circ S_0$ and $\tau =_D R(\tau_0)$. Hence $R(S_0(\Gamma)) =_D S_0(\Gamma)$ so by Proposition 2.7 $\text{Gen}(S_0(\Gamma), \tau_0) \preccurlyeq_D \tau$ as required.

($\Leftarrow$). If $\text{Gen}(S_0(\Gamma), \tau_0) \preccurlyeq_D \tau$ then there is some substitution R such that $R(S_0(\Gamma)) \cong_D S_0(\Gamma)$ and $R(\tau_0) =_D \tau$. From Theorem 3.3 (soundness of Infer) there must be a derivation of $S_0(\Gamma) \vdash e : \tau_0$. Then applying R to this derivation (by Lemma 2.14) we obtain $S_0(\Gamma) \vdash e : \tau$ as required. □

Corollary 3.6 (Completeness of InferScheme**).** *If the scheme* σ *is a principal type scheme of* e *under* $S(\Gamma)$ *then* $\text{InferScheme}(\Gamma, e) = (S_0, \sigma_0)$ *such that*

$$R(S_0(\Gamma)) \cong_D S(\Gamma) \text{ and } R(\sigma_0) \preccurlyeq_D \sigma$$

for some substitution R*.*

Proof. Again let $\text{Infer}(\text{fv}(\Gamma), \Gamma, e) = (S_0, \tau_0)$ and $\sigma_0 = \text{Gen}(S_0(\Gamma), \tau_0)$. Also, if σ is a principal type scheme of e under $S(\Gamma)$ then $\sigma \cong_D \text{Gen}(S(\Gamma), \tau)$ for some type τ. Applying the completeness result for Infer to the typing $S(\Gamma) \vdash e : \tau$ we obtain a substitition R such that

$$S =_D R \circ S_0 \text{ and } \tau =_D R(\tau_0).$$

Clearly $R(S_0(\Gamma)) \cong_D S(\Gamma)$ as required. Now consider $R(\sigma_0)$:

$$\begin{aligned} R(\text{Gen}(S_0(\Gamma), \tau_0)) &\preccurlyeq_D \text{Gen}(R(S_0(\Gamma)), R(\tau_0)) && \text{by Lemma 2.9} \\ &\cong_D \text{Gen}(S(\Gamma), \tau) && \text{from above} \\ &\cong_D \sigma. \end{aligned}$$

□

When the type assignment Γ is closed, these results show that InferScheme finds a principal type scheme for e under Γ if e is typable at all. This scheme is unique up to type scheme equivalence. In the next section we show how it is always possible to find a canonical representative in each equivalence class.

3.3 Simplification of types

In this section we present a type *simplification* algorithm which serves two purposes:

1. When applied to the *bound* variables of a type scheme it determines a canonical form for the type scheme under the equivalence relation $\cong_D$. For the programmer, this is important: the canonical form is natural in an intuitive sense and is unique—so types can be displayed in a consistent way by the inference algorithm. Furthermore, one effect of simplification is that non-essential free variables are removed, a necessary precondition for the correct operation of the generalisation procedure $\mathrm{Gen}(\Gamma, \tau)$.

2. When applied to the *free* variables of a type scheme σ the algorithm determines an invertible substitution U such that $U(\sigma)$ is in free variable reduced form. This procedure extends straightforwardly to type assignments where it is used to calculate $\mathrm{Gen}(\Gamma, \tau)$ according to Definition 2.5.

The original motivation for designing this algorithm was the need for a 'most natural' form for type schemes. This is the way that we describe the algorithm initially. However, it turned out that the canonical form which it calculates corresponds directly to a well-known form in matrix theory. This in turn explains its application to the generalisation procedure as a 'change of basis'.

Equivalent type schemes

We have seen already how a single type scheme can have a variety of equivalent forms under the equivalence relation $\cong_D$. The inference algorithm as presented makes no guarantees about the final form of the inferred type, and it probably depends on the particular order in which unification of types is performed. In terms of typable programs, this is not a problem, but if types are displayed to the programmer then we must choose a suitable representation. For ordinary ML, this merely involves choosing a set of names for the bound type variables, typically generating names in the order in which the type variables first appear in the type, reading from left to right. For dimension types, this is not enough.

Consider the following type which an early prototype implementation assigned to the `correlation` function from Chapter 1:

$$\forall d_1.\forall d_2.\mathsf{real}\ d_1\ \mathsf{list} \to \mathsf{real}\ d_1^{-1} \cdot d_2\ \mathsf{list} \to \mathsf{real}\ \mathbf{1}.$$

This is equivalent under $\cong_D$ to

$$\forall d_1.\forall d_2.\mathsf{real}\ d_1\ \mathsf{list} \to \mathsf{real}\ d_2\ \mathsf{list} \to \mathsf{real}\ \mathbf{1}.$$

The second of these makes more sense as it expresses the symmetry inherent in the type. Sometimes the choice is not so clear. For example, the differentation function of Chapter 1 can be assigned these equivalent types, amongst many:

$$\begin{aligned}&\forall d_1.\forall d_2.\mathsf{real}\ d_1 \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2) \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^{-1} \cdot d_2)\\ \cong_D &\forall d_1.\forall d_2.\mathsf{real}\ d_1 \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_1 \cdot d_2) \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2).\end{aligned}$$

We must decide what makes one type scheme 'simpler' than an equivalent type scheme. We can confidently state the following:

1. The fewer the dimension variables, the better. For example, this favours $\forall d.\mathsf{real}\ d \rightarrow \mathsf{real}\ d^2$ over $\forall d_1.\forall d_2.\mathsf{real}\ d_1 \cdot d_2 \rightarrow \mathsf{real}\ d_1^2 \cdot d_2^2$. Here the number of bound dimension variables in the simpler form matches the number of 'degrees of freedom' in the polymorphic part of the type scheme: we surely require at least this of a canonical form.

   Now consider the two types for `correlation` given above, where the number of bound variables cannot be reduced. Nevertheless, the 'asymmetric' type has more dimension variables in its second argument than does the 'symmetric' type, so the heuristic makes sense here too.

   Finally, simplification should remove any non-essential *free* variables from the type scheme, for example $\forall d_1.\mathsf{real} d_1 \cdot d_2 \rightarrow \mathsf{real} d_1 \cdot d_2$ should be reduced to $\forall d_1.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1$.

2. Positive exponents are simpler than negative ones. Thus $\forall d.\mathsf{real}\, d \rightarrow \mathsf{real}\, d^2$ is simpler than the equivalent $\forall d.\mathsf{real}\ d^{-1} \rightarrow \mathsf{real}\ d^{-2}$.

3. Small exponents are simpler than large exponents.

We now suggest a more arguable rule:

4. In a type $\tau_1 \rightarrow \tau_2$, give precedence to the simplification of τ_1 over τ_2. For example, this favours the type scheme

   $$\forall d_1.\forall d_2.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_2 \rightarrow \mathsf{real}\ d_1 \cdot d_2^{-2}$$

   over the equivalent scheme

   $$\forall d_1.\forall d_2.\mathsf{real}\ d_1 \cdot d_2 \rightarrow \mathsf{real}\ d_2 \rightarrow \mathsf{real}\ d_1 \cdot d_2^{-1}.$$

   Similarly, the first type for `diff` from above is regarded as simpler than the second.

Putting these heuristics together, one arrives at the following sketch for an algorithm:

> Simplify a type τ from left to right, and for each dimension component δ attempt to reduce as much as possible the number of dimension variables, the size of its exponents, and the number of negative exponents.

The realisation of the sketch is two functions shown in Figures 3.4 and 3.5. We describe them each in turn.

DimSimplify$(\mathcal{V}, \delta) =$
let $\mathrm{nf}(\delta) = d_1^{x_1} \cdots d_m^{x_m} \cdot v_1^{y_1} \cdots v_n^{y_n}$
where each v_i is a variable from $\mathcal{V}$ or a base dimension, no d_i is in $\mathcal{V}$
and $|x_1| \leqslant \cdots \leqslant |x_m|$
in
if $m = 0$ then I

else if $x_1 < 0$ then $U_2 \circ U_1$ where
$U_1 = \{d_1 \mapsto d_1^{-1}\}$
$U_2 = \mathrm{DimSimplify}(\mathcal{V}, U_1(\delta))$

else if $m = 1$ then
$\left\{d_1 \mapsto d_1 \cdot v_1^{-\lfloor y_1/x_1 \rfloor} \cdots v_n^{-\lfloor y_n/x_1 \rfloor}\right\}$

else $U_2 \circ U_1$ where
$U_1 = \left\{d_1 \mapsto d_1 \cdot d_2^{-\lfloor x_2/x_1 \rfloor} \cdots d_m^{-\lfloor x_m/x_1 \rfloor} \cdot v_1^{-\lfloor y_1/x_1 \rfloor} \cdots v_n^{-\lfloor y_n/x_1 \rfloor}\right\}$
$U_2 = \mathrm{DimSimplify}(\mathcal{V}, U_1(\delta))$

Figure 3.4: Algorithm DimSimplify

Simplifying a dimension

The function DimSimplify in Figure 3.4 takes a dimension δ and a set of dimension variables $\mathcal{V}$, and returns an invertible substitution which simplifies δ as much as possible without substituting for any of the variables in $\mathcal{V}$. If fact, 'as much as possible' means reducing the part of δ not contained in $\mathcal{V}$ to at most a single dimension variable d with positive exponent x. Furthermore, if such a variable d exists then the exponents of all other variables in δ are positive and smaller than x. The following lemma provides formal justification.

Lemma 3.7 (DimSimplify**).** *For any dimension δ and set of dimension variables $\mathcal{V}$, the algorithm* DimSimplify$(\mathcal{V}, \delta)$ *returns an invertible substitution U such that* $\mathrm{dom}(U) \cap \mathcal{V} = \emptyset$ *and the normal form of $U(\delta)$ is either*

$$v_1^{y_1} \cdots v_n^{y_n}$$

where all v_i are base dimensions or dimension variables from $\mathcal{V}$, or

$$d^x \cdot v_1^{y_1} \cdots v_n^{y_n}$$

for some dimension variable $d \notin \mathcal{V}$ and all v_i either base dimensions or variables from $\mathcal{V}$, with $x > 0$ and $0 < y_i < x$ for $1 \leqslant i \leqslant n$.

Proof. By induction on $\mu(\mathcal{V},\delta)$, where μ is the following function (with m and x_1 defined as in Figure 3.4):

$$\mu(\mathcal{V},\delta) = \begin{cases} 0 & \text{if } m = 0 \text{ or } m = 1 \text{ and } x_1 > 0, \\ 1 & \text{if } m = 1 \text{ and } x_1 < 0, \\ 2|x_1| & \text{if } m > 1 \text{ and } x_1 > 0, \\ 2|x_1| + 1 & \text{if } m > 1 \text{ and } x_1 < 0. \end{cases}$$

- The case for $m = 0$ is trivial.
- When $x_1 < 0$ it is clear that for the invertible substitution $U_1 = \{d_1 \mapsto d_1^{-1}\}$ we have $\mu(\mathcal{V}, U_1(\delta)) = \mu(\mathcal{V},\delta) - 1$, so we can apply the induction hypothesis and obtain the required result.
- When $m = 1$ and $x_1 > 0$,

  $$U(\delta) =_D d_1^{x_1} \cdot v_1^{y_1 \bmod x_1} \cdots v_n^{y_n \bmod x_1}.$$

  Then $y_i \bmod x_1 < x_1$ for all i so this has the second of the forms required by the lemma. Furthermore, it is easy to see that U is invertible and $\mathrm{dom}(U) \cap \mathcal{V} = \emptyset$.
- When $m > 0$ then the normal form of $U_1(\delta)$ is

  $$d_1^{x_1} \cdot d_2^{x_2 \bmod x_1} \cdots d_m^{x_m \bmod x_1} \cdot v_1^{y_1 \bmod x_1} \cdots v_n^{y_n \bmod x_1}.$$

  Clearly the smallest exponent in $U_1(\delta)$ is smaller than $|x_1|$, or else only the variable d_1 is left, so that $\mu(\mathcal{V}, U_1(\delta)) < \mu(\mathcal{V},\delta)$. Then the induction hypothesis can be applied to show that $U_2(U_1(\delta))$ has the required form. Invertibility of $U_2 \circ U_1$ is obtained from the induction hypothesis and the fact that U_1 is invertible.

□

The algorithms DimSimplify and DimUnify are closely related. Both apply the same invertible substitution when iterating, except that DimSimplify can also change the sign of the smallest exponent. They differ in the way that they finish: whereas Simplify terminates when there is just one variable remaining, Unify completes the unification process by removing the variable and the rest of the term if it can, or reports failure if it cannot.

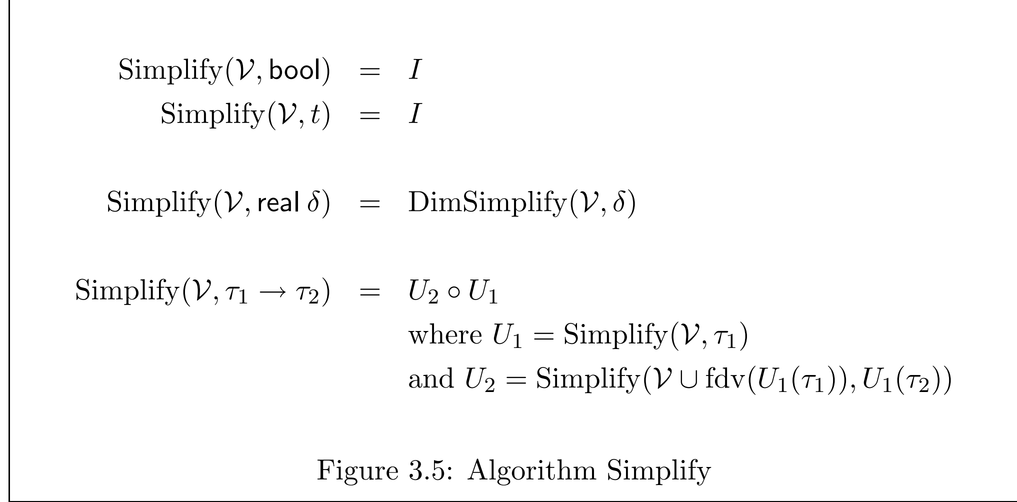


Figure 3.5: Algorithm Simplify

Example. Consider the (rather unlikely) dimension $\delta = d_1^6 \cdot d_2^{15} \cdot d_3^{-7} \cdot d_4^{12}$ and set of variables $\mathcal{V} = \{d_3, d_4\}$. Then DimSimplify would proceed to simplify δ away from $\mathcal{V}$ by the following composition of invertible substitutions:

$$
\begin{array}{c}
d_1^6 \cdot d_2^{15} \cdot d_3^{-7} \cdot d_4^{12} \\
\downarrow {\scriptstyle d_1 \mapsto d_1 \cdot d_2^{-2} \cdot d_3^{2} \cdot d_4^{-2}} \\
d_1^6 \cdot d_2^3 \cdot d_3^5 \\
\downarrow {\scriptstyle d_2 \mapsto d_2 \cdot d_1^{-2} \cdot d_3^{-1}} \\
d_2^3 \cdot d_3^2 \\
\downarrow {\scriptstyle d_2 \mapsto d_2} \\
d_2^3 \cdot d_3^2.
\end{array}
$$

Simplifying a type

The function Simplify in Figure 3.5 calculates a simplifying substitution for τ by traversing the type from left to right without touching the variables in $\mathcal{V}$, taking care not to change any already-simplified dimensions as it traverses the type.

The lemma which follows shows first that the substitution is invertible and away from $\mathcal{V}$: this ensures that it can be used to simplify a type scheme (away from the free variables) and as a change of basis prior to generalisation (away from the bound variables). Then the gist of proposition (1) below is that the variables from $\mathcal{V}$ which remain in the simplified type are *essential* in the sense that they cannot be removed by any substitution S away from $\mathcal{V}$. When the algorithm is used to simplify a type scheme then this indeed shows that the only free variables which remain are essential. Proposition (2) says that the variables not in $\mathcal{V}$ which remain in the simplified type are independent with respect to substitution.

Lemma 3.8 (Simplify)**.** *For any type τ and set of dimension variables $\mathcal{V}$, the algorithm* $\text{Simplify}(\mathcal{V}, \tau)$ *returns an invertible substitution U with* $\text{dom}(U) \cap \mathcal{V} = \emptyset$ *and satisfying the following propositions:*

1. *For any dimension substitution S such that* $\text{dom}(S) \subseteq \text{fdv}(U(\tau)) \setminus \mathcal{V}$ *and any $d \in \mathcal{V}$ it is the case that*

$$d \in \text{fdv}(U(\tau)) \textit{ implies } d \in \text{fdv}(S(U(\tau)))$$

   and also

$$d \in \text{fdv}(S(d')) \textit{ for some } d' \in \text{dom}(S) \textit{ implies } d \in \text{fdv}(S(U(\tau))).$$

2. *for any two dimension substitutions S_1 and S_2 such that* $\text{dom}(S_1) \cap \mathcal{V} = \text{dom}(S_2) \cap \mathcal{V} = \emptyset$ *it is the case that*

$$S_1(U(\tau)) =_D S_2(U(\tau)) \textit{ implies } S_1(d) =_D S_2(d) \textit{ for all } d \in \text{fdv}(U(\tau)).$$

Proof. By induction on the structure of τ.

- If $\tau = \mathsf{real}\,\delta$, then from Lemma 3.7 we know that U is invertible and away from $\mathcal{V}$, and that there are two possibilities for the normal form of $U(\tau)$. If $\text{fdv}(U(\tau)) \subseteq \mathcal{V}$ then the results follow trivially. Otherwise, the normal form is given by

$$U(\tau) =_D d_0^x \cdot v_1^{y_1} \cdots v_n^{y_n}$$

  where $v_1, \ldots, v_n$ are all base dimensions or variables in $\mathcal{V}$ and $v_0 \notin \mathcal{V}$. For part (1), consider a substitution S as specified *i.e.* $\text{dom}(S) = \{d_0\}$. If $d = v_i$ for some i, then it is easy to see that S cannot remove it, since $0 < y_i < x$ for $1 \leqslant i \leqslant n$. Otherwise, if $d \in \text{fdv}(S(d_0))$ then clearly $d \in \text{fdv}(S(d_0^x \cdot v_1^{y_1} \cdots v_n^{y_1}))$ as required.

  For part (2), consider two substititions S_1 and S_2 as described. Then

$$\begin{aligned} & S_1(U(\tau)) =_D (S_1(d))^x \cdot v_1^{y_1} \cdots v_n^{y_n} \\ \text{and} \quad & S_2(U(\tau)) =_D (S_2(d))^x \cdot v_1^{y_1} \cdots v_n^{y_n}. \end{aligned}$$

  Hence $S_1(U(\tau)) =_D S_2(U(\tau))$ implies $S_1(d) =_D S_2(d)$ as required.

- If $\tau = \tau_1 \rightarrow \tau_2$ then $U = U_2 \circ U_1$, where

$$U_1 = \text{Simplify}(\mathcal{V}, \tau_1) \quad \text{(a)}$$
$$U_2 = \text{Simplify}(\mathcal{V} \cup \text{fdv}(U_1(\tau_1)), U_1(\tau_2)) \quad \text{(b)}$$

  First note that by applying the induction hypothesis to (b) we know that $\text{dom}(U_2) \cap (\mathcal{V} \cup \text{fdv}(U_1(\tau_1))) = \emptyset$, so $U_2(U_1(\tau_1)) =_D U_1(\tau_1)$.

  For part (1) of the lemma, consider some $d \in \text{fdv}(U(\tau_1 \rightarrow \tau_2))$. If $d \in \text{fdv}(U(\tau_1))$ then by applying the induction hypothesis to (a) we can obtain the result. Otherwise we must have $d \notin \text{fdv}(U(\tau_1))$ but $d \in \text{fdv}(U(\tau_2))$.

Now either $d \in \text{fdv}(S(U(\tau_2)))$ or $d \notin \text{fdv}(S(U(\tau_2)))$. If it is the former, then of course $d \in \text{fdv}(S(U(\tau_1 \to \tau_2)))$ as required. If it is the latter, then it is easy to see that there must be some $d' \in \text{dom}(S)$ such that $d \in \text{fdv}(S(d'))$. Hence by applying the induction hypothesis to (a) we have $d \in \text{fdv}(S(U(\tau_1)))$ so $d \in \text{fdv}(S(U(\tau_1 \to \tau_2)))$ as required.

For part (2) consider substitutions S_1 and S_2 as described. Then

$$S_1(U_2(U_1(\tau_1 \to \tau_2))) =_D S_2(U_2(U_1(\tau_1 \to \tau_2))).$$

Then from the induction hypothesis on (1),

$$S_1(d) =_D S_2(d) \text{ for all } d \in \text{fdv}(U_1(\tau_1)) \setminus \mathcal{V}.$$

Now let S_1' and S_2' be the restrictions of S_1 and S_2 to all variables except those in $\text{fdv}(U_1(\tau_1))$. We can then apply the induction hypothesis to (2) to obtain that

$$S_1'(d) =_D S_2'(d) \text{ for all } d \in \text{fdv}(U_2(U_1(\tau_2))) \setminus \text{fdv}(U_1(\tau_1)) \setminus \mathcal{V}.$$

Putting these together,

$$S_1(d) =_D S_2(d) \text{ for all } d \in \text{fdv}(U_2(U_1(\tau_1 \to \tau_2))) \setminus \mathcal{V}$$

as required.

□

Example. Consider the (extremely unlikely) type[†]

$$\tau = \textsf{real}\, \overbrace{d_1^4 \cdot d_2^6 \cdot d_3^2}^{\delta_1} \to \textsf{real}\, \overbrace{d_1 \cdot d_2^2 \cdot d_3^5}^{\delta_2} \to \textsf{real}\, \overbrace{d_1 \cdot d_2^{-7} \cdot d_3^3}^{\delta_3}.$$

Then $\text{Simplify}(\emptyset, \tau)$ would proceed to simplify τ by the composition of invertible substitutions shown below. Each substitution in the diagram is a single iteration

[†]This example is a translation of the integer matrix on page 16 of Newman's book on integral matrices [46]

$$\begin{aligned} \text{SimplifyScheme}(\forall \vec{d}.\tau) &= \forall \vec{d_0}.U(\tau) \\ \text{where } U &= \text{Simplify}(\mathcal{V}, \tau), \quad \mathcal{V} = \text{fdv}(\tau) \setminus \vec{d} \\ \vec{d_0} &= \text{fdv}(U(\tau)) \setminus \mathcal{V}. \end{aligned}$$

Figure 3.6: Algorithm SimplifyScheme

of DimSimplify, processing the dimension component indicated to its left.

$$\mathsf{real}\ d_1^4 \cdot d_2^6 \cdot d_3^2 \to \mathsf{real}\ d_1 \cdot d_2^2 \cdot d_3^5 \to \mathsf{real}\ d_1 \cdot d_2^{-7} \cdot d_3^3$$

$$(\delta_1) \Big\downarrow d_3 \mapsto d_3 \cdot d_2^{-3} d_1^{-2}$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_1^{-9} \cdot d_2^{-13} \cdot d_3^5 \to \mathsf{real}\ d_1^{-5} \cdot d_2^{-16} \cdot d_3^3$$

$$(\delta_2) \Big\downarrow d_1 \mapsto d_1^{-1}$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_1^9 \cdot d_2^{-13} \cdot d_3^5 \to \mathsf{real}\ d_1^5 \cdot d_2^{-16} \cdot d_3^3$$

$$(\delta_2) \Big\downarrow d_1 \mapsto d_1 \cdot d_2^{-2}$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_1^9 \cdot d_2^5 \cdot d_3^5 \to \mathsf{real}\ d_1^5 \cdot d_2^{-6} \cdot d_3^3$$

$$(\delta_2) \Big\downarrow d_2 \mapsto d_2 \cdot d_1^{-1} \cdot d_3^{-1}$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_1^4 \cdot d_2^5 \to \mathsf{real}\ d_1^{11} \cdot d_2^{-6} \cdot d_3^9$$

$$(\delta_2) \Big\downarrow d_1 \mapsto d_1 \cdot d_2^{-1}$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_1^4 \cdot d_2 \to \mathsf{real}\ d_1^{11} \cdot d_2^{-17} \cdot d_3^9$$

$$(\delta_2) \Big\downarrow d_2 \mapsto d_2 \cdot d_1^{-4}$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1^{79} \cdot d_2^{-17} \cdot d_3^9$$

$$(\delta_3) \Big\downarrow d_1 \mapsto d_1 \cdot d_2$$

$$\mathsf{real}\ d_3^2 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1^{79} \cdot d_2^{62} \cdot d_3^9.$$

Simplifying a type scheme

Figure 3.6 shows how Simplify can be applied to the problem of determining a unique form for type schemes under the equivalence relation $\cong_D$. For simplicity, we consider type schemes with quantification only over dimensions.

First it is clear that simplification preserves type scheme equivalence: by using Lemma 2.3 we have $\forall \vec{d_0}.U(\tau) \cong_D \forall \vec{d}.\tau$ via the invertible substitution U.

Next we show that simplification minimises the number of bound variables in the type scheme so that they correctly match the 'degree of polymorphism' in the scheme. Suppose that there are two ways of obtaining $\forall \vec{d_0}.U(\tau) \preccurlyeq_D \tau'$, that is, two substitutions R_1 and R_2 such that $\text{dom}(R_i) \subseteq \vec{d_0}$ and $R_i(U(\tau)) =_D \tau'$ for $i = 1, 2$. Then by the second part of Lemma 3.8 we know that $R_1 =_D$

R_2. Hence there is only ever one way of specialising a simplified type scheme to a particular type. Now hypothesise some type scheme $\forall \vec{d_1}.\tau_1$ equivalent to $\forall \vec{d_0}.U(\tau)$ but with $|\vec{d_1}| < |\vec{d_0}|$. By the argument above, the set of types to which this scheme specialises is in one-to-one correspondence (up to $=_D$) with the set of substitutions whose domain is $\vec{d_0}$. Clearly there are less substitutions with domain $\vec{d_1}$, so there are not enough to generate the same set of types, in contradiction with our hypothesis. Therefore it must be the case that $|\vec{d_1}| \geqslant |\vec{d_0}|$, so the simplification procedure does indeed minimise the number of bound variables.

We now turn our attention to the free variables in the simplified type scheme. Recall the three definitions of the essential free variables in a type scheme (page 28): those for which some substitution changes the type scheme with respect to type scheme equivalence, those present in the simplified form determined by SimplifyScheme, and those present in all types to which the type scheme specialises. The following lemma proves that all three definitions are equivalent.

Lemma 3.9. *Let* $\sigma = \forall \vec{d}.\tau$ *and* $\sigma_0 = \text{SimplifyScheme}(\sigma) = \forall \vec{d_0}.U(\tau)$ *as shown. Then for any dimension variable* d*,*

$$\begin{aligned} & \{d \mapsto \delta\}\sigma \not\cong_D \sigma \text{ for some } \delta && (1) \\ \Leftrightarrow \quad & d \in \text{fdv}(U(\tau)) \setminus \vec{d_0} && (2) \\ \Leftrightarrow \quad & d \in \text{fdv}(\tau') \text{ for all } \tau' \text{ such that } \sigma \preccurlyeq_D \tau'. && (3) \end{aligned}$$

Proof. To prove (1) $\Rightarrow$ (2) we first note that $\{d \mapsto \delta\}\sigma \not\cong_D \sigma_0$ and then using the fact that substitutions preserve type scheme equivalence deduce that $\{d \mapsto \delta\}\sigma_0 \not\cong_D \sigma_0$. Without loss of generality we can assume that d is not present in $\vec{d_0}$. Then it is clear that

$$\forall \vec{d_0}.U(\tau) \not\cong_D \forall \vec{d_0}.\{d \mapsto \delta\}(U(\tau))$$

only if $d \in \text{fdv}(U(\tau)) \setminus \vec{d_0}$.

Now consider some simple type τ' such that $\forall \vec{d_0}.U(\tau) \preccurlyeq_D \tau'$. Then there is some substitution R with $\text{dom}(R) \subseteq \vec{d_0}$ such that $R(U(\tau)) =_D \tau'$. By the first part of Lemma 3.8, if $d \in \text{fdv}(U(\tau)) \setminus \vec{d_0}$ then $d \in \text{fdv}(\tau')$. As $\sigma \cong_D \sigma_0$ we have shown (2) $\Rightarrow$ (3).

Finally we show that (3) $\Rightarrow$ (1). Trivially we have $\forall \vec{d}.\tau \preccurlyeq_D \tau$ and $\forall \vec{d}.\{d \mapsto \mathbf{1}\}\tau \preccurlyeq_D \{d \mapsto \mathbf{1}\}\tau$. If (3) holds then $\forall \vec{d}.\tau \not\preccurlyeq_D \{d \mapsto \mathbf{1}\}\tau$ because $d \notin \text{fdv}(\{d \mapsto \mathbf{1}\}\tau)$. Hence $\forall \vec{d}.\tau \not\cong_D \forall \vec{d}.\{d \mapsto \delta\}\tau$ for $\delta = \mathbf{1}$, which is proposition (1). $\square$

Example. Consider the type scheme

$$\sigma = \forall d_1.\forall d_2.\tau = \forall d_1.\forall d_2.\mathsf{real}\ d_1 \cdot d_2 \cdot d_3^2 \to \mathsf{real}\ d_1^2 \cdot d_2^2 \cdot d_3^4.$$

Applying Simplify$(\{d_3\}, \tau)$ reduces the type immediately:

$$\begin{array}{c} \mathsf{real}\ d_1 \cdot d_2 \cdot d_3^2 \to \mathsf{real}\ d_1^2 \cdot d_2^2 \cdot d_3^4 \\ \big\downarrow {\scriptstyle d_1 \mapsto d_1 \cdot d_2^{-1} \cdot d_3^{-2}} \\ \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2 \end{array}$$

Hence a simplified scheme $\forall d_1.\mathsf{real}\, d_1 \to \mathsf{real}\, d_1^2$ is obtained, with the non-essential free variable d_3 removed and correctly containing only one bound variable d_1.

One final application of simplification is a syntactic characterisation of type scheme equivalence by an *invertible* substitution: this is a stronger result than Lemma 2.3.

Lemma 3.10 (Type scheme equivalence). $\forall \vec{d_1}.\tau_1 \cong_D \forall \vec{d_2}.\tau_2$ *if and only if there is some invertible substitution* U *with* $\mathrm{dom}(U) \subseteq \vec{d_1}$ *and* $\mathrm{dom}(U^{-1}) \subseteq \vec{d_2}$ *such that* $U(\tau_1) =_D \tau_2$.

Proof. Without loss of generality we can assume that $\vec{d_1}$ and $\vec{d_2}$ are disjoint because we can always apply an invertible renaming substitution first. Then the two directions of the equivalence are proved as follows.

($\Leftarrow$). This direction follows trivially from Lemma 2.3.

($\Rightarrow$). Let $\mathcal{V}_1 = \mathrm{fdv}(\tau_1) \setminus \vec{d_1}$ and $\mathcal{V}_2 = \mathrm{fdv}(\tau_2) \setminus \vec{d_2}$. Then apply the simplification procedure to both schemes to give invertible substitutions

$$\begin{aligned} U_1 &= \mathrm{Simplify}(\mathcal{V}_1, \tau_1) \\ \text{and } U_2 &= \mathrm{Simplify}(\mathcal{V}_2, \tau_2). \end{aligned}$$

The normal forms of $\forall \vec{d_1}.\tau_1$ and $\forall \vec{d_2}.\tau_2$ are $\forall \vec{d_3}.U_1(\tau_1)$ and $\forall \vec{d_4}.U_2(\tau_2)$ respectively, where $\vec{d_3} = \mathrm{fdv}(U_1(\tau_1)) \setminus \mathcal{V}_1$ and $\vec{d_4} = \mathrm{fdv}(U_2(\tau_2)) \setminus \mathcal{V}_2$, as given by Figure 3.6. We have already shown that SimplifyScheme does return a type scheme equivalent to its input, so we can deduce that

$$\forall \vec{d_3}.U_1(\tau_1) \cong_D \forall \vec{d_4}.U_2(\tau_2).$$

Then by Lemma 2.3 there are substitutions R_1 and R_2 such that $R_1(U_1(\tau_1)) =_D U_2(\tau_2)$ and $R_2(U_2(\tau_2)) =_D U_1(\tau_1)$ with $\mathrm{dom}(R_1) \subseteq \vec{d_3}$ and $\mathrm{dom}(R_2) \subseteq \vec{d_4}$. Hence

$$\begin{aligned} R_1(R_2(U_2(\tau_2))) &=_D U_2(\tau_2) \\ \text{and trivially } I(U_2(\tau_2)) &=_D U_2(\tau_2). \end{aligned}$$

Then by the second part of Lemma 3.8 it is clear that $R_1 \circ R_2 =_D I$, and by a symmetric argument $R_2 \circ R_1 =_D I$. Let $U = U_2^{-1} \circ R_1 \circ U_1$ with inverse $U^{-1} = U_1^{-1} \circ R_2 \circ U_2$. This is the invertible substitution required. □

Hermite normal form

For the *uniqueness* of the canonical form determined by Simplify we look to matrix theory. There is an exact correspondence between dimension types and rectangular matrices of integers, between dimension substitutions and square matrices of integers, and between equivalence of closed type schemes and row-equivalence of matrices. This is described more formally in Appendix B; in this section we argue informally that the representative type scheme calculated by SimplifyScheme corresponds to a well-known matrix form.

Consider a type $\tau = \mathsf{real}\ \delta_1 \to \cdots \to \mathsf{real}\ \delta_n$ with the normal form

$$\mathsf{real}\ d_1^{x_{1,1}} \cdots d_m^{x_{m,1}} \to \cdots \to \mathsf{real}\ d_1^{x_{1,n}} \cdots d_m^{x_{m,n}}.$$

This can be written as the following integer matrix:

$$\begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{pmatrix}.$$

If the closed type scheme $\forall d_1 \ldots \forall d_m.\tau$ is simplified using $\mathrm{Simplify}(\emptyset, \tau)$ then the matrix representing the resulting type (after renaming of variables) has the following 'stairstep' form:

$$\begin{pmatrix} 0\ldots0 & a_1 & ?\ldots? & b_1 & ?\ldots? & c_1 & ?\ldots? & \cdots \\ 0\ldots0 & 0 & 0\ldots0 & b_2 & ?\ldots? & c_2 & ?\ldots? & \cdots \\ 0\ldots0 & 0 & 0\ldots0 & 0 & 0\ldots0 & c_3 & ?\ldots? & \cdots \\ 0\ldots0 & 0 & 0\ldots0 & 0 & 0\ldots0 & 0 & 0\ldots0 & \ddots \end{pmatrix}$$

It is easy to see from the definition of Simplify and from Lemma 3.7 that all corner entries (a_1, b_2, *etc.*) are positive and that entries lying above them (b_1, c_1, *etc.*) are non-negative and smaller than the corresponding corner entry. The symbol $?\ldots?$ denotes a block of arbitrary entries, and $0\ldots0$ denotes a block of zeros. In the language of matrix theory this is known as a *reduced row echelon* matrix or *Hermite* matrix. It is unique [1], that is, for every non-zero $m \times n$ matrix A there is a unique $m \times n$ Hermite matrix UA for some invertible $n \times n$ matrix U. In the language of types, this means that $\mathrm{SimplifyScheme}(\sigma)$ does find a canonical representative for the $\cong_D$-equivalence class containing σ, as we hoped.

Calculating Gen

The calculation of $\mathrm{Gen}(\Gamma, \tau)$ specified by Definition 2.5 left one question unanswered, namely how to obtain an invertible substitution U to bring Γ into free variable reduced form. The algorithm CofB shown in Figure 3.7 calculates such a 'change of basis'. It takes two arguments: a set of dimension variables $\mathcal{V}$ and a type assignment Γ. It works by applying Simplify to the monomorphic (or 'free') part of every type scheme in Γ, taking care not to undo any changes made already (including those on variables in $\mathcal{V}$). The invertible substitution U which

$$
\begin{array}{rcl}
\mathrm{CofB}(\mathcal{V}, \emptyset) & = & I \\
\mathrm{CofB}(\mathcal{V}, \{x:\sigma\} \cup \Gamma) & = & U_2 \circ U_1 \\
& & \text{where } \sigma = \forall \vec{d}.\tau \\
& & \text{and } U_1 = \mathrm{Simplify}(\mathcal{V} \cup \vec{d}, \tau) \\
& & \text{and } U_2 = \mathrm{CofB}(\mathcal{V} \cup \mathrm{fdv}(U_1(\sigma)), U_1(\Gamma))
\end{array}
$$

Figure 3.7: Algorithm CofB

it returns then ensures that $U(\Gamma)$ is in free variable reduced form *away from* $\mathcal{V}$. This is formalised by a lemma which follows the same lines as Lemma 3.5.

Lemma 3.11. *For any type assignment* Γ *and set of dimension variables* $\mathcal{V}$, $\mathrm{CofB}(\mathcal{V}, \Gamma)$ *terminates with an invertible substitution* U *such that* $\mathrm{dom}(U) \cap \mathcal{V} = \emptyset$. *Furthermore, for any two dimension substitutions* S_1 *and* S_2 *with* $\mathrm{dom}(S_1) \cap \mathcal{V} = \mathrm{dom}(S_2) \cap \mathcal{V} = \emptyset$,

$$S_1(U(\Gamma)) \cong_D S_2(U(\Gamma)) \text{ implies } S_1(d) =_D S_2(d) \text{ for all } d \in \mathrm{fdv}(U(\Gamma)).$$

Proof. By induction on the size of Γ. □

When $\mathcal{V} = \emptyset$ we obtain Proposition 2.4 as a trivial corollary.

Corollary (Change of Basis). *Let* $U = \mathrm{CofB}(\emptyset, \Gamma)$. *Then* U *is an invertible substitution such that* $\mathrm{fdv}(U(\Gamma))$ *is in free variable reduced form.*

To sum up, the calculation of $\sigma = \mathrm{Gen}(\Gamma, \tau)$ proceeds as follows:

1. Determine $U = \mathrm{CofB}(\emptyset, \Gamma)$.
2. Calculate $\sigma' = U^{-1}(\forall \vec{v}.U(\tau))$, where $\vec{v} = \mathrm{fv}(U(\tau)) \setminus \mathrm{fv}(U(\Gamma))$.

3.4 Related work

There have been two other proposals for dimension inference in the style of ML devised independently from the system described in this dissertation. They are outlined here along with some comments concerning other type systems based on equational theories.

Wand and O'Keefe's system

Wand and O'Keefe define an ML-like type system extended with a single numeric type parameterised on dimension [68]. 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 $\mathsf{real}\,\delta$ type of this chapter 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 there are three base dimensions M, L and T, then the expression $Q(n_1, n_2, n_3)$ represents a quantity with the dimensions $\mathrm{M}^{n_1} \cdot \mathrm{L}^{n_2} \cdot \mathrm{T}^{n_3}$.

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

$$\forall d. \mathsf{real}\ d^2 \rightarrow \mathsf{real}\ d$$

   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 real M.

Their inference algorithm, like Infer, must solve equations between dimensions. But 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. The arguments for and against fractional exponents have already been mentioned in Chapter 1.

Finally, Wand and O'Keefe propose a construct `newdim` which introduces a *local* dimension. This is equivalent to the local use of the `dimension` declaration in the ML Kit implementation described in the next chapter.

Goubault's system

Jean Goubault has also looked at the problem of adding dimension types to ML [18]. His extension is closer to the system described here in that dimension types are formed from any combination of base dimensions which have been predeclared in the program. Goubault chooses to make dimension variables a special kind of type variable analogous to the equality type variables of Standard ML. Syntactic restrictions then ensure that it is not possible to write types such as $\tau_1 \cdot \tau_2$, the 'dimension product' of two ordinary types, or $\delta_1 \rightarrow \delta_2$, a 'function' from dimensions to dimensions. Like Wand and O'Keefe, Goubault allows exponents which are not integers. Dimensions then form a vector space over the rationals. A unification algorithm for this equational theory is presented.

As with our ML Kit implementation described in the next chapter, a base dimension declaration must be accompanied by a default unit for that dimension. Furthermore, the existing type synonym facility is used to provide names for derived dimensions, for example, using Force to stand for $\mathrm{M} \cdot \mathrm{L} \cdot \mathrm{T}^{-2}$.

Goubault also suggests an intriguing extension to the module system of Standard ML which would allow different modules to use different units of measure, the compiler inserting conversions between them automatically. This is discussed in more detail in the final chapter of this thesis.

Related type systems

The work on polymorphic type inference began with Milner's algorithm $\mathcal{W}$ [42], the discovery that it inferred principal types [13], and the detailed proofs in Damas's thesis [12]. Since then, hundreds of papers on type systems and inference algorithms have been published. I would like to acknowledge two theses which I have found useful: those by Tofte [64] and by Mark Jones [26].

There are few other type inference algorithms which apply equational unification. Thatte's application to type coercions is one such [62]; another is Rémy's type inference with extensible records [54]. Order-sorted unification has been applied to the problem of type inference for Haskell's type classes [47].

Rémy has also studied 'equational' type inference more generally [55]. However, he restricts his presentation to *regular* equational theories. To the author's knowledge, dimension inference is the first application of a non-regular equational theory to type inference, and leads to the subtleties of generalisation discussed at length in Chapter 2.

Rémy makes the observation that even in the standard ML type system, the generalisation step used in the (let) rule is computationally expensive: it involves calculating the free type variables in a type assignment which can be arbitrarily large. He goes on describe an equivalent system which avoids recalculation of these free variables by assigning a *rank* to type variables as a measure of their 'freshness'. In the inference algorithm for this system ranks are calculated incrementally by extending unification to 'ranked' unification. Rémy's idea can be applied only to type systems based on a regular equational theory. For systems based on non-regular theories such as the one discussed in this dissertation the generalisation of a type with respect to a type assignment is even more expensive. For our system it involves first calculating a change of basis which brings the type assignment into a form in which its free variables are independent with respect to substitution. It might be possible to adapt Rémy's idea of ranking to such a system in order to improve the efficiency of type inference and perhaps simplify the presentation of the system. This is one direction for future research.

3.5 Some refinements

In this final section we discuss some possible refinements to the type system of ML_δ which would allow more expressions to be typed.

Polymorphic recursion

In the ML type system, and our extension ML_δ, only the let construct can introduce polymorphism. In particular, lambda-bound variables can not be used at more than one type, and the rule for letrec ensures that recursive functions are not used polymorphically within their own definition.

Consider the following program written in ML:

```
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. However, in the ML_δ type system a version without the exchange is assigned a type scheme

$$\forall d_1.\forall d_2.\mathsf{real}\ d_1\ \mathsf{list} \times \mathsf{real}\ d_2\ \mathsf{list} \to (\mathsf{real}\ d_1 \cdot d_2)\ \mathsf{list}$$

whereas the version above would be given the less general type

$$\forall d.\mathsf{real}\ d\ \mathsf{list} \times \mathsf{real}\ d\ \mathsf{list} \to \mathsf{real}\ d^2\ \mathsf{list}.$$

More realistic examples involve the use of user-defined datatypes. Consider creating a list of successive derivatives of a function f, that is, $[f, f', f'', \ldots]$. If the function f has type $\mathsf{real}\ d_1 \to \mathsf{real}\ d_2$ then its derivative f' has type $\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \cdot d_1^{-1}$, its second derivative f'' has type $\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \cdot d_1^{-2}$, and so on. The type of this list can be expressed by the following ML-style datatype definition:

$$\begin{array}{l} \mathsf{datatype}\ \mathsf{difflist}(d_1, d_2) = \\ \quad \mathsf{Nil} \\ \ \mid\ \mathsf{Cons\ of}\ (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2) \times \mathsf{difflist}(d_1, d_2 \cdot d_1^{-1}). \end{array}$$

Then for a particular function this list can be created by the following ML program which makes use of the function `diff` from page 10 in Chapter 1:

```
fun diffs(h, f, 0) = Nil
  | diffs(h, f, n) = Cons(f, diffs(h, diff(h, f), n-1))
```

If (dimension-)polymorphic recursion was allowed, then this function would be assigned the type

$$\forall d_1.\forall d_2.\mathsf{real}\ d_1 \times (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2) \times \mathsf{int} \to \mathsf{difflist}(d_1, d_2).$$

More examples can be found in Rittri's article [58].

The ordinary ML typing rules can be extended to allow polymorphism inside recursive definitions. The rule shown below, first proposed by Mycroft [45], is for a call-by-name language with a fixed-point construct $\mathsf{rec}\ x.e$. Our (letrec) rule could be modified similarly.

$$(\text{polyrec}) \quad \frac{\Gamma[x : \sigma] \vdash e : \sigma}{\Gamma \vdash \mathsf{rec}\ x.e : \sigma}$$

Unfortunately, it has been shown by Henglein [23] and by Kfoury *et al.* [31] that type inference is undecidable in the presence of this rule.

For our language ML_δ the rule could be weakened so that it permits polymorphism over dimensions but not over types. This is then expressive enough to type the function `prodlists` from above. Then the question arises: is type inference still decidable? Some progress on this has been made by Rittri. The proof of undecidability of inference under polymorphic recursion for *types* rests

on a reduction from the problem of *semi-unification* to the problem of type inference. The semi-unification problem is the following. We are given a set $\mathcal{I}$ of pairs of types which represent *inequations*:

$$\mathcal{I} = \{(\tau_1, \tau_1'), \ldots, (\tau_n, \tau_n')\}.$$

Now find a substitution S such that $S(\tau_i) \leqslant S(\tau_i')$ for $1 \leqslant i \leqslant n$. The inequality $\tau_i \leqslant \tau_i'$ holds whenever there is some substitution R_i such that $R_i(\tau_i) = \tau_i'$. The substitution S is said to be a *semi-unifier* of the inequations in $\mathcal{I}$. Notice that the same substitution S is applied to each inequation, whereas the matching substitutions R_i may differ.

The analogous notion for a language extended with dimensions would be semi-unification in the equational theory of Abelian groups. Then an inequation $\tau \leqslant_D \tau'$ holds if there is a substitution R such that $R(\tau) =_D \tau'$. An Abelian group semi-unifier of the inequations represented by

$$\mathcal{I} = \{(\tau_1, \tau_1'), \ldots, (\tau_n, \tau_n')\}$$

is a substitution S such that there exists a matching substitution R_i for each inequation giving $R_i(S(\tau_i)) =_D S(\tau_i')$ for all i. Rittri has shown that for a single inequation this problem is decidable, and he presents an algorithm which solves it [57]. However, in order to use semi-unification to perform type inference in the presence of polymorphic recursion, it might be necessary to solve several inequations. For unification, the problem of solving several equations can be reduced to that of solving a single equation by simple tupling. For semi-unification, this is not the case, and the decidability of semi-unification over Abelian groups in this general case is still open.

Rittri has also investigated dimension type inference for a language in which dimensions have rational exponents. For such a language, the appropriate equational theory is that of vector spaces over the rationals. Rittri gives an algorithm for semi-unification in this theory and presents a type inference system modelled on Henglein's [23] which determines a most general type even in the presence of polymorphic recursion [58]. This time, his algorithm works for the general case of several inequations.

There is a further question: can we solve the type inference problem by some means other than semi-unification? For ordinary ML extended with polymorphic recursion, Henglein and Kfoury *et al.* have given reductions both ways, showing that the type inference problem is *equivalent* to the semi-unification problem. For dimension types, so far it has only been shown that inference can be reduced to equational semi-unification; it is not yet known whether a reduction can be made in the other direction.

So the situation can be summed up as follows:

- Type inference for ML in the presence of polymorphic recursion is equivalent to the semi-unification problem. That problem is undecidable.

- Type inference for ML in the presence of dimension-polymorphic recursion where dimensions have rational exponents can be reduced to semi-

unification over vector spaces. That problem is decidable and admits a straightforward algorithm.

- Type inference for ML in the presence of dimension-polymorphic recursion where dimensions have integral exponents can be reduced to semi-unification over Abelian groups. There is an algorithm which solves a single inequation in this theory; the situation for the general case is not known.
- Semi-unification is *sufficient* to solve the problem of type inference with dimension-polymorphic recursion; it is not known whether it is *necessary.*

Dependent types

Consider a function which takes an integer argument n, a real-valued argument x, and returns x^n, the result of raising x to the power n. Here is an implementation written in Standard ML:

```
fun power 0 x = 1.0
  | power n x = if n < 0 then 1.0 / power (~n) x
                else x * power (n-1) x
```

If x had type $\mathsf{real}\,\delta$ for some dimension δ, we would like the result to have type $\mathsf{real}\,\delta^n$. However, this type depends on the *value* of the argument n, so the only way of expressing the intuitive type of this function is by a *dependent type* such as

$$\forall d.\forall n \in \mathbb{Z}.\mathsf{real}\, d \to \mathsf{real}\, d^n.$$

The inability to write such functions in a language with a type system like ML_δ is not a problem in practice, as almost all exponentiation operations in numerical algorithms are used with constant integer powers. If necessary, a real language with dimension types could provide a built-in shorthand for this operation and assign the result a dimension accordingly.

Higher-order polymorphism

Consider the following function written in ML:

```
fun polyadd prod = prod 2.0 kg + prod kg 3.0
```

Assuming that `kg` has the type $\mathsf{real}\,\mathrm{M}$, the function `prod` is used at more than one dimension type. This is sometimes called *polymorphic abstraction.* The type of `polyadd` can only be expressed by allowing a type *scheme* in the the argument position in a function type, as follows:

$$(\forall d_1.\forall d_2.\mathsf{real}\, d_1 \to \mathsf{real}\, d_2 \to \mathsf{real}\, d_1 \cdot d_2) \to \mathsf{real}\,\mathrm{M}.$$

In such a system, top-level types have the form

$$\forall \vec{v}.(\sigma_1 \to \ldots \to \sigma_n \to \tau)$$

where each σ_i is a type scheme and τ is a simple type, defined as before. The problem of type inference for ML in the presence of polymorphic abstraction is undecidable and equivalent to the problem of semi-unification [31]. For polymorphism over dimensions alone, the problem is open.

Now consider a function `makecube` which accepts a polymorphic 'product' operation of type $\forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2$ and returns a polymorphic 'cube' operation of type $\forall d.\mathsf{real}\ d \to \mathsf{real}\ d^3$. This time both argument and result are polymorphic.

In Chapters 5 and 6 we study an *explicitly-typed* language, in the style of System F, in which dimension quantifiers can appear at any depth in a type. It is powerful enough to express both `polyadd` and `makecube`. In Chapter 8 we sketch an even more expressive dimension type system which supports quantification over dimension *operators* (functions from dimensions to dimensions), formalised by a system of *kinds*.

Chapter 4

Implementation

After implementing a dimension type system and inference algorithm for a toy language resembling ML_δ, the decision was taken to test the practical viability of dimension types by implementing an extension to Standard ML.

The ML Kit [5] was chosen for this purpose. It is a full implementation of Standard ML designed with extensibility as its primary objective rather than efficiency. It can be seen as a direct implementation of the rules in the Definition of Standard ML [43], treating the rules for the static semantics as a prototypical type inference algorithm and the rules for the dynamic semantics as a prototypical interpreter. To be completely precise, we should describe the changes to these rules which are necessary to support dimension types. This would involve only modest changes to the existing rules and a small number of additional rules. We do not do this, as there is plenty of formalism elsewhere in this thesis; moreover, it is used there as the precursor to *theorems* about a language, and proving such theorems for an extension to the complete Definition of Standard ML would be a major undertaking. Also, the Definition is large and complex with its own jargon and notation. Therefore instead we describe informally the issues involved in extending Standard ML to provide dimension types without assuming detailed knowledge of the Definition.

Apart from the problem of overloading discussed later, the extension is *conservative*, in the sense that existing Standard ML programs would run unchanged but with more refined types inferred for numerical programs. Type checking only catches dimension errors when base dimensions have been declared and used in a program.

4.1 Syntax

Figure 4.1 lists the changes to the grammar of the Core language and Modules language of Standard ML, in the style of the Definition [43, Sections 2 and 3]. The meta-syntax *xseq* stands for an empty sequence, a singleton x, or a sequence `(`x_1`,...,`x_n`)`. Notice how the dimension type extensions have the same flavour as other constructs in Standard ML. For example, the syntax for declaring dimensions follows the same pattern as for type and datatype declarations, so

Extensions to the core:

dec	::=	...	other declarations
		`dimension` *dimbind*	dimenson declaration
dimbind	::=	*dimcon* `(` *var* `)` ⟨`and` *dimbind*⟩	dimension binding
typbind	::=	*tdvarseq tycon* `=` *ty* ⟨`and` *typbind*⟩	type binding
datbind	::=	*tdvarseq tycon* `=` *conbind* ⟨`and` *datbind*⟩	datatype binding
ty	::=	...	
		tdseq longtycon	type construction
		...	
td	::=	*ty*	type
		`[` ⟨*dim*⟩ `]`	dimension
dim	::=	*dimvar*	dimension variable
		longdimcon	base dimension
		dim `^` *int*	exponentiation
		dim_1 `*` dim_2	dimension product
		dim_1 `/` dim_2	dimension quotient
		`(` *dim* `)`	grouping
tdvar	::=	*tyvar*	type variable
		`[` *dimvar* `]`	dimension variable

Extensions to the modules:

spec	::=	...	other specifications
		`dimension` *dimdesc*	dimension spec.
dimdesc	::=	*dimcon* ⟨`(` *var* `)`⟩ ⟨`and` *dimdesc*⟩	dimension description
typdesc	::=	*tdvarseq tycon* ⟨`and` *typdesc*⟩	type description
datdesc	::=	*tdvarseq tycon* `=` *condesc* ⟨`and` *datdesc*⟩	datatype description

Figure 4.1: Grammar

dimension parameters precede constructors in constrast to the syntax used for the toy language ML_δ. Also, base dimension names are *generative* in the same way that datatype names are generative in Standard ML. All this is described in detail below.

4.2 A new type

In ML_δ, the type of real numbers was parameterised on a dimension. For the extension to Standard ML it was decided to use the language's existing parameterisation mechanism and extend it to allow any type or datatype to be parameterised on a dimension. This required that the syntax distinguishes between dimensions and types. Square brackets were chosen for this purpose, in line with the conventional scientific notation for dimensions. The new numeric type is written `[`*dim*`] dim`, where the dimension parameter *dim* precedes the type constructor `dim`, as is the convention in Standard ML. The type of dimensionless reals is written `[] dim`, but to ensure compatibility with the original language, `real` is provided as a convenient type synonym:

```
type real = [] dim
```

Then the syntax of dimension expressions is the following:

- Base dimensions (*dimcon* in the grammar) are just alphabetic identifiers such as `M` or `Temp`. These must be predeclared in a `dimension` construct as described in Section 4.4.
- Dimension variables (*dimvar* in the grammar) are written in the same way as type variables, with a preceding apostrophe. The presence of the square brackets around dimensions ensures that there can be no confusion between type variables and dimension variables.
- The product of dimensions dim_1 and dim_2 is denoted by dim_1`*`dim_2.
- The exponentiation of dimension *dim* to the power *n* is denoted by *dim*`^`*n*.
- The notation dim_1`/`dim_2 is syntactic sugar for dim_1`*`dim_2`^~1`.

A suitable type synonym defining density would be the following:

```
type Density = [M / L^3] dim
```

In our language ML_δ this would have been written real $(\mathrm{M} \cdot \mathrm{L}^{-3})$. Here is a function which expresses Newton's second law:

```
fun force (mass : [M] dim, acc : [L / T^2] dim) = mass * acc
```

Finally, here is a favourite toy example of object-oriented programmers, reinterpreted to illustrate the utility of dimension types:

```
type ColouredPoint =
{
  x     : [L] dim,
  y     : [L] dim,
  red   : [Intensity] dim,
  green : [Intensity] dim,
  blue  : [Intensity] dim
}
```

Every type constructor in Standard ML has an *arity*, which is the number of type parameters it expects. For our extension, type constructors take a mixture of type and dimension parameters. This shows up in the grammar as *td* (a type or dimension parameter) and *tdseq* (a sequence of type or dimension parameters). Hence the arity is no longer just a number—internally, the implementation uses a `bool list`. The grammar for `type`, `datatype` and `abstype` declarations is changed accordingly, with square brackets used again, this time to distinguish between type and dimension variables (*tdvar* and *tdvarseq* in the grammar). Here is an example which illustrates this—a datatype which represents a 3-vector whose components all have the same dimension:

```
datatype ['a] Vector = Vec of ['a] dim * ['a] dim * ['a] dim
```

A second example uses both type and dimension parameters:

```
datatype ('a, ['b]) Pos = Map of ('a * ['b] Vector) list
```

4.3 New operations

Figure 4.2 lists the new polymorphic types assigned to arithmetic functions. All other operations have a dimensionless type, unchanged from Standard ML, assuming the presence of the type synonym described in the previous section. For example, the trigonometric function `sin` has the type

```
val sin : real -> real
```

There is one major problem with this scheme: *overloading*. Standard ML provides arithmetic operations and comparison functions at more than one type, but does so in an ad-hoc way which is not available to the programmer. Functions such as addition and subtraction are tentatively assigned the type

```
num * num -> num
```

where `num` is a placeholder which is replaced by `real` or `int` during type inference. The program context is used to determine how to *resolve* such types, and if this is not possible then the type checker generates an error. To extend this scheme

```
val *     : ['a] dim * ['b] dim -> ['a * 'b] dim
val /     : ['a] dim * ['b] dim -> ['a / 'b] dim
val +     : ['a] dim * ['a] dim -> ['a] dim
val -     : ['a] dim * ['a] dim -> ['a] dim
val ~     : ['a] dim -> ['a] dim
val abs   : ['a] dim -> ['a] dim
val sqrt  : ['a^2] dim -> ['a] dim
val <     : ['a] dim * ['a] dim -> bool
val >     : ['a] dim * ['a] dim -> bool
val <=    : ['a] dim * ['a] dim -> bool
val >=    : ['a] dim * ['a] dim -> bool
```

Figure 4.2: Part of the initial static basis

to dimension types we cannot just replace a placeholder by `int` or `dim` because of the dimension parameter to `dim`, which in the case of multiplication is not even the same for both arguments. Perhaps the tentative type scheme for `*` should be

```
['a] num * ['b] num -> ['a*'b] num
```

and then `['a] num` is replaced by `['a] dim` or `int` according to the context. All this has yet to be worked out, but a generalisation of this idea is sketched in Chapter 8 as an area of future research.

4.4 Dimension declarations

In ML_δ it was assumed that base dimensions were predefined. For an extension to Standard ML it was necessary to devise some kind of declaration mechanism analogous to the existing `type`, `datatype` and `exception` constructs. The new construct is best illustrated by an example:

```
dimension M(kg) and L(m) and T(s)
```

This declaration introduces three base dimensions (`M` for mass, `L` for length and `T` for time) and their associated SI units. The aim is that these units are just values with types `[M] dim`, `[L] dim` and `[T] dim` respectively. They can then be used to form constants of any chosen dimension. For example, to bind an identifier `force` to the value $2.3\,\mathrm{N}$ ($\equiv \mathrm{kg\,m\,s^{-2}}$) we could write

```
val force = 2.3 * kg * m / (s*s)
```

which would assign the type shown below:

```
val force : [M * L * T ^ ~2] dim
```

Note that this should be the *only* means of constructing basic values. In particular, non-zero real constants should not be polymorphic, as was discussed on page 8 in the Introduction.

As with other binding constructs in Standard ML, dimension declarations are *generative*, meaning that each binding generates a new name to stand for the base dimension identifier. For example, consider the following program:

```
dimension M(kg)
val x = 3.0*kg
fun f (m : [M] dim) = m*5.0
dimension M(lb)
val y = 4.0*kg
fun g (m : [M] dim) = m*6.0
```

The second declaration of `M` hides the first, so that all of the bindings shown below would be ill-typed if added to the end of the above program.

```
val z1 = g x
val z2 = f y
val z3 = kg + lb
```

Dimension declarations can be localised to other declarations (with `local`) or expressions (with `let`). Here is an example:

```
fun f x =
let dimension Apples(apple) and Oranges(orange)
in
  ...
end
```

Wand and O'Keefe suggest a similar construct in their paper on dimensional inference [68].

4.5 Modules

Dimension *specifications* can appear in signatures. Then any structure matching such a signature must provide a dimension declaration matching the specification. By analogy with datatypes, in which a signature may choose to hide the value constructors for the datatype or make them visible, a dimension specification can hide the default unit associated with the dimension or make it visible.

These extensions to the module system of Standard ML are minimal. A more flexible extension would include a dimension *sharing* mechanism analogous to the type sharing which already exists. The dimension sharing construct might even express unit conversions between different default units used for the respective dimensions. This is an idea of Goubault [18] and is discussed further in Chapter 8.

Chapter 5

Operational semantics

In this chapter we study the operational semantics of an explicitly-typed language with dimension types which we call Λ_δ. In Section 5.1 we justify this choice of language, which is strictly more expressive than the language ML_δ studied for the purposes of type inference. Sections 5.2 and 5.3 define the syntax, typing rules and operational semantics of Λ_δ. Then in Section 5.4 we prove a fundamental property of the language, that "well-dimensioned programs do not go wrong"—in other words, dimension errors cannot occur at run-time. Finally in Section 5.5 we specify the semantics of the dimension types fragment of ML_δ by translating its typing derivations into terms in Λ_δ.

5.1 An explicitly-typed language

It is possible to define the semantics of the language ML_δ described in Chapter 3 directly, either by induction on the typing derivations of expressions, or by first translating the language into an explicitly-typed variant with the same expressive power. This approach is taken by Harper and Mitchell in their study of ML [20], where an implicitly-typed source language *Core*-ML is translated into an explicitly-typed target language *Core*-XML. A similar idea is pursued by Benton in his thesis [4]; in this case, the target language is very much in the spirit of ML but every variable is tagged with its type and the use of polymorphism in let constructs is made explicit by binding distinct variables to distinct type instances of the polymorphic expression. It is possible to do the same for ML_δ, as in the following example whose typing derivation is shown in Figure A.2 in Appendix A.

$$\lambda y.\ \mathsf{let}\ sqr = \lambda x.\ x * x\ \ \mathsf{in}\ \ sqr(3.14) * sqr(y)$$

This would be written

$$\begin{aligned}\lambda y^{\mathsf{real}\ d}.\ \ & \mathsf{let}\ sqr^{\mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}}, sqr^{\mathsf{real}\ d \to \mathsf{real}\ d^2} = \lambda x^{\mathsf{real}\ d'}.\ x^{\mathsf{real}\ d'} * x^{\mathsf{real}\ d'} \\ & \mathsf{in}\ sqr^{\mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}}(3.14) * sqr^{\mathsf{real}\ d \to \mathsf{real}\ d^2}(y^{\mathsf{real}\ d})\end{aligned}$$

We choose instead to study an explicitly-typed language in which dimensions are passed as arguments in the same way that types are passed as arguments

$$
\begin{array}{rcll}
\delta & ::= & d & \textit{dimension variables} \\
& | & \mathbf{1} & \textit{unit dimension} \\
& | & \delta_1 \cdot \delta_2 & \textit{dimension product} \\
& | & \delta^{-1} & \textit{dimension inverse} \\
\\
\tau & ::= & \mathsf{bool} & \textit{booleans} \\
& | & \mathsf{real}\ \delta & \textit{dimensioned reals} \\
& | & \tau_1 \to \tau_2 & \textit{function types} \\
& | & \forall d.\tau & \textit{dimension quantification}
\end{array}
$$

Figure 5.1: Dimensions and types

in Reynolds' polymorphic lambda calculus or Girard's System F [56, 17]. We call this second-order language Λ_δ. One advantage of this approach is that we do not need to consider the semantics of a let construct explicitly. Instead, let constructs can be translated into the dimension abstraction ($\Lambda d.e$) and value abstraction ($\lambda x\colon \tau.\ e$) constructs of Λ_δ. For example, the function above would be rendered as the following expression whose derivation is shown in Figure A.3 in Appendix A.

$$
\lambda y\colon \mathsf{real}\ d.\ (\lambda sqr\colon (\forall d.\mathsf{real}\ d \to \mathsf{real}\ d^2).\ sqr\ \mathbf{1}\ (3.14) * sqr\ d\ (y)) \\
(\Lambda d.\lambda x\colon \mathsf{real}\ d.\ x * x)
$$

It is straightforward to express a semantics for this language and prove results using the semantics. In the final chapter of this thesis we show how the language can be generalised in a natural way to give an even more richly-typed language in the style of F_ω. In fact, the language Λ_δ is already more expressive than ML_δ. The problematic `makecube` function of Chapter 3 can be written:

$$
\begin{array}{l}
\lambda f\colon \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2. \\
\quad \Lambda d.\lambda x\colon \mathsf{real}\ d. \\
\quad\quad f\ d\ d^2\ x\ (f\ d\ d\ x\ x)
\end{array}
$$

This function accepts a polymorphic product function f and returns a polymorphic cube function as result.

The trend in programming language design seems to be towards defining a very expressive, explicitly-typed core language and then treating type reconstruction as a 'front-end' issue akin to parsing. Recent work by Odersky and Läufer [49] shows how a mixture of explicit type annotations and Milner-style type inference can achieve the expressiveness of System F in a practical programming language. Therefore we consider a more richly-typed language such as Λ_δ worthy of study in its own right.

$$
\begin{array}{rcll}
e & ::= & x & \textit{variable } (x \in \textit{Vars}) \\
 & | & r & \textit{constant } (r \in \mathbb{Q}) \\
 & | & 0_\delta & \textit{polymorphic zero} \\
 & | & e_1\, e_2 & \textit{application} \\
 & | & \lambda x : \tau .\, e & \textit{typed abstraction} \\
 & | & \mathsf{rec}\, y(x : \tau_1) : \tau_2 .\, e & \textit{recursion} \\
 & | & e\, \delta & \textit{dimension application} \\
 & | & \Lambda d.e & \textit{dimension abstraction} \\
 & | & e_1 + e_2 & \textit{addition} \\
 & | & e_1 - e_2 & \textit{subtraction} \\
 & | & e_1 * e_2 & \textit{multiplication} \\
 & | & e_1 / e_2 & \textit{division} \\
 & | & e_1 < e_2 & \textit{comparison} \\
 & | & \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 & \textit{conditional}
\end{array}
$$

Figure 5.2: Expressions

5.2 Syntax

The syntax for dimensions and types in Λ_δ is defined by the grammar shown in Figure 5.1. Dimensions are the same as in ML_δ, except that base dimensions are not distinguished from dimension variables. The equivalence relation $=_D$ is identical and lifts to types in the obvious way, again identifying polymorphic types up to renaming of bound variables.

Simple types and type schemes now are not separate syntactic classes. This permits polymorphic types with nested quantifiers such as

$$(\forall d_1.\forall d_2.\mathsf{real}\; d_1 \to \mathsf{real}\; d_2 \to \mathsf{real}\; d_1 \cdot d_2) \to (\forall d.\mathsf{real}\; d \to \mathsf{real}\; d^3)$$

which is the type of the function `makecube` from the previous page. Also notice that we are allowing quantification only over dimensions. Conventional type polymorphism is orthogonal to our study of dimension types so to simplify the theory we do not consider it here.

Figure 5.2 gives the grammar for expressions in the language, and Figure 5.4 presents a set of typing rules. As in previous chapters, Γ is a *type assignment*: a finite mapping between identifiers and types. This time we take more care with free and bound dimension variables, so the context in a judgment $\mathcal{V}; \Gamma \vdash e : \tau$ contains a set of dimension variables $\mathcal{V}$ in addition to the type assignment Γ.

We define a *well-formed* judgment to be one in which the free dimension variables of the type assignment Γ, expression e and type τ are contained in $\mathcal{V}$. Then the typing rules in Figure 5.4 are assumed to apply only to well-formed judgments. An alternative to defining away ill-formed judgments in this way would be to set up some inductive rules expressing the well-formedness of dimensions, types and type assignments.

$$
\begin{aligned}
\mathrm{fdv}(\mathsf{bool}) &= \emptyset \\
\mathrm{fdv}(\mathsf{real}\ \delta) &= \mathrm{fdv}(\delta) \\
\mathrm{fdv}(\tau_1 \rightarrow \tau_2) &= \mathrm{fdv}(\tau_1) \cup \mathrm{fdv}(\tau_2) \\
\mathrm{fdv}(\forall d.\tau) &= \mathrm{fdv}(\tau) \setminus \{d\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{fdv}(x) &= \emptyset \\
\mathrm{fdv}(r) &= \emptyset \\
\mathrm{fdv}(0_\delta) &= \mathrm{fdv}(\delta) \\
\mathrm{fdv}(e_1\ e_2) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \\
\mathrm{fdv}(\lambda x\colon \tau.\ e) &= \mathrm{fdv}(\tau) \cup \mathrm{fdv}(e) \\
\mathrm{fdv}(\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\ e) &= \mathrm{fdv}(\tau_1) \cup \mathrm{fdv}(\tau_2) \cup \mathrm{fdv}(e) \\
\mathrm{fdv}(e\ \delta) &= \mathrm{fdv}(e) \cup \mathrm{fdv}(\delta) \\
\mathrm{fdv}(\Lambda d.e) &= \mathrm{fdv}(e) \setminus \{d\} \\
\mathrm{fdv}(e_1 + e_2) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \\
\mathrm{fdv}(e_1 - e_2) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \\
\mathrm{fdv}(e_1 * e_2) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \\
\mathrm{fdv}(e_1 / e_2) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \\
\mathrm{fdv}(e_1 < e_2) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \\
\mathrm{fdv}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) &= \mathrm{fdv}(e_1) \cup \mathrm{fdv}(e_2) \cup \mathrm{fdv}(e_3)
\end{aligned}
$$

Figure 5.3: Free dimension variables in types and expressions

In summary, then, a typing judgment

$$\mathcal{V};\Gamma \vdash e : \tau$$

means

> "In the context of a typing assignment Γ and set of dimension variables $\mathcal{V}$, the expression e has type τ with $\mathrm{fdv}(\Gamma) \cup \mathrm{fdv}(e) \cup \mathrm{fdv}(\tau) \subseteq \mathcal{V}$."

Free dimension variables for dimensions are defined as in Chapter 2, and Figure 5.3 gives the definition for types and for expressions.

We will often write $\mathcal{V};\Gamma \vdash e : \tau$ to mean that there is a well-formed typing derivation in Λ_δ with conclusion $\mathcal{V};\Gamma \vdash e : \tau$. If it is necessary to make clear that the derivation is in Λ_δ then we will write $\mathcal{V};\Gamma \vdash_\delta e : \tau$.

Now consider each rule in turn.

Rules (const) and (zero). As with ML_δ dimensionless constants are drawn from the rationals; it would be perverse to allow irrational constants such

$$(\text{var})\ \frac{}{\mathcal{V};\Gamma \vdash x : \Gamma(x)} \qquad (\text{const})\ \frac{}{\mathcal{V};\Gamma \vdash r : \mathsf{real}\ \mathbf{1}}\ r \neq 0 \qquad (\text{zero})\ \frac{}{\mathcal{V};\Gamma \vdash 0_\delta : \mathsf{real}\ \delta}$$

$$(\text{app})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \mathcal{V};\Gamma \vdash e_2 : \tau_1}{\mathcal{V};\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\text{abs})\ \frac{\mathcal{V};\Gamma[x : \tau_1] \vdash e : \tau_2}{\mathcal{V};\Gamma \vdash (\lambda x\colon \tau_1.\ e) : \tau_1 \to \tau_2}$$

$$(\text{rec})\ \frac{\mathcal{V};\Gamma[x : \tau_1, y : \tau_1 \to \tau_2] \vdash e : \tau_2}{\mathcal{V};\Gamma \vdash (\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\ e) : \tau_1 \to \tau_2}$$

$$(\text{dgen})\ \frac{\mathcal{V} \cup \{d\};\Gamma \vdash e : \tau}{\mathcal{V};\Gamma \vdash \Lambda d.e : \forall d.\tau}\ d \text{ not free in } \Gamma \qquad (\text{dspec})\ \frac{\mathcal{V};\Gamma \vdash e : \forall d.\tau}{\mathcal{V};\Gamma \vdash e\ \delta : \{d \mapsto \delta\}\tau}$$

$$(\text{add})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\ \delta \quad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\ \delta}{\mathcal{V};\Gamma \vdash e_1 + e_2 : \mathsf{real}\ \delta}$$

$$(\text{sub})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\ \delta \quad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\ \delta}{\mathcal{V};\Gamma \vdash e_1 - e_2 : \mathsf{real}\ \delta}$$

$$(\text{mul})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\ \delta_1 \quad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\ \delta_2}{\mathcal{V};\Gamma \vdash e_1 * e_2 : \mathsf{real}\ \delta_1 \cdot \delta_2}$$

$$(\text{div})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\ \delta_1 \quad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\ \delta_2}{\mathcal{V};\Gamma \vdash e_1 / e_2 : \mathsf{real}\ \delta_1 \cdot \delta_2^{-1}}$$

$$(\text{lt})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\ \delta \quad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\ \delta}{\mathcal{V};\Gamma \vdash e_1 < e_2 : \mathsf{bool}}$$

$$(\text{if})\ \frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{bool} \qquad \mathcal{V};\Gamma \vdash e_2 : \tau \qquad \mathcal{V};\Gamma \vdash e_3 : \tau}{\mathcal{V};\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

(Types identified up to $=_D$)

Figure 5.4: Typing rules

as $\sqrt{2}$ and π. Even the luxury of rational constants is a notational convenience, as we could include only the constant 1 of type $\mathsf{real}\ \mathbf{1}$ and obtain the rest through repeated application of the arithmetic operations.

Notice that zero must be tagged with a dimension explicitly. An alternative is the provision of a constant with type $\forall d.\mathsf{real}\ d$.

Rules (abs) and (rec). Lambda and rec-bound variables must be annotated with their types, thus ensuring that the syntax of a term uniquely determines its typing derivation.

The rec construct provides for general recursive functions, and hence non-terminating programs. We will be defining a call-by-value semantics for Λ_δ, and so the syntax for rec forces it to stand for a *function*. Think of $\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\ e$ as sugar for $\mathit{fix}(\lambda y\colon \tau_1 \to \tau_2.\ \lambda x\colon \tau_1.\ e)$, where *fix* is a generic fixed-point operator.

Rules (dgen) and (dspec). The instantiation and generalisation of dimension variables is made explicit with special constructs $\Lambda d.e$ for dimension abstraction and $e\delta$ for dimension application. This is analogous to System F's type abstraction and type application.

As with the rules for ML_δ, dimensions and types are identified up to the equivalence induced by $=_D$, removing the need for a special (deq) rule.

Rules (add), (sub), (mul) and (div). In contrast to ML_δ, arithmetic is built into the language itself, rather than being provided by a pervasive set of polymorphic functions. This decision is largely one of convenience: passing explicit dimension parameters to every arithmetic operation would make programs extremely verbose.

Rule (lt) and (if). The significance of including a comparison operation and conditional construct to make use of it will become clear in the next chapter when we consider changes in the units of measure used by a program. For the moment, observe that it is possible to write most of the functions described in Chapter 1. The provision of lists would be an easy extension to the language.

It is assumed that programs are typed under an initial type assignment Γ_{base} which contains dimensioned values representing default units for each base dimension, for example $\Gamma_{\text{base}} = \{kg : \mathsf{real}\ \mathrm{M},\ m : \mathsf{real}\ \mathrm{L},\ s : \mathsf{real}\ \mathrm{T}\}$. As mentioned earlier, base dimensions are just a subset of ordinary dimension variables, so an initial set of dimension variables $\mathcal{V}_{\text{base}}$ is also necessary, e.g. $\mathcal{V}_{\text{base}} = \{\mathrm{M}, \mathrm{L}, \mathrm{T}\}$ to match this type assignment.

Our aim in defining an explicitly-typed language is encapsulated in the following result.

Proposition 5.1 (Uniqueness of typing). *For any type assignment Γ, set of dimension variables $\mathcal{V}$ and Λ_δ-expression e the typing derivation $\mathcal{V}; \Gamma \vdash e : \tau$ is unique (up to $=_D$) if it exists at all.*

Proof. Induction on the structure of e. □

5.3 Operational semantics

In this section we describe a call-by-value operational semantics for Λ_δ, in the style sometimes called *natural semantics* [28]. This kind of semantics was used for the formal definition of Standard ML [43]. We define a set of basic *values* (ranged over by v) and define *environments* (ranged over by E) to be finite mappings between identifiers and values. Then an evaluation judgment

$$E \vdash e \Downarrow v$$

means

> "In the context of an environment E the expression e evaluates to the value v".

Values are divided into four kinds: booleans, dimensioned reals, function closures, and dimension abstractions. They are defined by the following grammar:

$$\begin{array}{lcll} v & ::= & \mathsf{true} \mid \mathsf{false} & \textit{booleans} \\ & \mid & \langle r, \delta \rangle & \textit{dimensioned reals} \\ & \mid & \langle E, \lambda x \colon \tau.\, e \rangle & \textit{function closures} \\ & \mid & \langle E, \mathsf{rec}\; y(x \colon \tau_1) \colon \tau_2.\, e \rangle & \textit{recursive function closures} \\ & \mid & \Lambda d.v & \textit{polymorphic values} \end{array}$$

Closures represent yet-to-be-applied functions and consist of an environment E and an abstraction (either lambda or rec) whose free variables refer to values in E. An interpreter for the language is likely to use a similar mechanism. The alternative is to define a subset of the expressions which represent fully-evaluated expressions or *canonicals* [69, 19]. Then instead of an environment in which free variables are looked up, arguments are substituted directly into the body of an abstraction (beta reduction).

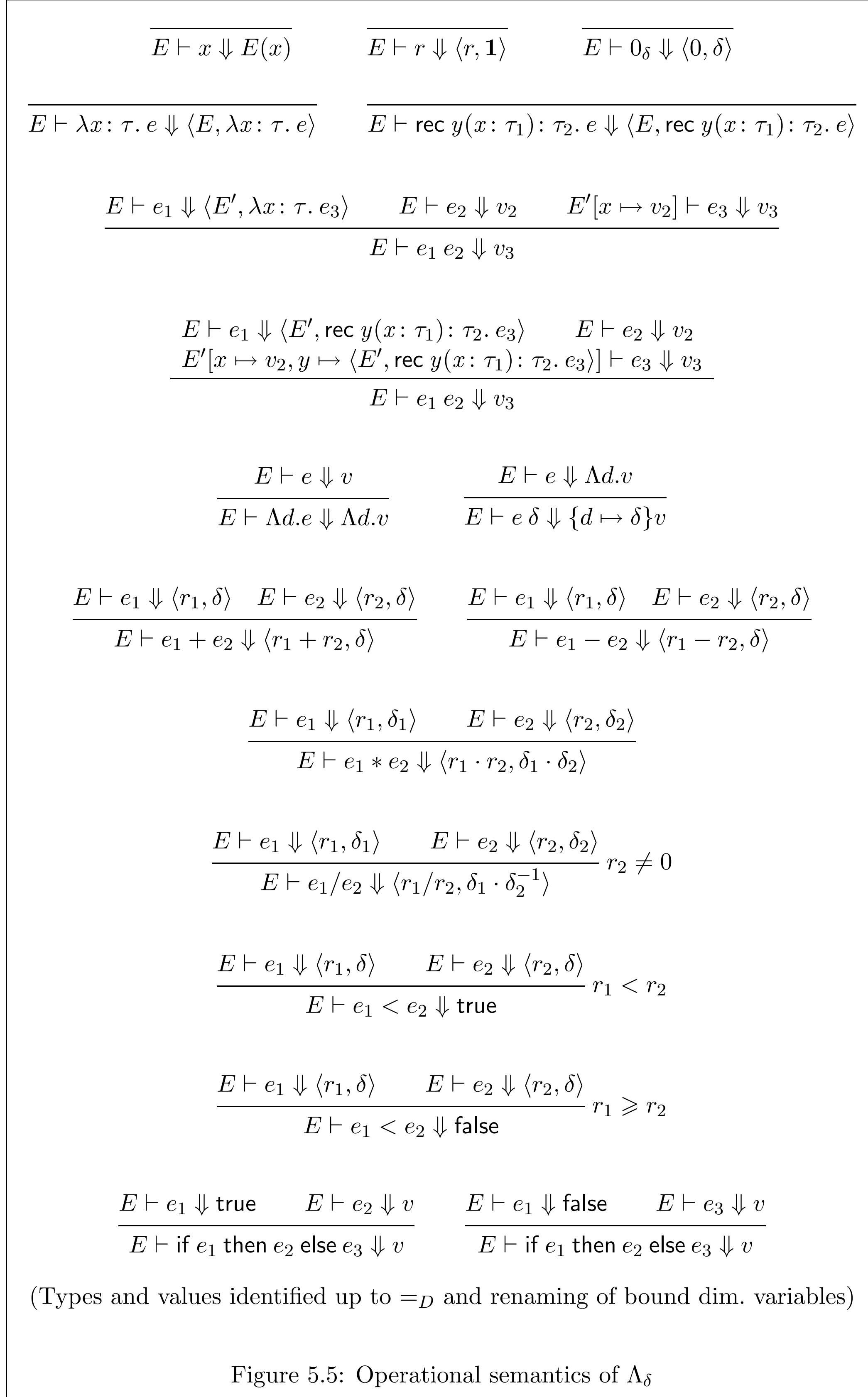
Figure 5.5 gives an inductive definition of the evaluation relation. Evaluation order is not particularly relevant to our study of dimensions, but a call-by-value semantics was chosen to maintain consistency with the implementation of dimension types as an extension to Standard ML. This shows up in the rule for application, where the evaluation of a term $e_1\, e_2$ proceeds by evaluating e_1 to a closure and e_2 to a value v_2, and then extending the closure's environment with this value and evaluating its body under the new environment. If the closure is recursive, then the environment is also extended with a recursive binding.

In contrast to value abstractions ($\lambda x \colon \tau.\, e$), the bodies of dimension abstractions ($\Lambda d.e$) are evaluated 'underneath' the lambda. This means that

$$\Lambda d.\mathsf{rec}\; y(x \colon \mathsf{real}\; d) \colon \mathsf{real}\; d.\; y\; x$$

of type $\forall d.\mathsf{real}\; d \to \mathsf{real}\; d$ loops, whereas

$$\lambda z \colon \mathsf{real}\; \mathbf{1}.\; \mathsf{rec}\; y(x \colon \mathsf{real}\; \mathbf{1}) \colon \mathsf{real}\; \mathbf{1}.\; y\; x$$



$$\frac{}{E \vdash x \Downarrow E(x)} \qquad \frac{}{E \vdash r \Downarrow \langle r, \mathbf{1} \rangle} \qquad \frac{}{E \vdash 0_\delta \Downarrow \langle 0, \delta \rangle}$$

$$\frac{}{E \vdash \lambda x\colon \tau.\, e \Downarrow \langle E, \lambda x\colon \tau.\, e \rangle} \qquad \frac{}{E \vdash \mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\, e \Downarrow \langle E, \mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\, e \rangle}$$

$$\frac{E \vdash e_1 \Downarrow \langle E', \lambda x\colon \tau.\, e_3 \rangle \qquad E \vdash e_2 \Downarrow v_2 \qquad E'[x \mapsto v_2] \vdash e_3 \Downarrow v_3}{E \vdash e_1\ e_2 \Downarrow v_3}$$

$$\frac{\begin{array}{c} E \vdash e_1 \Downarrow \langle E', \mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\, e_3 \rangle \qquad E \vdash e_2 \Downarrow v_2 \\ E'[x \mapsto v_2, y \mapsto \langle E', \mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\, e_3 \rangle] \vdash e_3 \Downarrow v_3 \end{array}}{E \vdash e_1\ e_2 \Downarrow v_3}$$

$$\frac{E \vdash e \Downarrow v}{E \vdash \Lambda d.e \Downarrow \Lambda d.v} \qquad \frac{E \vdash e \Downarrow \Lambda d.v}{E \vdash e\,\delta \Downarrow \{d \mapsto \delta\} v}$$

$$\frac{E \vdash e_1 \Downarrow \langle r_1, \delta \rangle \quad E \vdash e_2 \Downarrow \langle r_2, \delta \rangle}{E \vdash e_1 + e_2 \Downarrow \langle r_1 + r_2, \delta \rangle} \qquad \frac{E \vdash e_1 \Downarrow \langle r_1, \delta \rangle \quad E \vdash e_2 \Downarrow \langle r_2, \delta \rangle}{E \vdash e_1 - e_2 \Downarrow \langle r_1 - r_2, \delta \rangle}$$

$$\frac{E \vdash e_1 \Downarrow \langle r_1, \delta_1 \rangle \qquad E \vdash e_2 \Downarrow \langle r_2, \delta_2 \rangle}{E \vdash e_1 * e_2 \Downarrow \langle r_1 \cdot r_2, \delta_1 \cdot \delta_2 \rangle}$$

$$\frac{E \vdash e_1 \Downarrow \langle r_1, \delta_1 \rangle \qquad E \vdash e_2 \Downarrow \langle r_2, \delta_2 \rangle}{E \vdash e_1 / e_2 \Downarrow \langle r_1 / r_2, \delta_1 \cdot \delta_2^{-1} \rangle}\ r_2 \neq 0$$

$$\frac{E \vdash e_1 \Downarrow \langle r_1, \delta \rangle \qquad E \vdash e_2 \Downarrow \langle r_2, \delta \rangle}{E \vdash e_1 < e_2 \Downarrow \mathsf{true}}\ r_1 < r_2$$

$$\frac{E \vdash e_1 \Downarrow \langle r_1, \delta \rangle \qquad E \vdash e_2 \Downarrow \langle r_2, \delta \rangle}{E \vdash e_1 < e_2 \Downarrow \mathsf{false}}\ r_1 \geqslant r_2$$

$$\frac{E \vdash e_1 \Downarrow \mathsf{true} \qquad E \vdash e_2 \Downarrow v}{E \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow \mathsf{false} \qquad E \vdash e_3 \Downarrow v}{E \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}$$

(Types and values identified up to $=_D$ and renaming of bound dim. variables)

Figure 5.5: Operational semantics of Λ_δ

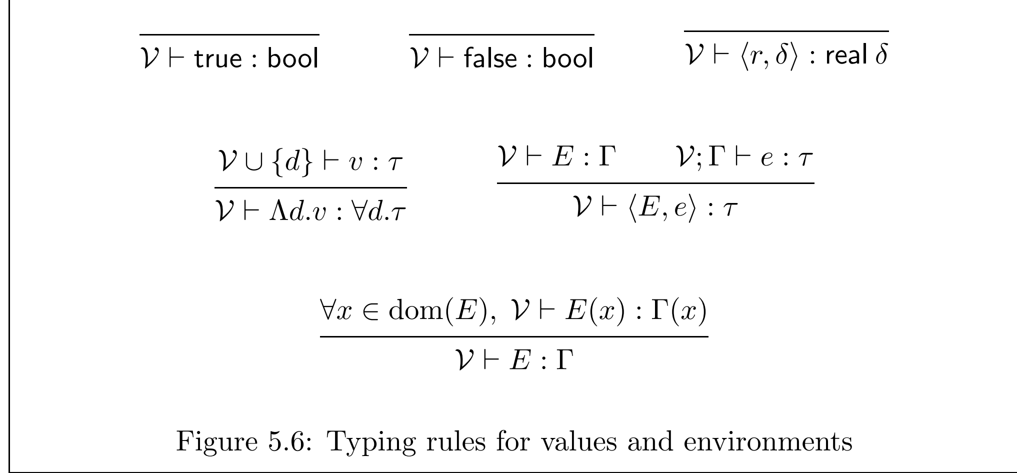


Figure 5.6: Typing rules for values and environments

of type real **1** → real **1** does not. Again, this is consistent with an ML-like language. We can think of dimensions as no more than 'annotations' to a program which ensure dimensional consistency and convey special properties of the program.

The rule for dimension application substitutes a dimension δ for a dimension variable d. As with the substitution on type schemes defined in Chapter 2, substitution on types, expressions and values is assumed to be *capture-avoiding*: bound dimension variables in the type $\forall d.\tau$, the expression $\Lambda d.e$ or the value $\Lambda d.v$ are renamed to prevent capture by a free variable in δ.

Note how the rules dealing with arithmetic and comparison are defined only for dimensionally consistent expressions, and there is no rule for a division by zero. The symbols $+$, $-$, $\cdot$, and $/$ stand for the conventional arithmetic operations on the rationals.

5.4 Semantic soundness

We want to be sure that the operational semantics respects types and dimensions. In order to formalise this, we first need a typing relation for values and environments, defined inductively by the set of rules shown in Figure 5.6. As with the rules for expressions, judgments are assumed to be well-formed: $\mathcal{V} \vdash v : \tau$ is well-formed if $\mathrm{fdv}(v) \cup \mathrm{fdv}(\tau) \subseteq \mathcal{V}$. Free dimension variables in values and environments are specified by the following equations:

$$\begin{aligned}
\mathrm{fdv}(\mathsf{true}) = \mathrm{fdv}(\mathsf{false}) &= \emptyset \\
\mathrm{fdv}(\langle r, \delta \rangle) &= \mathrm{fdv}(\delta) \\
\mathrm{fdv}(\langle E, e \rangle) &= \mathrm{fdv}(E) \cup \mathrm{fdv}(e) \\
\mathrm{fdv}(\Lambda d.v) &= \mathrm{fdv}(v) \setminus \{d\} \\
\mathrm{fdv}(E) &= \bigcup_{x \in \mathrm{dom}(E)} \mathrm{fdv}(E(x))
\end{aligned}$$

We will sometimes use the notation $\text{Val}(\mathcal{V}, \tau)$ to stand for all possible values of type τ containing free dimension variables from $\mathcal{V}$, and $\text{Env}(\mathcal{V}, \Gamma)$ for all possible environments matching the type assignment Γ with dimension variables drawn from $\mathcal{V}$. Also $\text{Expr}(\mathcal{V}, \Gamma, \tau)$ stands for the set of all expressions which have the type τ under a set of dimension variables $\mathcal{V}$ and type assignment Γ. Formally,

$$\begin{aligned} \text{Val}(\mathcal{V}, \tau) &= \{ \, v \mid \mathcal{V} \vdash v : \tau \, \}, \\ \text{Env}(\mathcal{V}, \Gamma) &= \{ \, E \mid \mathcal{V} \vdash E : \Gamma \, \}, \\ \text{Expr}(\mathcal{V}, \Gamma, \tau) &= \{ \, e \mid \mathcal{V}; \Gamma \vdash e : \tau \, \}. \end{aligned}$$

Proposition 5.2 (Semantic soundness). *The semantics respects types and dimensions: if* $\mathcal{V}; \Gamma \vdash e : \tau$ *and* $\mathcal{V} \vdash E : \Gamma$ *then*

$$E \vdash e \Downarrow v \;\Rightarrow\; \mathcal{V} \vdash v : \tau.$$

Proof. Induction on the derivation of $E \vdash e \Downarrow v$. □

Unfortunately, this is not enough to convince us that “well-dimensioned programs do not go wrong”. It only says that *if* an expression e evaluates to a value v then we can be sure that there were no dimension errors. It may *fail* to evaluate to a value because it loops, or because it divides by zero, or because there is a dimension error. The usual solution to this problem is the introduction of a value wrong representing a type error. This approach is taken by Leroy, Tofte and others in their studies of ML [37, 64]. It is a clumsy technique because of the number of extra evaluation rules required. Although only a few rules actually introduce wrong, all the other rules must propagate such type errors to the top level. An alternative is to define a small-step *reduction* semantics which clearly distinguishes between non-terminating programs (infinite sequences of reductions) and programs which contain run-time type errors (sequences of reductions which ‘get stuck’ and never reach a value). Wright and Felleisen take this approach, which they call *syntactic type soundness*; they also produce a good survey of other research into the problem [70].

The addition of a wrong value is a nasty complication, and reduction semantics is too fine-grained for our purposes. Instead, we introduce the idea of erasing dimensions from types and expressions and then show that erasure makes no difference to evaluation. By a simple argument this indicates that no dimension errors can occur when evaluating well-typed expressions. Note that we do not prove the absence of ordinary type errors (e.g. attempting to apply a dimensionless number as if it were a function); it should be clear from our rules and the many previous papers on the subject that these cannot occur.

The type $\tau^\star$ is the result of replacing all dimensions in τ with the unit dimension and removing all quantifiers to leave a *dimensionless* type. To erase dimensions from a term e to give $e^\star$, all explicit types are made dimensionless, and dimension abstractions and dimension applications are removed. Erasure is also defined for values and environments in the obvious way. All this is formalised by the inductive definition given in Figure 5.7.

A straightforward lemma assures us that the type structure of the term is not lost after erasing dimensions:

Types:

$$
\begin{aligned}
\mathsf{bool}^\star &= \mathsf{bool} \\
(\mathsf{real}\ \delta)^\star &= \mathsf{real}\ \mathbf{1} \\
(\tau_1 \to \tau_2)^\star &= {\tau_1}^\star \to {\tau_2}^\star \\
(\forall d.\tau)^\star &= \tau^\star
\end{aligned}
$$

Expressions:

$$
\begin{aligned}
x^\star &= x \\
r^\star &= r \\
{0_\delta}^\star &= 0_{\mathbf{1}} \\
(\lambda x\colon \tau.\, e)^\star &= \lambda x\colon \tau^\star .\, e^\star \\
(e_1\ e_2)^\star &= {e_1}^\star\ {e_2}^\star \\
(\Lambda d.e)^\star &= e^\star \\
(e\ \delta)^\star &= e^\star \\
(\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\, e)^\star &= \mathsf{rec}\ y(x\colon {\tau_1}^\star)\colon {\tau_2}^\star .\, e^\star \\
(e_1 + e_2)^\star &= {e_1}^\star + {e_2}^\star \\
(e_1 - e_2)^\star &= {e_1}^\star - {e_2}^\star \\
(e_1 * e_2)^\star &= {e_1}^\star * {e_2}^\star \\
(e_1 / e_2)^\star &= {e_1}^\star / {e_2}^\star \\
(e_1 < e_2)^\star &= {e_1}^\star < {e_2}^\star \\
(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)^\star &= \mathsf{if}\ e_1^\star\ \mathsf{then}\ e_2^\star\ \mathsf{else}\ e_3^\star
\end{aligned}
$$

Values:

$$
\begin{aligned}
\mathsf{true}^\star &= \mathsf{true} \\
\mathsf{false}^\star &= \mathsf{false} \\
\langle r, \delta\rangle^\star &= \langle r, \mathbf{1}\rangle \\
\langle E, e\rangle^\star &= \langle E^\star, e^\star\rangle \\
\Lambda d.v^\star &= v^\star
\end{aligned}
$$

Environments:

$$
\text{for all } x, \quad E^\star(x) = (E(x))^\star
$$

Figure 5.7: Dimension erasure

Lemma 5.3. *If* $\mathcal{V};\Gamma \vdash e : \tau$ *then* $\emptyset;\Gamma^\star \vdash e^\star : \tau^\star$. *Furthermore, if* $\mathcal{V} \vdash v : \tau$ *then* $\emptyset \vdash v^\star : \tau^\star$, *and if* $\mathcal{V} \vdash E : \Gamma$ *then* $\emptyset \vdash E^\star : \Gamma^\star$.

Proof. Induction on e or v. □

Now the result we have been seeking is the following.

Theorem 5.4 (Dimension erasure). *Dimensions can be thrown away at run-time. That is, if* $\mathcal{V};\Gamma \vdash e : \tau$ *and* $\mathcal{V} \vdash E : \Gamma$ *then*

$$E \vdash e \Downarrow v \iff E^\star \vdash e^\star \Downarrow v^\star.$$

The reverse direction of this equivalence says that if an expression evaluates to some value without checking dimensions then it would have evaluated to the same value with dimension checking turned on. So we can be sure that an expression never fails to evaluate just because of a dimension error: if there is no v such that $E \vdash e \Downarrow v$ then either e is a non-terminating computation or a division by zero has occurred during evaluation.

Proof.
($\Rightarrow$) Easy induction on the expression e.
($\Leftarrow$) We actually wish to prove that

$$\text{if for some } v',\ E^\star \vdash e^\star \Downarrow v' \text{ then } E \vdash e \Downarrow v \text{ with } v^\star = v'.$$

Again, the proof is a straightforward induction on the structure of e. □

Because the erasure defined on expressions removes dimension applications and abstractions, this result also tells us that evaluation is not 'held up' by dimension abstractions, as we hoped.

5.5 Translation from ML-like language

We now define the semantics of the dimension types fragment of ML_δ by a translation into the explicitly-typed language Λ_δ of this chapter. (To be precise: we assume that expressions in the source language make no use of *type* polymorphism in let constructs). This is done by translating *typing derivations* of expressions in the syntax-directed variant of ML_δ into typed expressions in Λ_δ. A similar approach is taken by Harper and Mitchell [20]. The only challenge is in the rule for let:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \mathrm{Gen}(\Gamma,\tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

If $\mathrm{Gen}(\Gamma,\tau_1) = \forall\vec{d}.\tau_1$ for some list of dimension variables $\vec{d}$ then the obvious translation of the construct is the following:

$$(\lambda x : \forall\vec{d}.\tau_1.\ e_2')\ (\Lambda\vec{d}.e_1').$$

Unfortunately, as discussed at length in Chapter 2, generalisation needs to be more sophisticated so that sometimes $\text{Gen}(\Gamma, \tau_1) \not\cong_D \forall \vec{d}.\tau_1$. Therefore we first apply Lemma 2.16 to obtain a derivation in which every occurrence of the (let′) rule uses only the naive generalisation procedure NGen.

Then the translation is specified simply by extending the syntax-directed inference rules of Figure 2.2 with translated expressions, so that a typing judgment now has the form

$$\Gamma \vdash e \leadsto e' : \tau$$

which means

> "In the context of type assignment Γ, the ML_δ-expression e has type τ and translates to the Λ_δ-expression e'."

The new rules are shown in Figure 5.8. Notice that none of the rules deals with arithmetic operations, because these were not primitive operations in ML_δ. Instead, their types were assumed to be present in a pervasive type assignment. Hence their semantics is defined by a pervasive value environment which matches this type assignment, as follows:

$$\begin{aligned}
\{ \, \mathsf{+} \; &: \; \Lambda d.\lambda x : \mathsf{real}\; d.\; \lambda y : \mathsf{real}\; d.\; x + y, \\
\mathsf{-} \; &: \; \Lambda d.\lambda x : \mathsf{real}\; d.\; \lambda y : \mathsf{real}\; d.\; x - y, \\
\mathsf{*} \; &: \; \Lambda d_1.\Lambda d_2.\lambda x : \mathsf{real}\; d_1.\; \lambda y : \mathsf{real}\; d_2.\; x * y, \\
\mathsf{/} \; &: \; \Lambda d_1.\Lambda d_2.\lambda x : \mathsf{real}\; d_1.\; \lambda y : \mathsf{real}\; d_2.\; x / y, \\
\mathsf{<} \; &: \; \Lambda d.\lambda x : \mathsf{real}\; d.\; \lambda y : \mathsf{real}\; d.\; x < y \, \}.
\end{aligned}$$

We will write $\Gamma \vdash e \leadsto e' : \tau$ to indicate that there is a derivation of $\Gamma \vdash e : \tau$ in which the expression e translates to e'. We want two properties to hold of this translation. First, it should preserve types.

Proposition 5.5 (Soundness). *If* $\Gamma \vdash e \leadsto e' : \tau$ *then* $\mathcal{V}; \Gamma \vdash_\delta e' : \tau$ *where* $\mathcal{V} = \text{fdv}(\Gamma) \cup \text{fdv}(\tau)$.

Proof. Induction on the structure of e. □

$$(\mathrm{var}')\ \frac{}{\Gamma \vdash x \rightsquigarrow x\,\vec{\delta} : \{\vec{d} \mapsto \vec{\delta}\}\tau}\ \Gamma(x) = \forall \vec{d}.\tau$$

$$(\mathrm{const})\ \frac{}{\Gamma \vdash r \rightsquigarrow r : \mathsf{real}\ \mathbf{1}}\ \ r \neq 0 \qquad (\mathrm{zero})\ \frac{}{\Gamma \vdash 0 \rightsquigarrow 0_\delta : \mathsf{real}\ \delta}$$

$$(\mathrm{abs})\ \frac{\Gamma[x : \tau_1] \vdash e \rightsquigarrow e' : \tau_2}{\Gamma \vdash \lambda x.e \rightsquigarrow \lambda x{:}\,\tau_1.\,e' : \tau_1 \rightarrow \tau_2}$$

$$(\mathrm{app})\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_1}{\Gamma \vdash e_1\, e_2 \rightsquigarrow e_1'\, e_2' : \tau_2}$$

$$(\mathrm{let}')\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma[x : \forall \vec{d}.\tau_1] \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow (\lambda x{:}\,\forall \vec{d}.\tau_1.\,e_2')\,(\Lambda \vec{d}.e_1') : \tau_2}\ \ \vec{d} = \mathrm{fdv}(\tau_1) \setminus \mathrm{fdv}(\Gamma)$$

$$(\mathrm{letrec})\ \frac{\begin{array}{c}\Gamma[x : \tau_1, y : \tau_1 \rightarrow \tau_2] \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow \tau_2 \\ \Gamma[y : \tau_1 \rightarrow \tau_2] \vdash e_2 \rightsquigarrow e_2' : \tau_3\end{array}}{\Gamma \vdash \mathsf{letrec}\ y(x) = e_1\ \mathsf{in}\ e_2 \rightsquigarrow (\lambda y{:}\,\tau_1 \rightarrow \tau_2.\,e_2')\,(\mathsf{rec}\ y(x{:}\,\tau_1){:}\,\tau_2.\,e_1') : \tau_3}$$

$$(\mathrm{if})\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{bool} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau \qquad \Gamma \vdash e_3 \rightsquigarrow e_3' : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightsquigarrow \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3' : \tau}$$

Figure 5.8: Translation from ML_δ into Λ_δ

Second, the translation of different (normalised) derivations of the same expression should not assign different meanings to the expression. This property is called *coherence*. For our language, it is sufficient to show that the translated expressions have the same dimension erasure: Theorem 5.4 then implies that the expressions have the same behaviour (in a sense to be defined formally in the next chapter, the expressions are *observationally equivalent*).

Proposition 5.6 (Coherence). *If* $\Gamma \vdash e \leadsto e_1 : \tau$ *and* $\Gamma \vdash e \leadsto e_2 : \tau$ *then* ${e_1}^\star = {e_2}^\star$.

Proof. Induction on the structure of e. □

To see why a result of this kind is required, consider the expression

$$e \stackrel{\text{def}}{=} \mathsf{let}\ s = \lambda z.z * z\ \mathsf{in}\ s\ 5.$$

Two distinct typing derivations of $\vdash e : \mathsf{real}\ \mathbf{1}$ give rise to two distinct translations:

$$\vdash e \leadsto (\lambda s\colon \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}.\ s\ 5)\ (\lambda z\colon \mathsf{real}\ \mathbf{1}.\ z * z) : \mathsf{real}\ \mathbf{1}$$

and

$$\vdash e \leadsto (\lambda s\colon (\forall d.\mathsf{real}\ d \to \mathsf{real}\ d^2).\ s\ \mathbf{1}\ 5)\ (\Lambda d.\lambda z\colon \mathsf{real}\ d.\ z * z) : \mathsf{real}\ \mathbf{1}.$$

The resulting Λ_δ expressions are not the same, even up to alpha-conversion, so we need Proposition 5.6 to ensure that their meanings are identical.

Chapter 6

Denotational semantics

In the previous chapter we were interested in whether or not dimension errors can occur during the *evaluation* of a well-typed program. This emphasis on evaluation led us to define an *operational* semantics for the language.

In this chapter we will describe a more abstract model of the language in order to investigate some more general properties in the next chapter. This is the *denotational* approach, in which an expression in the language *denotes* some element of a mathematical structure. Ideally programs would be modelled as ordinary mathematical functions, so that, for instance, a program of type $\forall d.\mathsf{real}\ d \rightarrow \mathsf{real}\ d^2$ is an element of the function space $\mathbb{Q} \rightarrow \mathbb{Q}$ and the function $\lambda x.x * x$ with that type *really is* the function which squares its argument. Unfortunately this approach does not work. First, the possibility of non-termination must be dealt with, and second, it is not possible to treat recursion and higher-order functions in a simple model based on sets. The closest we can get to this mathematical ideal is a model based on complete partial orders, or *domains*.

The chapter is structured as follows. First we motivate the semantics by considering ordinary type polymorphism in ML and the analogous situation for dimension types. In Section 6.2 we define the denotational semantics for Λ_δ and in Section 6.3 prove that it agrees with the operational semantics in a useful way. Then in Section 6.4 we prove the *dimensional invariance* theorem. This is used in the next chapter as a powerful tool for reasoning about programs in Λ_δ.

6.1 Motivation

Types as properties

One way of viewing a type is as a *property*; this idea is taken to its logical conclusion in type theory [63] where the type of a program type is a complete *specification* of its behaviour. In the ML type system, and in the dimension type system presented here, a type represents a *partial* specification.

Consider a function f in ML with the following polymorphic type:

$$f : \forall t.t \rightarrow (t \times t).$$

What could this function be? One possibility is the pathological $\lambda x.\, loop$ where *loop* is a divergent expression. In Standard ML another candidate is the expression

```
fn x => raise Alarm
```

Then of course there is $\lambda x.\,(x,x)$ and in fact this is the *only* terminating function with the type given.

This observation is justified by a simple property of f which can be inferred purely from its type: that for any function k,

$$f(k(x)) = k^2(f(x))$$

where the function k^2 is given by

$$k^2(x,y) = (k(x), k(y)).$$

Wadler calls such results 'theorems for free' [66] because it is not necessary to know anything about f other than its type.

Consider also a function g in ML with the type

$$g : \forall t_1.\forall t_2.t_1 \to t_2.$$

This time, the only function with this type is the non-terminating function, or in Standard ML, one which always raises an exception. Again, this property was deduced purely from the type.

Dimension types as scaling properties

Now consider a function written in ML_δ with an polymorphic dimension type analogous to the type of f from above:

$$f : \forall d.\mathsf{real}\ d \to \mathsf{real}\ d^2.$$

What could this function be? Again, there is a pathological function which never terminates. This time there are many examples of functions which *do* terminate. However, they all have the property that for any constant $k > 0$,

$$f(k * x) = k^2 * f(x).$$

In essence, this says that if the argument to f is scaled then the result scales in a corresponding way.

The dimensional invariance theorem proved at the end of this chapter captures this idea of 'invariance under scaling'. It is used in Chapter 7 to prove equivalences such as the above, and also to show that certain types are not 'inhabited' except by trivial expressions. For example, there are no expressions g with the type

$$g : \forall d.\mathsf{real}\ d^2 \to \mathsf{real}\ d$$

except for trivial functions which return zero or do not terminate.

6.2 The denotational semantics

The meanings of expressions in Λ_δ will be defined using domains, or complete partial orders. Traditionally, domains have been a useful tool for reasoning about 'flow of control' properties of programs which use higher-order functions or recursive datatypes (see, for example, work on strictness analysis [4]). We are merely interested in properties of the *data*, *viz.* real numbers with dimensions, so we will not be concerned with these matters. However, for concreteness it seems appropriate to use a cpo-based model.

Mathematical preliminaries

The terminology and notation of domain theory varies slightly from author to author. To prevent confusion, in this section we define all notation used in this chapter. For a readable introduction to domains see Winksel's book on semantics [69] or Davey and Priestley's textbook on the theory of ordered sets [14].

Let P be a partially ordered set: a set with an operation which is reflexive, transitive and anti-symmetric, usually written $\sqsubseteq$. Then P is a complete partial order (cpo or *domain* for short) if least upper bounds exist for all chains $\{p_i\}_{i \in \mathbb{N}}$ in P, written $\bigsqcup_{i \in \mathbb{N}} p_i$. We do not require every cpo to contain a bottom element, but when it does, it is denoted by $\bot$.

For two domains D and E the *continuous function space* $D \to_c E$ is the domain containing all continuous functions between D and E, with a pointwise ordering.

Any domain D can be *lifted* to give a domain $D_\bot$ with a new bottom element:

$$D_\bot = \{\bot\} \cup \{\ [d]\ \mid\ d \in D\ \}$$

The notation $[d]$ just makes clear that the non-bottom elements of $D_\bot$ are 'tagged' so that they are distinct from the new element $\bot$.

If a function $f : D \to E$ is continuous and E already has a bottom element then we can *lift* f to get a continuous function $f_\bot : D_\bot \to E$ defined by

$$\begin{aligned} f_\bot([d]) &= f(d) \\ f_\bot(\bot) &= \bot. \end{aligned}$$

Similarly, a relation $R \subseteq D \times E$ can be lifted to give a relation $R_\bot \subseteq D_\bot \times E_\bot$ given by

$$R_\bot = \{(\bot, \bot)\}\ \cup\ \{\ ([d]\,, [e])\ \mid\ (d, e) \in R\ \}\,.$$

Domains of values

We start by defining for each type τ a domain of values $[\![\tau]\!]^\star$, specified inductively in Figure 6.1.

It is instructive to compare Figure 6.1 to the definition of values in the operational semantics given on page 80 in Section 5.3. Notice how we have ignored the dimension component δ and simply model 'reals' in the language as elements

$$
\begin{aligned}
[\![\mathsf{bool}]\!]^\star &= \mathbb{B} \\
[\![\mathsf{real}\ \delta]\!]^\star &= \mathbb{Q} \\
[\![\tau_1 \to \tau_2]\!]^\star &= [\![\tau_1]\!]^\star \to_c [\![\tau_2]\!]^\star_\perp \\
[\![\forall d.\ \tau]\!]^\star &= [\![\tau]\!]^\star
\end{aligned}
$$

Figure 6.1: Domains of values

of $\mathbb{Q}$ in contrast to the pair $\langle r, \delta \rangle$ used in the operational semantics. The definition likewise ignores dimension quantification and does not distinguish between monomorphic and polymorphic types. We already know that the evaluation of a well-typed expression does not depend on the dimensions in its type, so it is safe (in a certain sense) to throw away the dimensions when defining a denotational semantics. To emphasise this, we write $[\![\tau]\!]^\star$ instead of the more usual $[\![\tau]\!]$ to indicate that we are really defining a domain for $\tau^\star$, the dimension-erasure of τ.

Also consider carefully our treatment of functions. Arguments to functions are just *values*, *i.e.* fully evaluated expressions, but functions may not terminate, so the result of a function is modelled by a lifted domain in which $\perp$ represents non-termination.

We will freely use b to range over elements of $\mathbb{B}$, use r to range over $\mathbb{Q}$, use f to stand for a function, and use w to range over semantic values including all of the above.

Semantic function

The operational semantics was presented by specifying the result of evaluating an expression e in some environment E. We make use of a similar notion in the denotational semantics. Here an *environment* ρ is a finite map from identifiers to values belonging to some semantic domain. Then the meaning of an expression e in the context of an environment ρ is given by $[\![e]\!]^\star(\rho)$. The rules which define this *semantic function* inductively are shown in Figure 6.2. This is all very standard, as we are in effect just modelling the semantics of dimensionless expressions given by $e^\star$, which explains the notation $[\![e]\!]^\star$ again. Winksel's treatment corresponds closely [69].

As with the operational semantics, we require that the denotational semantics is well-defined with respect to the typing rules. We first formalise what it means for the domains in an environment ρ to 'match' the types in a type assigment Γ. If Γ is a finite mapping between identifiers and types, then $[\![\Gamma]\!]^\star$ is the set of all finite mappings between the same identifiers and elements of the domain of values corresponding to the identifier's type in Γ:

$$[\![\Gamma]\!]^\star = \{\ \rho \ \mid\ \mathrm{dom}(\rho) = \mathrm{dom}(\Gamma) \text{ and } \rho(x) \in [\![\Gamma(x)]\!]^\star \text{ for all } x \in \mathrm{dom}(\Gamma)\ \}$$

Now we can prove the following lemma which shows that the denotational semantics is well-defined. This is analogous to the semantic soundness result for

$$
\begin{aligned}
[\![\cdot]\!]^\star \quad &: \quad \mathrm{Expr}(\mathcal{V},\Gamma,\tau) \to ([\![\Gamma]\!]^\star \to [\![\tau]\!]^\star_\perp) \\[1em]
[\![x]\!]^\star(\rho) &= [\rho(x)] \\
[\![r]\!]^\star(\rho) &= [r] \\
[\![0_\delta]\!]^\star(\rho) &= [0] \\
[\![\lambda x\colon \tau.\, e]\!]^\star(\rho) &= [\lambda w \in [\![\tau]\!]^\star.\, [\![e]\!]^\star(\rho[x \mapsto w])] \\
[\![e_1\, e_2]\!]^\star(\rho) &= \begin{cases} w_1(w_2) & \text{if } [\![e_1]\!]^\star(\rho) = [w_1] \text{ and } [\![e_2]\!]^\star(\rho) = [w_2]\,, \\ \perp & \text{otherwise.} \end{cases} \\
[\![\Lambda d.e]\!]^\star(\rho) &= [\![e]\!]^\star(\rho) \\
[\![e\,\delta]\!]^\star(\rho) &= [\![e]\!]^\star(\rho) \\[1em]
[\![\mathsf{rec}\; y(x\colon \tau_1)\colon \tau_2.\, e]\!]^\star(\rho) &= \Big[\bigsqcup_{i\in\mathbb{N}} w_i\Big]\,, \text{ where} \\
& \quad\; w_0 = \lambda w \in [\![\tau_1]\!]^\star.\, \perp \\
& \quad\; w_{i+1} = \lambda w \in [\![\tau_1]\!]^\star.\, [\![e]\!]^\star(\rho[x \mapsto w,\, y \mapsto w_i]) \\[1em]
[\![e_1 + e_2]\!]^\star(\rho) &= \begin{cases} [r_1 + r_2] & \text{if } [\![e_1]\!]^\star(\rho) = [r_1] \text{ and } [\![e_2]\!]^\star(\rho) = [r_2]\,, \\ \perp & \text{otherwise.} \end{cases} \\
[\![e_1 - e_2]\!]^\star(\rho) &= \begin{cases} [r_1 - r_2] & \text{if } [\![e_1]\!]^\star(\rho) = [r_1] \text{ and } [\![e_2]\!]^\star(\rho) = [r_2]\,, \\ \perp & \text{otherwise.} \end{cases} \\
[\![e_1 * e_2]\!]^\star(\rho) &= \begin{cases} [r_1 \cdot r_2] & \text{if } [\![e_1]\!]^\star(\rho) = [r_1] \text{ and } [\![e_2]\!]^\star(\rho) = [r_2]\,, \\ \perp & \text{otherwise.} \end{cases} \\
[\![e_1 / e_2]\!]^\star(\rho) &= \begin{cases} [r_1 / r_2] & \text{if } [\![e_1]\!]^\star(\rho) = [r_1] \text{ and } [\![e_2]\!]^\star(\rho) = [r_2] \\ & \text{and } r_2 \neq 0, \\ \perp & \text{otherwise.} \end{cases} \\
[\![e_1 < e_2]\!]^\star(\rho) &= \begin{cases} [\mathsf{true}] & \text{if } [\![e_1]\!]^\star(\rho) = [r_1] \text{ and } [\![e_2]\!]^\star(\rho) = [r_2] \\ & \text{and } r_1 < r_2, \\ [\mathsf{false}] & \text{if } [\![e_1]\!]^\star(\rho) = [r_1] \text{ and } [\![e_2]\!]^\star(\rho) = [r_2] \\ & \text{and } r_1 \geqslant r_2, \\ \perp & \text{otherwise.} \end{cases} \\
[\![\mathsf{if}\; e_1\; \mathsf{then}\; e_2\; \mathsf{else}\; e_3]\!]^\star(\rho) &= \begin{cases} [\![e_2]\!]^\star(\rho) & \text{if } [\![e_1]\!]^\star(\rho) = [\mathsf{true}]\,, \\ [\![e_3]\!]^\star(\rho) & \text{if } [\![e_1]\!]^\star(\rho) = [\mathsf{false}]\,, \\ \perp & \text{otherwise.} \end{cases}
\end{aligned}
$$

Figure 6.2: Denotational semantics

the operational semantics.

Lemma 6.1. *If* $\mathcal{V};\Gamma \vdash e : \tau$ *and* $\rho \in [\![\Gamma]\!]^\star$ *then* $[\![e]\!]^\star(\rho) \in [\![\tau]\!]^\star_\perp$.

Proof. Induction on e. □

6.3 Relating the two semantics

The results proved using the denotational semantics will be of no use unless the operational and denotational semantics agree in some way. For example, we will show later that if $\vdash e : \forall d.\mathsf{real}\, d \rightarrow \mathsf{real}\, d^2$ and $[\![e]\!]^\star(\emptyset) = [f]$ then for any $r \in \mathbb{Q}$ and $k \in \mathbb{Q}^+$, whenever $f(r) = [r']$ it is the case that $f(k \cdot r) = \left[k^2 \cdot r'\right]$. We would like to infer from this that the expression $e\,(k * x)$ is 'equivalent' in some sense to the expression $k * k * e\, x$.

What kind of equivalence do we require? A programmer would say that two program fragments are equivalent if one can be replaced by the other in any complete program without affecting its behaviour. Formalising this notion, an expression e_1 is said to be *contextually equivalent* to another expression e_2 if for any program context $\mathcal{C}[\cdot]$ the observable behaviour of $\mathcal{C}[e_1]$ is the same as the observable behaviour of $\mathcal{C}[e_2]$.

We must now make precise what we mean by *program context* and *observable behaviour*. We choose to take programs to be closed expressions of type $\mathsf{real}\,\mathbf{1}$. It turns out that it makes no difference if the expressions are allowed to make use of dimensioned units drawn from Γ_{base}, so for simplicity we consider only closed expressions; we will justify this formally later. Then a $(\mathcal{V}, \Gamma, \tau)$-context $\mathcal{C}[\phi]$ is a expression with a 'hole' ϕ in it which is to be filled by any expression e which has a typing $\mathcal{V};\Gamma \vdash e : \tau$, so that $\vdash \mathcal{C}[e] : \mathsf{real}\,\mathbf{1}$. Here $\mathcal{C}[e]$ stands for the result of replacing all occurrences of ϕ in $\mathcal{C}[\phi]$ by e, permitting variable capture in contrast to our standard definition of substitution. For example, the program

$$(\Lambda d.\ \lambda x\colon \mathsf{real}\, d.\ \lambda y\colon \mathsf{real}\, d.\ \phi\, x * \phi\, y)\ \mathbf{1}\ 5\ 6$$

is a $(d, \{x : \mathsf{real}\, d, y : \mathsf{real}\, d\}, \mathsf{real}\, d^2)$-context.

For observable behaviour, we choose to observe the value of type $\mathsf{real}\,\mathbf{1}$ which results from evaluating $\mathcal{C}[e]$. Then two expressions e_1 and e_2 with typings $\mathcal{V};\Gamma \vdash e_1 : \tau$ and $\mathcal{V};\Gamma \vdash e_2 : \tau$ are contextually (or observationally) equivalent if for any $(\mathcal{V}, \Gamma, \tau)$-context $\mathcal{C}[\phi]$,

$$\vdash \mathcal{C}[e_1] \Downarrow \langle r, \mathbf{1}\rangle \text{ if and only if } \vdash \mathcal{C}[e_2] \Downarrow \langle r, \mathbf{1}\rangle.$$

We write $e_1 \approx e_2$.

Full abstraction

The two styles of semantics would match up completely if for any two expressions e_1 and e_2,

$$e_1 \approx e_2 \iff [\![e_1]\!]^\star = [\![e_2]\!]^\star.$$

This is known as *full abstraction*. Our semantics is not fully abstract—it is a very difficult problem to find such a semantics for 'sequential' languages such as the one discussed here [50]. However, equivalences do match up in one direction, that is,

$$[\![e_1]\!]^\star = [\![e_2]\!]^\star \;\Rightarrow\; e_1 \approx e_2.$$

This property is a simple consequence of the *adequacy* of the semantics. It is enough for our purposes, as we are interested only in proving equivalences (and not non-equivalences) in the denotational semantics.

What kind of examples cause the reverse direction to fail? We want two expressions e_1 and e_2 which have different meanings in the denotational semantics ($[\![e_1]\!]^\star \neq [\![e_2]\!]^\star$) but cannot be distinguished by any context in the operational semantics ($e_1 \approx e_2$).

Let R be $\mathsf{real}\ \mathbf{1}$, the type of dimensionless reals. Let *zero* be $\lambda x\colon R.\, 0_{\mathbf{1}}$, the constant zero function. Let *loop* be $\mathsf{rec}\ y(x\colon R)\colon R.\, y\, x$, a function which loops for any argument value. Then consider the two expressions below.

$$\begin{aligned} e_1 &\stackrel{\text{def}}{=} \lambda y\colon (R \to R) \to (R \to R) \to R.\, y\ \mathit{loop}\ \mathit{zero} + y\ \mathit{zero}\ \mathit{loop} \\ \text{and } e_2 &\stackrel{\text{def}}{=} \lambda y\colon (R \to R) \to (R \to R) \to R.\, (y\ \mathit{loop}\ \mathit{loop}) * 2 \end{aligned}$$

There is no context which can distinguish these expressions, because any context would apply the expression to a function which applied one of its arguments (which for both expressions would loop) or applied neither (which for both expressions would return twice the value of the result).

Now consider the function $\mathit{por} \in [\![(R \to R) \to (R \to R) \to R]\!]^\star$ defined by

$$\mathit{por}\ f\ g = \begin{cases} [0] & \text{if } f(0) \neq \bot, \\ [0] & \text{if } g(0) \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

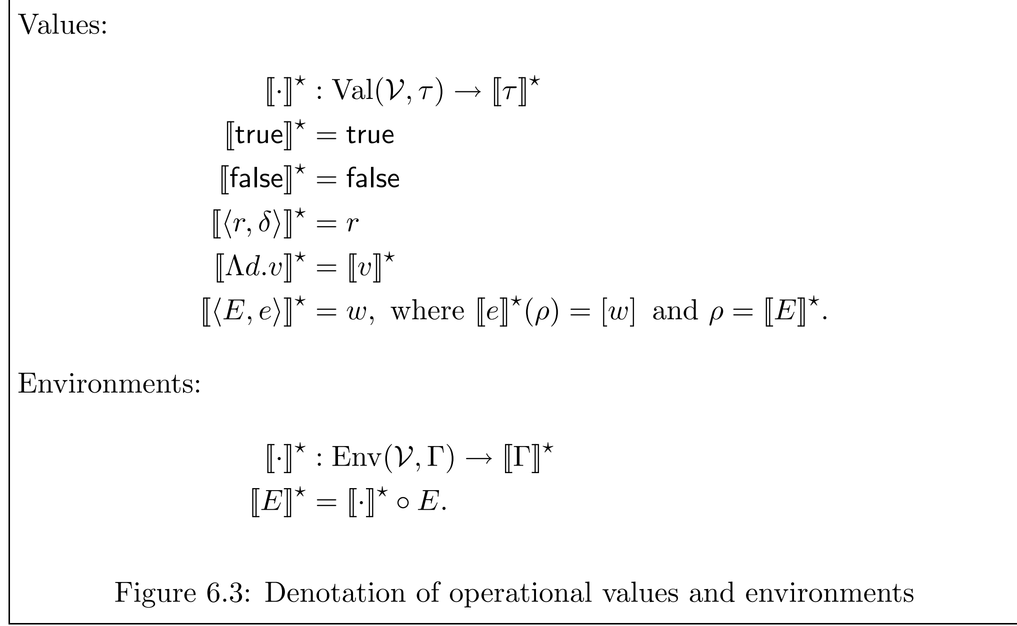
Then $[\![e_1]\!]^\star(\emptyset)(\mathit{por}) = [0]$ whereas $[\![e_2]\!]^\star(\emptyset)(\mathit{por}) = \bot$ so $[\![e_1]\!]^\star \neq [\![e_2]\!]^\star$.

This example is standard for typed lambda calculi such as Λ_δ [52]. Intuitively, there are functions such as *por* in the denotational semantics which 'evaluate their arguments in parallel', and which correspond to no expression in the operational semantics.

In fact, there are expressions in Λ_δ which defeat full abstraction in a different way. Consider the following expressions:

$$\begin{aligned} e_1 &\stackrel{\text{def}}{=} \lambda y\colon (\forall d.\mathsf{real}\ d \to \mathsf{real}\ d).\, y\ \mathbf{1}\ 2 \\ \text{and } e_2 &\stackrel{\text{def}}{=} \lambda y\colon (\forall d.\mathsf{real}\ d \to \mathsf{real}\ d).\, 2 * y\ \mathbf{1}\ 1 \end{aligned}$$

Clearly $[\![e_1]\!]^\star \neq [\![e_2]\!]^\star$ as there are many functions $f \in \mathbb{Q} \to \mathbb{Q}_\bot$ for which $[\![e_1]\!]^\star(\emptyset)(f) \neq [\![e_2]\!]^\star(\emptyset)(f)$. But the type constraint on y actually ensures that the expression $y\ \mathbf{1}\ 2$ always evaluates to the same value as $2 * y\ \mathbf{1}\ 1$, or both fail to evaluate at all. Hence it is actually the case that $e_1 \approx e_2$. Later we will show that this equivalence is an instance of *dimensional invariance*, and use the theorem proved at the end of this chapter to justify it.



Values:

$$
\begin{aligned}
[\![\cdot]\!]^\star &: \mathrm{Val}(\mathcal{V},\tau) \to [\![\tau]\!]^\star \\
[\![\mathsf{true}]\!]^\star &= \mathsf{true} \\
[\![\mathsf{false}]\!]^\star &= \mathsf{false} \\
[\![\langle r, \delta\rangle]\!]^\star &= r \\
[\![\Lambda d.v]\!]^\star &= [\![v]\!]^\star \\
[\![\langle E, e\rangle]\!]^\star &= w, \text{ where } [\![e]\!]^\star(\rho) = [w] \text{ and } \rho = [\![E]\!]^\star.
\end{aligned}
$$

Environments:

$$
\begin{aligned}
[\![\cdot]\!]^\star &: \mathrm{Env}(\mathcal{V},\Gamma) \to [\![\Gamma]\!]^\star \\
[\![E]\!]^\star &= [\![\cdot]\!]^\star \circ E.
\end{aligned}
$$

Figure 6.3: Denotation of operational values and environments

Adequacy

In order to relate the two styles of semantics, we must first define a meaning for values and environments from the operational semantics in terms of the denotational semantics. This is shown in Figure 6.3.

First we prove that the denotational semantics respects the evaluation relation of the operational semantics.

Proposition 6.2 (Soundness). *If $E \vdash e \Downarrow v$ and $\rho = [\![E]\!]^\star$ then $[\![e]\!]^\star(\rho) = [w]$ where $w = [\![v]\!]^\star$.*

Proof. By induction on the derivation of $E \vdash e \Downarrow v$. □

The converse fails to hold, but we can prove a weaker result, that for any *program* e it is the case that

$$[\![e]\!]^\star(\emptyset) = [r] \text{ implies } \vdash e \Downarrow \langle r, \mathbf{1}\rangle.$$

This is more difficult to prove as we need something stronger than the denotation given in Figure 6.3. We construct a *logical relation* [44] between values in the denotational semantics and values in the operational semantics. The relation $\mathcal{A}^{\mathcal{V}}_\tau \subseteq [\![\tau]\!]^\star \times \mathrm{Val}(\mathcal{V},\tau)$ is defined inductively by:

- $\mathcal{A}^{\mathcal{V}}_{\mathsf{bool}}(b, b')$ holds if and only if $b = b'$.
- $\mathcal{A}^{\mathcal{V}}_{\mathsf{real}\,\delta}(r, \langle r', \delta\rangle)$ holds if and only if $r = r'$.
- $\mathcal{A}^{\mathcal{V}}_{\tau_1\to\tau_2}(f, \langle E, \lambda x\colon \tau_1.\, e\rangle)$ holds if and only if for all $w \in [\![\tau_1]\!]^\star$ and $\mathcal{V} \vdash v : \tau_1$ such that $\mathcal{A}^{\mathcal{V}}_{\tau_1}(w, v)$ it is the case that

$$
\begin{aligned}
f(w) = \big[w'\big] \text{ implies } & E[x \mapsto v] \vdash e \Downarrow v' \\
& \text{such that } \mathcal{A}^{\mathcal{V}}_{\tau_2}(w', v').
\end{aligned}
$$

- $\mathcal{A}^{\mathcal{V}}_{\tau_1 \to \tau_2}(f, \langle E, \mathsf{rec}\, y(x\colon \tau_1)\colon \tau_2.\, e\rangle)$ holds if and only if for all $w \in [\![\tau_1]\!]^\star$ and $\mathcal{V} \vdash v : \tau_1$ such that $\mathcal{A}^{\mathcal{V}}_{\tau_1}(w, v)$ it is the case that

$$f(w) = \lfloor w' \rfloor \text{ implies } E[x \mapsto v, y \mapsto \langle E, \mathsf{rec}\, y(x\colon \tau_1)\colon \tau_2.\, e\rangle] \vdash e \Downarrow v' \\ \text{such that } \mathcal{A}^{\mathcal{V}}_{\tau_2}(w', v').$$

- $\mathcal{A}^{\mathcal{V}}_{\forall d.\tau}(w, \Lambda d.v)$ holds if and only if $\mathcal{A}^{\mathcal{V}\cup\{d\}}_{\tau}(w, v)$.

This relation is extended to environments in a pointwise fashion. If $\rho \in [\![\Gamma]\!]^\star$ and $\mathcal{V} \vdash E : \Gamma$ then

- $\mathcal{A}^{\mathcal{V}}_{\Gamma}(\rho, E)$ if and only if for all $x \in \mathrm{dom}(\Gamma)$, $\mathcal{A}^{\mathcal{V}}_{\Gamma(x)}(\rho(x), E(x))$.

Before proving the main theorem, we state a couple of lemmas. The first of these just asserts that dimensions are ignored by the denotational semantics.

Lemma 6.3. *If $\mathcal{A}^{\mathcal{V}}_{\tau}(w, v)$ then $\mathcal{A}^{\mathcal{V}}_{\{d \mapsto \delta\}\tau}(w, \{d \mapsto \delta\}v)$ for any $d \in \mathcal{V}$ and any dimension δ such that $\mathrm{fdv}(\delta) \subseteq \mathcal{V}$.*

We must also show that the relation preserves least upper bounds of infinite chains.

Lemma 6.4. *Let $\{w_i\}_{i\in\mathbb{N}}$ be a chain in $[\![\tau]\!]^\star$ and let $v \in \mathrm{Val}(\mathcal{V}, \tau)$ be a value in the operational semantics. Then if $\mathcal{A}^{\mathcal{V}}_{\tau}(w_i, v)$ for all $i \in \mathbb{N}$, then $\mathcal{A}^{\mathcal{V}}_{\tau}(\bigsqcup_{i\in\mathbb{N}} w_i, v)$.*

Using the logical relation we can prove the following result. The proof is omitted but follows the same pattern as adequacy proofs elsewhere [69, 19, 4] except that we use environments and closures in the operational semantics instead of explicit substitution of values for free variables.

Proposition 6.5. *If $\mathcal{V}; \Gamma \vdash e : \tau$ then for all $\rho \in [\![\Gamma]\!]^\star$ and $\mathcal{V} \vdash E : \Gamma$ such that $\mathcal{A}^{\mathcal{V}}_{\Gamma}(\rho, E)$ it is the case that*

$$[\![e]\!]^\star(\rho) = \lfloor w \rfloor \;\Rightarrow\; E \vdash e \Downarrow v$$

for some v such that $\mathcal{A}^{\mathcal{V}}_{\tau}(w, v)$.

Proof. Induction on the structure of e. □

A corollary of Propositions 6.2 and 6.5 is that the operational and denotational meanings of programs (but not expressions in general) coincide.

Corollary 6.6 (Adequacy). *If $\vdash e : \mathsf{real}\,\mathbf{1}$ then $[\![e]\!]^\star(\emptyset) = \lfloor r \rfloor$ if and only if $\vdash e \Downarrow \langle r, \mathbf{1}\rangle$.*

To prove that the semantics is sound with respect to observational equivalence we need the following lemma which states that the semantics is *compositional*: that is, the meaning of compound expressions is defined only in terms of the meaning of their subexpressions.

Lemma 6.7 (Compositionality). *If* $\mathcal{V};\Gamma \vdash e_1 : \tau$ *and* $\mathcal{V};\Gamma \vdash e_2 : \tau$ *then for any* $(\mathcal{V},\Gamma,\tau)$*-context* $\mathcal{C}[\phi]$*,*

$$[\![e_1]\!]^\star = [\![e_2]\!]^\star \text{ implies } [\![\mathcal{C}[e_1]]\!]^\star = [\![\mathcal{C}[e_2]]\!]^\star.$$

Proof. By induction on $\mathcal{C}[\cdot]$. □

Putting this together with the previous corollary gives us the result we want.

Theorem 6.8 (Denotational semantics respects $\approx$). *If* $\mathcal{V};\Gamma \vdash e_1 : \tau$ *and* $\mathcal{V};\Gamma \vdash e_2 : \tau$*, then*

$$[\![e_1]\!]^\star = [\![e_2]\!]^\star \Rightarrow e_1 \approx e_2.$$

Proof. By Compositionality we know that for all $(\mathcal{V},\Gamma,\tau)$-contexts $\mathcal{C}[\phi]$ it is the case that $[\![\mathcal{C}[e_1]]\!]^\star = [\![\mathcal{C}[e_2]]\!]^\star$. Hence by Adequacy, $\vdash \mathcal{C}[e_1] \Downarrow \langle r, \mathbf{1}\rangle$ if and only if $\vdash \mathcal{C}[e_2] \Downarrow \langle r, \mathbf{1}\rangle$, that is, $e_1 \approx e_2$. □

6.4 Dimensional invariance

Consider an expression e which has typing $\mathcal{V}_{\text{base}};\Gamma_{\text{base}} \vdash e : \textsf{real}\ \mathbf{1}$. For concreteness, assume that $\mathcal{V}_{\text{base}} = \{\mathrm{M},\mathrm{L},\mathrm{T}\}$ and $\Gamma_{\text{base}} = \{kg : \textsf{real}\ \mathrm{M},\ m : \textsf{real}\ \mathrm{L},\ s : \textsf{real}\ \mathrm{T}\}$. Thus e is almost a closed expression except that it makes use of a predetermined system of base dimensions and their associated units of measure.

Now consider some value environment E_{base} which matches this type assignment so that $\mathcal{V}_{\text{base}} \vdash E_{\text{base}} : \Gamma_{\text{base}}$. Then what values should we choose for E_{base}? We could pick $E_{\text{base}} = \{kg : \langle 1,\mathrm{M}\rangle, m : \langle 1,\mathrm{L}\rangle, s : \langle 1,\mathrm{T}\rangle\}$, but it is clear that it should not matter what values we pick. In a sense we are choosing the *units* in which mass, length and time are measured, and the result of evaluating e will not be affected by this choice of units.

There is a proviso: the values chosen for E_{base} must be positive. This is due to the presence of the comparison construct $e_1 < e_2$. The expression e could take different courses of action depending on the signs of the values in E_{base}. Intuitively, it makes no sense for units of measure to be negative or zero, and so a change of unit should only ever alter a value's magnitude.

More generally, consider scaling an arbitrary environment E in accordance with the dimensions of its values to give a new environment E'. If an expression e is first evaluated under E to produce a result v, and then evaluated under E' to produce a result v', then we expect v to scale in accordance with its dimensions to give v'. This is the essence of *dimensional invariance*. In what follows, we formalise the idea of dimensional invariance for the denotational semantics, applying scaling to environments ρ and values w. Then in the next chapter we show how it can be used to prove certain equivalences which hold in the operational semantics.

Scaling environment

A *scaling environment* is a mapping $\psi : \mathcal{V} \to \mathbb{R}^+$ which assigns a positive 'scale factor' to each dimension variable in a set $\mathcal{V}$. A scaling environment can be thought of as a change in the system of units used in an expression. For example, if an expression manipulates masses (dimension M) measured in kilograms, and we wish to change to Imperial units of pounds, then ψ would be the mapping $\{\mathrm{M} \mapsto 2.205\}$.

Note carefully the choice of positive reals for the scaling environment in contrast to the rationals used for semantic values. It turns out that this is required to prove type inhabitation results as described later in Section 7.2.

A scaling environment ψ is extended to arbitrary dimension expressions by the following equations which ensure that ψ is a homomorphism between the Abelian group of dimensions and the Abelian group $\langle \mathbb{R}^+, \cdot \rangle$.

$$\begin{aligned} \psi(\mathbf{1}) &= 1 \\ \psi(\delta_1 \cdot \delta_2) &= \psi(\delta_1) \cdot \psi(\delta_2) \\ \psi(\delta^{-1}) &= 1/\psi(\delta) \end{aligned}$$

Then if we wanted to convert metres to feet as well as changing kilograms to pounds, we would set ψ to be the mapping $\{\mathrm{M} \mapsto 2.205, \mathrm{L} \mapsto 3.281\}$. A conversion of density from kilograms per cubic metre to pounds per cubic foot would be given by $\psi(\mathrm{M} \cdot \mathrm{L}^3)$.

The action of a substitution S on a scaling environment ψ is defined by

$$S(\psi)(d) = \psi(S(d)),$$

and it is easily checked that for any dimension expression δ,

$$S(\psi)(\delta)) = \psi(S(\delta)).$$

The notation ψ^{-1} denotes the pointwise inverse of ψ: a scaling environment defined by $\psi^{-1}(d) \stackrel{\text{def}}{=} 1/\psi(d)$ for each dimension variable d.

Scaling relation

For an ML-like language with implicit polymorphism it is possible to define a scaling *function* on values of any type τ according to some scaling environment ψ—in fact, we do exactly this in Section 7.1 in the next chapter. But if explicit polymorphism is present, as in our language Λ_δ, it is not clear how such a function would operate on values of type $\forall d.\tau$. Instead, we define the stronger notion of a *relation* between values of the same type so that a value w is related to another value w' if w *scales* to give w'. Then a value w of polymorphic dimension type $\forall d.\tau$ is related to another value w' if the values are related for all possible scalings with respect to the dimension variable d. Think of the quantifier in $\forall d.\tau$ as standing for "for all units of measure δ".

The relation $\mathcal{R}^\psi_\tau$ is parameterised on a scaling environment ψ and defined by induction on a type τ, as shown in Figure 6.4. It is extended pointwise to

$$
\begin{array}{rcl}
\mathcal{R}^{\psi}_{\tau} & \subseteq & [\![\tau]\!]^{\star} \times [\![\tau]\!]^{\star} \\
\mathcal{R}^{\psi}_{\mathsf{bool}}(b, b') & \iff & b = b' \\
\mathcal{R}^{\psi}_{\mathsf{real}\ \delta}(r, r') & \iff & \psi(\delta) \cdot r = r' \\
\mathcal{R}^{\psi}_{\tau_1 \to \tau_2}(f, f') & \iff & \text{for all } w, w' \in [\![\tau_1]\!]^{\star},\ \mathcal{R}^{\psi}_{\tau_1}(w, w') \Rightarrow \mathcal{R}^{\psi}_{\tau_{2\perp}}(f(w), f'(w')) \\
\mathcal{R}^{\psi}_{\forall d.\tau}(w, w') & \iff & \text{for all } k \in \mathbb{R}^{+},\ \mathcal{R}^{\psi[d \mapsto k]}_{\tau}(w, w') \\
\\
\mathcal{R}^{\psi}_{\Gamma} & \subseteq & [\![\Gamma]\!]^{\star} \times [\![\Gamma]\!]^{\star} \\
\mathcal{R}^{\psi}_{\Gamma}(\rho, \rho') & \iff & \text{for all } x \in \mathrm{dom}(\Gamma),\ \mathcal{R}^{\psi}_{\Gamma(x)}(\rho(x), \rho'(x))
\end{array}
$$

Figure 6.4: Scaling relation

type assignments to give a relation $\mathcal{R}^{\psi}_{\Gamma}$. Note that $\mathcal{R}^{\psi}_{\tau_{\perp}}$ is shorthand for $(\mathcal{R}^{\psi}_{\tau})_{\perp}$. Informally, the clauses of the definition have the following interpretation.

- Basic values are related if one scales to give the other under the scaling environment ψ. As r and r' are both elements of $\mathbb{Q}$, whereas the scaling environment maps dimensions to $\mathbb{R}^{+}$, then the values will be related only if they are both zero or if $\psi(\delta)$ is rational. In Section 7.2 it is shown how this property is vital to the proof of type inhabitation results.
- The rule for function types can be summed up by the slogan “related arguments give related results”; this makes $\mathcal{R}^{\psi}_{\tau}$ another example of a *logical relation* [44]. In particular notice that if a function fails to terminate on some argument w, then it must fail to terminate on any related argument w'.
- Values of a polymorphic dimension type $\forall d.\tau$ are related if they are related for all possible scalings with respect to the dimension variable d as explained above.

The following lemma characterises the effect on $\mathcal{R}^{\psi}_{\tau}$ of applying a substitution to the type τ or scaling environment ψ.

Lemma 6.9. *For any scaling environment ψ, substitution S and type τ,*

$$\mathcal{R}^{\psi}_{S(\tau)}(w, w') \iff \mathcal{R}^{S(\psi)}_{\tau}(w, w').$$

Proof. Induction on the structure of τ. □

A similar result holds for environments. Another lemma asserts that the relation preserves least upper bounds of chains.

Lemma 6.10. *If $\{w_i\}_{i \in \mathbb{N}}$ and $\{w'_i\}_{i \in \mathbb{N}}$ are chains in $[\![\tau]\!]^{\star}$ such that $\mathcal{R}^{\psi}_{\tau}(w_i, w'_i)$ for all $i \in \mathbb{N}$, then $\mathcal{R}^{\psi}_{\tau}\left(\bigsqcup_{i \in \mathbb{N}} w_i, \bigsqcup_{i \in \mathbb{N}} w'_i\right)$.*

Proof. By induction on τ. □

The dimensional invariance theorem

The remainder of this chapter consists of a proof of the following theorem, the gist of which is that the interpretation of some expression e under an environment ρ is related to the interpretation of the same expression e under an environment ρ' if the environments are related in the same way. The proof follows the same lines as the so-called Fundamental Theorem of Logical Relations [44]; in particular, the cases for application, abstraction and recursion are standard. In addition we have dimension abstraction and application to consider. The remaining cases essentially prove that the built-in constants and arithmetic operations preserve the relation. We present the cases for division and comparison as illustrative of the rest.

Theorem 6.11 (Dimensional invariance). *If $\mathcal{V};\Gamma \vdash e : \tau$ then for any two environments $\rho, \rho' \in [\![\Gamma]\!]^\star$ and scaling environment $\psi : \mathcal{V} \to \mathbb{R}^+$ such that $\mathcal{R}^\psi_\Gamma(\rho, \rho')$ it is the case that*

$$\mathcal{R}^\psi_{\tau_\perp}([\![e]\!]^\star(\rho), [\![e]\!]^\star(\rho')).$$

Proof. By induction on the structure of e.

- For a variable x the result follows immediately from $\Gamma(x) = \tau$.

- For a non-zero constant we have the trivial derivation

$$\frac{}{\mathcal{V};\Gamma \vdash r : \mathsf{real}\ \mathbf{1}}\ (\text{const})$$

  Now $\psi(\mathbf{1}) = 1$ from definition. Hence $r = \psi(\mathbf{1}) \cdot r$ and the result follows immediately.

- For a constant zero we have the trivial derivation

$$\frac{}{\mathcal{V};\Gamma \vdash 0_\delta : \mathsf{real}\ \delta}\ (\text{zero})$$

  and so $0 = \psi(\delta) \cdot 0$ for *any* δ as required.

- For a lambda abstraction we have the derivation

$$\frac{\mathcal{V};\Gamma[x : \tau_1] \vdash e : \tau_2}{\mathcal{V};\Gamma \vdash \lambda x \colon \tau_1.\, e : \tau_1 \to \tau_2}\ (\text{abs}).$$

  Now the result we want is that under the assumption that $\mathcal{R}^\psi_\Gamma(\rho, \rho')$ it is the case that

$$\begin{aligned}
& \mathcal{R}^\psi_{(\tau_1\to\tau_2)_\perp}([\![\lambda x \colon \tau_1.\, e]\!]^\star(\rho), [\![\lambda x \colon \tau_1.\, e]\!]^\star(\rho')) \\
\iff\ & \mathcal{R}^\psi_{(\tau_1\to\tau_2)_\perp}\left(\lfloor \lambda w \in [\![\tau_1]\!]^\star.\, [\![e]\!]^\star(\rho[x \mapsto w]) \rfloor, \lfloor \lambda w \in [\![\tau_1]\!]^\star.\, [\![e]\!]^\star(\rho'[x \mapsto w]) \rfloor\right) \\
\iff\ & \mathcal{R}^\psi_{\tau_1\to\tau_2}\left(\lambda w \in [\![\tau_1]\!]^\star.\, [\![e]\!]^\star(\rho[x \mapsto w]), \lambda w \in [\![\tau_1]\!]^\star.\, [\![e]\!]^\star(\rho'[x \mapsto w])\right).
\end{aligned}$$

From the definition of $\mathcal{R}^{\psi}_{\tau_1 \to \tau_2}$ this says that

$$\forall w, w' \in [\![\tau_1]\!]^\star,\ \mathcal{R}^{\psi}_{\tau_1}(w, w') \Rightarrow \mathcal{R}^{\psi}_{\tau_{2\perp}}([\![e]\!]^\star(\rho[x \mapsto w]), [\![e]\!]^\star(\rho'[x \mapsto w']))$$

and combining this with the original assumption we obtain

$$\mathcal{R}^{\psi}_{\Gamma[x:\tau_1]}(\rho[x \mapsto w], \rho'[x \mapsto w']) \Rightarrow \mathcal{R}^{\psi}_{\tau_{2\perp}}([\![e]\!]^\star(\rho[x \mapsto w]), [\![e]\!]^\star(\rho'[x \mapsto w']))$$

as the conclusion we want to reach. We get there by applying the induction hypothesis to the premise of the (abs) rule.

- For a function application we have the derivation

$$\frac{\mathcal{V}; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \mathcal{V}; \Gamma \vdash e_2 : \tau_1}{\mathcal{V}; \Gamma \vdash e_1\ e_2 : \tau_2}\ (\text{app})$$

By the induction hypothesis we know that

$$\begin{aligned} &\mathcal{R}^{\psi}_{(\tau_1 \to \tau_2)_\perp}([\![e_1]\!]^\star(\rho), [\![e_1]\!]^\star(\rho')) \\ \text{and} \quad &\mathcal{R}^{\psi}_{\tau_{1\perp}}([\![e_2]\!]^\star(\rho), [\![e_2]\!]^\star(\rho')). \end{aligned}$$

We consider two possibilities.

1. $[\![e_1]\!]^\star(\rho) = \perp$ or $[\![e_2]\!]^\star(\rho) = \perp$ which implies by the induction hypothesis that $[\![e_1]\!]^\star(\rho') = \perp$ or $[\![e_2]\!]^\star(\rho') = \perp$. Then $[\![e_1\ e_2]\!]^\star(\rho) = \perp$ and $[\![e_1\ e_2]\!]^\star(\rho') = \perp$ so $\mathcal{R}^{\psi}_{\tau_{2\perp}}([\![e_1\ e_2]\!]^\star(\rho), [\![e_1\ e_2]\!]^\star(\rho'))$ as required.
2. $[\![e_1]\!]^\star(\rho) = [w_1]$ and $[\![e_1]\!]^\star(\rho') = [w_1']$ with $\mathcal{R}^{\psi}_{\tau_1}(w_1, w_1')$, and $[\![e_2]\!]^\star(\rho) = [w_2]$ and $[\![e_2]\!]^\star(\rho') = [w_2']$ with $\mathcal{R}^{\psi}_{\tau_1 \to \tau_2}(w_2, w_2')$. Then by the definition of $\mathcal{R}^{\psi}_{\tau_1 \to \tau_2}$ we have that

$$\forall w, w' \in [\![\tau_1]\!]^\star,\ \mathcal{R}^{\psi}_{\tau_1}(w, w') \Rightarrow \mathcal{R}^{\psi}_{\tau_{2\perp}}(w_2(w), w_2'(w')).$$

Hence $\mathcal{R}^{\psi}_{\tau_{2\perp}}(w_2(w_1), w_2'(w_1'))$ which is $\mathcal{R}^{\psi}_{\tau_{2\perp}}([\![e_1\ e_2]\!]^\star(\rho), [\![e_1\ e_2]\!]^\star(\rho'))$ as required.

- For a rec construct we have the derivation

$$\frac{\mathcal{V}; \Gamma[f : (\tau_1 \to \tau_2), x : \tau_1] \vdash e : \tau_2}{\mathcal{V}; \Gamma \vdash (\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\ e) : \tau_1 \to \tau_2}\ (\text{rec})$$

Now from Figure 6.2,

$$\begin{aligned} [\![\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\ e]\!]^\star(\rho) &= \bigsqcup_{i \in \mathbb{N}} w_i \\ \text{where } w_0 &= \lambda w \in [\![\tau_1]\!]^\star.\perp, \\ \text{and } w_{i+1} &= \lambda w \in [\![\tau_1]\!]^\star.[\![e]\!]^\star(\rho[x \mapsto w, y \mapsto w_i]). \end{aligned}$$

Similarly,

$$[\![\mathsf{rec}\ y(x\colon \tau_1)\colon \tau_2.\ e]\!]^\star(\rho') = \bigsqcup_{i\in\mathbb{N}} w'_i$$

$$\text{where } w'_0 = \lambda w \in [\![\tau_1]\!]^\star.\bot,$$
$$\text{and } w'_{i+1} = \lambda w \in [\![\tau_1]\!]^\star.[\![e]\!]^\star(\rho'[x \mapsto w, y \mapsto w'_i]).$$

We now show by induction that $\mathcal{R}^\psi_{\tau_1\to\tau_2}(w_i, w'_i)$ for all $i \in \mathbb{N}$.

- The base case is trivial: $\mathcal{R}^\psi_{\tau_1\to\tau_2}(\lambda w \in [\![\tau_1]\!]^\star.\bot, \lambda w \in [\![\tau_1]\!]^\star.\bot)$.
- For the step, show that $\mathcal{R}^\psi_{\tau_1\to\tau_2}(w_i, w'_i)$ implies $\mathcal{R}^\psi_{\tau_1\to\tau_2}(w_{i+1}, w'_{i+1})$. Expanding out w_{i+1} and w'_{i+1} gives

  $$\begin{aligned}\mathcal{R}^\psi_{\tau_1\to\tau_2}(&\lambda w \in [\![\tau_1]\!]^\star.[\![e]\!]^\star(\rho[x \mapsto w, y \mapsto w_i]),\\ &\lambda w \in [\![\tau_1]\!]^\star.[\![e]\!]^\star(\rho'[x \mapsto w, y \mapsto w'_i]))\end{aligned}$$

  which from the definition of the relation requires that for all $w, w' \in [\![\tau_1]\!]^\star$ such that $\mathcal{R}^\psi_{\tau_1}(w, w')$ it is the case that

  $$\begin{aligned}\mathcal{R}^\psi_{\tau_{2\bot}}(&[\![e]\!]^\star(\rho[x \mapsto w, y \mapsto w_i]),\\ &[\![e]\!]^\star(\rho'[x \mapsto w', y \mapsto w'_i])).\end{aligned}$$

  We can obtain this by applying the induction hypothesis of the main theorem to the premise of the rule for rec.

Finally we just use Lemma 6.10 to get

$$\mathcal{R}^\psi_{\tau_1\to\tau_2}\left(\bigsqcup_{i\in\mathbb{N}} w_i, \bigsqcup_{i\in\mathbb{N}} w'_i\right)$$

as required.

- For a dimension abstraction we have the derivation

$$\frac{\mathcal{V} \cup \{d\}; \Gamma \vdash e : \tau}{\mathcal{V}; \Gamma \vdash \Lambda d.e : \forall d.\tau}\ (\text{dgen})$$

Consider the scaling environment $\psi : \mathcal{V} \to \mathbb{R}^+$ and value environments ρ, ρ' such that $\mathcal{R}^\psi_\Gamma(\rho, \rho')$. By the side-condition on (dgen) we know that d is not free in Γ and hence $\mathcal{R}^{\psi[d\mapsto k]}_\Gamma(\rho, \rho')$ for any $k \in \mathbb{R}^+$. Hence by the induction hypothesis we can deduce that

$$\mathcal{R}^{\psi[d\mapsto k]}_{\tau_\bot}([\![e]\!]^\star(\rho), [\![e]\!]^\star(\rho')).$$

Under our initial assumptions about ρ, ρ' we know that the above is true for any $k \in \mathbb{R}^+$. Hence from the definition of $\mathcal{R}^\psi_{\forall d.\tau}$ we obtain

$$\mathcal{R}^\psi_{\forall d.\tau_\bot}([\![\Lambda d.e]\!]^\star(\rho), [\![\Lambda d.e]\!]^\star(\rho'))$$

as required.

- For a dimension application we have the derivation

$$\frac{\mathcal{V};\Gamma \vdash e : \forall d.\tau}{\mathcal{V};\Gamma \vdash e\,\delta : \{d \mapsto \delta\}\tau}\ \text{(dspec)}$$

By the induction hypothesis we know that for any $\psi : \mathcal{V} \to \mathbb{R}^+$,

$$\begin{array}{ll} & \mathcal{R}^{\psi}_{\forall d.\tau_\perp}([\![e]\!]^\star(\rho), [\![e]\!]^\star(\rho')) \\ \iff & \forall k \in \mathbb{R}^+,\ \mathcal{R}^{\psi[d\mapsto k]}_{\tau_\perp}([\![e]\!]^\star(\rho), [\![e]\!]^\star(\rho')). \end{array}$$

Now let $S = \{d \mapsto \delta\}$ and let $k = \psi(S(\delta))$. Then by Lemma 6.9 we know that

$$\mathcal{R}^{\psi}_{\{d\mapsto\delta\}\tau_\perp}([\![e\,\delta]\!]^\star(\rho), [\![e\,\delta]\!]^\star(\rho'))$$

as required.

- For a division we have the derivation

$$\frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\,\delta_1 \qquad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\,\delta_2}{\mathcal{V};\Gamma \vdash e_1/e_2 : \mathsf{real}\,\delta_1 \cdot \delta_2^{-1}}\ \text{(div)}$$

From the induction hypothesis we know that

$$\mathcal{R}^{\psi}_{\mathsf{real}\,\delta_1{}_\perp}([\![e_1]\!]^\star(\rho), [\![e_1]\!]^\star(\rho')) \text{ and } \mathcal{R}^{\psi}_{\mathsf{real}\,\delta_2{}_\perp}([\![e_2]\!]^\star(\rho), [\![e_2]\!]^\star(\rho')).$$

Now consider three possibilities.

1. $[\![e_1]\!]^\star(\rho) = \perp$ or $[\![e_2]\!]^\star(\rho) = \perp$. Then $[\![e_1]\!]^\star(\rho') = \perp$ or $[\![e_2]\!]^\star(\rho') = \perp$ from the induction hypothesis, so $[\![e_1/e_2]\!]^\star(\rho) = [\![e_1/e_2]\!]^\star(\rho') = \perp$ and $\mathcal{R}^{\psi}_{(\mathsf{real}\,\delta_1\cdot\delta_2^{-1})_\perp}([\![e_1/e_2]\!]^\star(\rho), [\![e_1/e_2]\!]^\star(\rho'))$ as required.
2. $[\![e_2]\!]^\star(\rho) = [0]$ so by the induction hypothesis we know that $[\![e_2]\!]^\star(\rho') = [\psi(\delta_2)\cdot 0] = [0]$. Then $[\![e2/e1]\!]^\star(\rho) = [\![e2/e1]\!]^\star(\rho') = \perp$ and the result follows.
3. $[\![e_1]\!]^\star(\rho) = [r_1]$ and $[\![e_2]\!]^\star(\rho) = [r_2]$ with $r_2 \neq 0$. Then

$$[\![e_1/e_2]\!]^\star(\rho) \quad = \quad [r_1/r_2]$$

and using the induction hypothesis,

$$\begin{array}{rcl} [\![e_1/e_2]\!]^\star(\rho') & = & [(\psi(\delta_1)\cdot r_1)/(\psi(\delta_2)\cdot r_2)] \\ & = & [(\psi(\delta_1)/\psi(\delta_2))\cdot(r_1/r_2)] \\ & = & \left[\psi(\delta_1\cdot\delta_2^{-1})\cdot(r_1/r_2)\right]. \end{array}$$

Putting these together gives

$$\mathcal{R}^{\psi}_{(\mathsf{real}\,\delta_1\cdot\delta_2^{-1})_\perp}([\![e_1/e_2]\!]^\star(\rho), [\![e_1/e_2]\!]^\star(\rho'))$$

which is the required result.

- For a comparison we have the derivation

$$\frac{\mathcal{V};\Gamma \vdash e_1 : \mathsf{real}\,\delta \qquad \mathcal{V};\Gamma \vdash e_2 : \mathsf{real}\,\delta}{\mathcal{V};\Gamma \vdash e_1 < e_2 : \mathsf{bool}} \text{(cond)}$$

  From the induction hypothesis we know that

$$\mathcal{R}^{\psi}_{\mathsf{real}\,\delta_\perp}([\![e_1]\!]^\star(\rho), [\![e_1]\!]^\star(\rho')) \text{ and } \mathcal{R}^{\psi}_{\mathsf{real}\,\delta_\perp}([\![e_2]\!]^\star(\rho), [\![e_2]\!]^\star(\rho')).$$

  Now consider three possibilities.

  1. $[\![e_1]\!]^\star(\rho) = \perp$ or $[\![e_2]\!]^\star(\rho) = \perp$ so the result follows as in the case for division.
  2. $[\![e_1]\!]^\star(\rho) = [r_1]$ and $[\![e_2]\!]^\star(\rho) = [r_2]$ with $r_1 < r_2$ so $[\![e_1 < e_2]\!]^\star(\rho) = [\mathsf{true}]$. Then from the induction hypothesis $[\![e_1]\!]^\star(\rho') = [\psi(\delta) \cdot r_1]$ and $[\![e_1]\!]^\star(\rho') = [\psi(\delta) \cdot r_2]$. But since $\psi(\delta) > 0$ we must have $\psi(\delta) \cdot r_1 < \psi(\delta) \cdot r_2$ so $[\![e_1 < e_2]\!]^\star(\rho') = [\mathsf{true}]$ as required.
  3. $[\![e_1]\!]^\star(\rho) = [r_1]$ and $[\![e_2]\!]^\star(\rho) = [r_2]$ with $r_1 \geqslant r_2$. This is similar to the previous case.

- The case for conditionals is similar to that for function application.

□

Chapter 7

Dimensional invariance

The dimensional invariance theorem proved at the end of Chapter 6 showed how a logical relation $\mathcal{R}^{\psi}_{\tau}$ can be constructed over the denotational semantics to capture the idea of invariance under scaling. In this chapter we apply dimensional invariance to investigate three properties of programs in Λ_δ.

- First we show that the type of an expression alone determines how it behaves with respect to *scaling*. For instance, any expression with type $\forall d.\mathsf{real}\, d \to \mathsf{real}\, d^2$ has the property that if its argument is multiplied by a scale factor $k \in \mathbb{Q}^+$ then its result is multiplied by k^2.
- An interesting consequence of dimensional invariance is that there are some types which are not inhabited by any non-trivial term. For example, the only expressions with polymorphic type $\forall d.\mathsf{real}\, d^2 \to \mathsf{real}\, d$ are useless ones such as the constant zero function and non-terminating function. The implication is that if we want to calculate square roots then an appropriate function would have type $\forall d.\mathsf{real}\, d \to \mathsf{real}\, d^2 \to \mathsf{real}\, d$, requiring an initial estimate for the root as its first argument.
- The central theorem of dimensional analysis states that any equation which holds between dimensioned variables $x_1, \ldots, x_n$ is equivalent to another equation between a smaller number of dimensionless variables. Each of these is a simple product of powers of the original variables, *i.e.*, having the form $x_1^{z_1} \cdots x_n^{z_n}$ where the exponents z_i are integers. The theorem is founded on the assumption that equations which express physical laws are dimensionally invariant. We describe some preliminary ideas on how an analogous 'Pi theorem' might hold for programming languages with dimension types such as Λ_δ. This would be more general than the classical result as it would hold for higher-order types, and we present an example function to illustrate this.

Finally we argue that the scaling relation can be used to define a better, more abstract semantics which distinguishes fewer observationally-equivalent expressions.

The base type $\mathsf{real}\delta$ is modelled by rationals in our semantics, but surprisingly it is necessary to pick scale factors in ψ from the reals in order to obtain type

$$
\begin{array}{rcl}
\Phi^{\psi}_{\tau} & : & [\![\tau]\!]^{\star} \to [\![\tau]\!]^{\star} \\
\Phi^{\psi}_{\mathsf{bool}}(b) & = & b \\
\Phi^{\psi}_{\mathsf{real}\,\delta}(r) & = & \psi(\delta) \cdot r \\
\Phi^{\psi}_{\tau_1 \to \tau_2}(f) & = & \lambda w \in [\![\tau_1]\!]^{\star}.\ \Phi^{\psi}_{\tau_{2\perp}}(f(\Phi^{\psi^{-1}}_{\tau_1}(w)))
\end{array}
$$

Figure 7.1: Scaling function

inhabitation properties such as the square root example above. We use r to range over all rationals (the elements of base type in the semantics), and k to range over positive rationals or positive reals (the scale factors used in the scaling relation). For the reader uncomfortable with our use of a set as large as the reals, it suffices to take any ordered field which includes all roots of positive elements.

7.1 Theorems for free

We began Chapter 6 with some examples of program equivalences in ML and ML_{δ} which can be deduced purely from the *type* of the program. It is possible to use the scaling relation $\mathcal{R}^{\psi}_{\tau}$ directly to obtain these 'theorems for free'. However, for ML-style types in which all quantifiers are outermost it is easier to make use of a scaling *function* which performs an invertible 'change of units of measure' on values which have quantifier-free types. For example, consider an expression which calculates the force on a body due to gravity. Its type would be

$$\mathit{weight} : \mathsf{real}\ \mathrm{M} \to \mathsf{real}\ \mathrm{M} \cdot \mathrm{L} \cdot \mathrm{T}^{-2}.$$

Now assume that this function is initially used in the metric system but we wish to convert it for use under Imperial units. A suitable conversion is given by

$$\mathit{weightimp} \stackrel{\mathrm{def}}{=} \lambda \mathit{mass} \colon \mathsf{real}\ \mathrm{M}.\ \mathit{weight}(\mathit{mass}/k_1) * k_2$$

where $k_1 = \psi(\mathrm{M})$, $k_2 = \psi(\mathrm{L} \cdot \mathrm{T}^{-2})$ and $\psi = \{\mathrm{M} \mapsto 2.205,\ \mathrm{L} \mapsto 3.281,\ \mathrm{T} \mapsto 1\}$, given that there are 2.205 pounds in a kilogram and 3.281 feet in a metre.

In general, the arguments to a function are scaled one way and the result scaled the opposite way; this works for higher-order functions as well. Formally, we define a family of scaling functions $\Phi^{\psi}_{\tau} : [\![\tau]\!]^{\star} \to [\![\tau]\!]^{\star}$ parameterised on a scaling environment $\psi : \mathit{DimVars} \to \mathbb{Q}^{+}$ and defined inductively on the type τ. This is shown in Figure 7.1. Because the basic values in $[\![\tau]\!]^{\star}$ are rationals, here we restrict the scaling environment to positive rational scale factors in contrast to the more general scaling relation which used reals. It is extended to environments in the obvious pointwise fashion, giving functions $\Phi^{\psi}_{\Gamma} : [\![\Gamma]\!]^{\star} \to [\![\Gamma]\!]^{\star}$.

In defining Φ^{ψ}_{τ} we are essentially picking out the functional subset of the scaling relation $\mathcal{R}^{\psi}_{\tau}$ when τ is quantifier-free. This is proved formally by the following lemma.

Lemma 7.1. *For any scaling environment $\psi : \mathit{DimVars} \to \mathbb{Q}^+$ and quantifier-free type τ,*

$$\mathcal{R}^{\psi}_{\tau}(w, w') \iff \Phi^{\psi}_{\tau}(w) = w'.$$

Proof. By induction on τ. □

Putting this together with the dimensional invariance theorem we get the following result for expressions with quantifier-free types.

Proposition 7.2. *If $\mathcal{V}; \Gamma \vdash e : \tau$ for some quantifier-free type τ and quantifier-free type assignment Γ, then for any scaling environment $\psi : \mathcal{V} \to \mathbb{Q}^+$ and value environment $\rho \in [\![\Gamma]\!]^\star$,*

$$[\![e]\!]^\star(\Phi^{\psi}_{\Gamma}(\rho)) = \Phi^{\psi}_{\tau_\bot}([\![e]\!]^\star(\rho)).$$

Summarised as a commuting diagram this is the following:

$$\begin{array}{ccc} [\![\Gamma]\!]^\star & \xrightarrow{[\![e]\!]^\star} & [\![\tau]\!]^\star_\bot \\ \Phi^{\psi}_{\Gamma}\downarrow & & \downarrow\Phi^{\psi}_{\tau_\bot} \\ [\![\Gamma]\!]^\star & \xrightarrow[{[\![e]\!]^\star}]{} & [\![\tau]\!]^\star_\bot \end{array}$$

It is interesting to note that the functions Φ^{ψ}_{τ} and $\Phi^{\psi^{-1}}_{\tau}$ are very similar to the coercions used by Leroy to optimise data representation in ML [38]. In place of our scaling conversions he has `wrap` and `unwrap` coercions which are used to convert data between boxed and unboxed forms when polymorphic functions are used at particular types.

We shall now see how Proposition 7.2 can be used to derive observational equivalences between expressions of the kind that Wadler calls 'theorems for free' [66]. We give two examples informally; the formal reasoning involved is tedious but trivial. For conciseness we omit type information from the expressions, as would be the case in ML.

Example (Exponentiation). Consider an expression with an 'exponentiation' type:

$$\vdash e : \forall d.\mathsf{real}\ d \to \mathsf{real}\ d^n, \quad n \in \mathbb{Z}.$$

Then for any $k \in \mathbb{Q}^+$ the following equivalence holds:

$$e(k * x) \ \approx\ k^n * e(x).$$

Example (Differentiation). Consider an expression *diff* with the type of the differentiation function from page 10 in Chapter 1:

$$\vdash \mathit{diff} : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2) \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \cdot d_1^{-1}).$$

Then for any $k_1, k_2 \in \mathbb{Q}^+$ the following equivalence holds:

$$\mathit{diff}\ h\ f\ x \ \approx\ \frac{k_2}{k_1} * \mathit{diff}\left(\frac{h}{k_1}\right)\left(\lambda x.\frac{f(x * k_1)}{k_2}\right)\left(\frac{x}{k_1}\right).$$

The scaling relation and scaling function can be extended to deal with data structures such as tuples, lists, or arbitrary algebraic datatypes, if the language Λ_δ were so extended. For example, tuples scale in a pointwise fashion, and likewise the scaling of a list is simply the scaling of each of its elements (all of which have the same dimension type).

Example (Variance). Consider an expression *variance* with the type of the variance function on page 9 of Chapter 1:

$$\vdash \mathit{variance} : \forall d.\mathsf{real}\ d\ \mathsf{list} \rightarrow \mathsf{real}\ d^2.$$

Then for any $k \in \mathbb{Q}^+$ the following equivalence holds:

$$\mathit{variance}(\mathit{map}\ (\lambda x.k * x)\ \mathit{xs})\ \approx\ k^2 * \mathit{variance}(\mathit{xs})$$

where $\mathit{map}\ f\ \mathit{xs}$ applies the function f to every element of the list xs.

It is unsurprising but very pleasing to note that these kinds of properties carry over correctly to the implementation of complex numbers defined using only primitive operations on the reals. Consider implementing a complex number package in which the complex number type has a dimension parameter, for example via the Standard ML parameterisation mechanism used in Chapter 4. Complex numbers might be implemented as pairs with real and imaginary components:

$$\mathsf{complex}\,\delta\ \stackrel{\text{def}}{=}\ \mathsf{real}\ \delta \times \mathsf{real}\ \delta.$$

Alternatively, a complex number could be represented in polar form by its magnitude and polar angle:

$$\mathsf{complex}\,\delta\ \stackrel{\text{def}}{=}\ \mathsf{real}\ \delta \times \mathsf{real}\ \mathbf{1}.$$

Either way, the usual arithmetic operations can be defined on complex numbers and given polymorphic dimension types matching those of the same operations on the reals—and the inference algorithm of Chapter 3 can deduce them. Scaling properties derived by dimensional invariance apply to dimensioned complex numbers: for the first representation multiplying both real and imaginary components by the same positive scale factor, and for the second representation multiplying just the magnitude. Of course, the properties could be broken if the representation was revealed to the undisciplined programmer; therefore the type and its primitive operations should be packaged up as an abstract data type if the language supports it.

7.2 Type inhabitation

Consider the typing

$$\vdash e : \forall d.\mathsf{real}\ d^2 \to \mathsf{real}\ d.$$

Assume that the evaluation of e does terminate so that $[\![e]\!]^\star(\emptyset) = [f]$ for some $f \in \mathbb{Q} \to \mathbb{Q}_\perp$. Now consider the behaviour of this function when applied to a rational number r. We know that either $f(r) = \perp$ or $f(r) = [r']$ for some rational number r' (f cannot be a true square root function because some rationals have irrational roots).

Now apply the dimensional invariance theorem, first considering only rational scale factors. Then for any $k \in \mathbb{Q}^+$, if $f(r) = \perp$ then $f(k^2 \cdot r) = \perp$, and if $f(r) = [r']$ then $f(k^2 \cdot r) = [k \cdot r']$. This cuts down the range of possible functions somewhat. If f fails to terminate on *any* value without a rational root then it must fail to terminate on all such values. Also, if it returns zero for any value without a rational root then it must return zero for all such values. So we have the intriguing possibility of a function which finds rational roots when they exist, but which otherwise returns zero or just loops. Indeed, it is possible to write such a function simply by enumerating all rationals until reaching the root but looping if none exists. Informally, an implementation of this function can not have a polymorphic type because the only rationals that it can enumerate must be dimensionless.

In order to show formally that there is no such rational square root function with a polymorphic type, we must allow the scale factor k to be irrational. For any $k' \in \mathbb{Q}^+$ there is some $k \in \mathbb{R}^+$ such that $k' = k^2$. Applying dimensional invariance gives two possibilities.

- If $f(r) = \perp$ then $f(k' \cdot r) = \perp$. Hence if f fails to terminate for *any* positive argument, then it must fail to terminate for all positive arguments. Similarly, if it loops for any negative argument then it loops for all negative arguments.
- If $f(r) = [r']$ then $f(k' \cdot r) = [k \cdot r']$. Now because $f \in \mathbb{Q} \to \mathbb{Q}_\perp$ it must be the case that $k \cdot r'$ is rational. This can only be true if $r' = 0$. Hence if f returns zero for any positive argument then it must return zero for all positive arguments. The same is true of negative arguments.

To sum up, we have drawn the scale factors from the positive reals in order to show that a function with the type $\forall d.\mathsf{real}\ d^2 \to \mathsf{real}\ d$ cannot distinguish between different positive or negative values. In fact, all functions f of this type are characterised by

$$f(r) = \begin{cases} w_1 & \text{if } r < 0, \\ w_2 & \text{if } r = 0, \\ w_3 & \text{if } r > 0, \end{cases}$$

where each of w_1, w_2 and w_3 is either $\perp$ or $[0]$. This gives just eight possible functions.

More generally, consider an expression e with the typing

$$\vdash e : \forall d.\mathsf{real}\, d^m \to \mathsf{real}\, d^n$$

for some $m, n \in \mathbb{Z}$. Let $[\![e]\!]^\star(\emptyset) = [f]$. Then f must be one of the trivial functions described above whenever m does not divide n.

These examples show how it is not possible to write polymorphic root-finding functions unless an initial estimate of the root is provided as an additional argument. For example, it *is* possible to write an approximate square root function with the type $\forall d.\mathsf{real}\, d \to \mathsf{real}\, d^2 \to \mathsf{real}\, d$. A real implementation might provide a primitive square root operation with type $\forall d.\mathsf{real}\, d^2 \to \mathsf{real}\, d$, but it would *not* be dimensionally invariant, at least not precisely. In an implementation based on floating-point arithmetic even the primitive addition and subtraction operations cannot be dimensionally invariant due to the non-linearity of floating-point representation, so we would not really lose anything by making this compromise.

It is interesting to note that one *can* write a dimensionally-polymorphic function which accepts two numbers a and b and returns an approximation to $\sqrt{a^2 + b^2}$. This function has the type $\forall d.\mathsf{real}\, d \to \mathsf{real}\, d \to \mathsf{real}\, d$, and would use some linear combination of a and b as an initial estimate for the root.

For our final example we formalise the use of the phrase *non-trivial*.

- A value $r \in [\![\mathsf{real}\, \delta]\!]^\star$ is non-trivial if $r \neq 0$.
- A value $f \in [\![\tau_1 \to \tau_2]\!]^\star$ is non-trivial if $f(w)$ is non-trivial for some $w \in [\![\tau_1]\!]^\star$.
- A value $w \in [\![\tau]\!]^\star_\perp$ is non-trivial if $w = [w']$ for some non-trivial $w' \in [\![\tau]\!]^\star$.

We now present an example of a type which is non-trivially inhabited if and only if there is a solution to a certain equation in integers.

Example. The expression e has the typing[†]

$$\vdash e : \forall d.\mathsf{real}\, d^{a_1} \to \cdots \to \mathsf{real}\, d^{a_n} \to \mathsf{real}\, d^b$$

for $a_1, \ldots, a_n, b \in \mathbb{Z}$. Then $[\![e]\!]^\star$ can be non-trivial only if there is a solution in integers $c_1, \ldots, c_n$ to the equation

$$a_1 c_1 + \cdots + a_n c_n = b.$$

Proof. Let $[\![e]\!]^\star(\emptyset) = [w]$. By dimensional invariance we know that

$$\mathcal{R}^{\emptyset}_{\forall d.\mathsf{real}\, d^{a_1} \to \cdots \to \mathsf{real}\, d^{a_n} \to \mathsf{real}\, d^b}(w, w).$$

Expanding out the definition of the relation gives us the following: for all $k \in \mathbb{R}^+$ and all $r_1, \ldots, r_n \in \mathbb{Q}$ such that $k^{a_1} \cdot r_1, \ldots, k^{a_n} \cdot r_n \in \mathbb{Q}$ it is the case that

$$\mathcal{R}^{\{d \mapsto k\}}_{\mathsf{real}\, d^b{}_\perp}(w\ (r_1)\ \cdots\ (r_n), w\ (k^{a_1} \cdot r_1) \cdots (k^{a_n} \cdot r_n)).$$

[†]This example suggested by Ian Stark

Now suppose that there are some $r_1, \dots, r_n \in \mathbb{Q}$ such that

$$w\ (r_1) \cdots\ (r_n) = [r]$$

for some $r \neq 0$. Let $g = \gcd(a_1, \dots, a_n)$ and pick $k \in \mathbb{R}^+$ such that $k^g \in \mathbb{Q}^+$ but $k^m \notin \mathbb{Q}^+$ for all $m < g$ (for example, let $k = 2^{1/g}$). Then clearly $k^{a_1}, \dots, k^{a_n} \in \mathbb{Q}^+$. So by the result above we must have

$$w\ (k^{a_1} \cdot r_1)\ \cdots\ (k^{a_n} \cdot r_n) = \left[k^b \cdot r\right].$$

But $k^b \cdot r \in \mathbb{Q}$ only if g divides b or $r = 0$, so w can be non-trivial only if $\gcd(a_1, \dots, a_n)$ divides b. This statement is equivalent to there being a solution in integers $c_1, \dots, c_n$ to the equation

$$a_1 c_1 + \cdots + a_n c_n = b.$$

□

7.3 Dimensional analysis and type isomorphisms

We start this section with a statement of the Pi Theorem from dimensional analysis.

Theorem (Pi Theorem). *Fix a set of n base dimensions. Let $x_1, \dots, x_n$ be positive variables with the dimension of x_i given by the i'th column of an $m \times n$ matrix A of dimension exponents. Then any dimensionally-invariant relation of the form*

$$f(x_1, \dots, x_n) = 0$$

is equivalent to a relation

$$f'(\Pi_1, \dots, \Pi_{n-r}) = 0$$

where r is the rank of the matrix A and $\Pi_1, \dots, \Pi_{n-r}$ are dimensionless power-products of $x_1, \dots, x_n$.

Proof. See Birkhoff [6]. □

The proof of the theorem constructs f' explicitly. We now do the same for a particular instance: the equation which governs the behaviour of a simple pendulum as discussed in Chapter 1.

Example (Pendulum). Suppose that there is some function f which relates the mass m, the length of pendulum l, the initial angle from the vertical θ, the acceleration due to gravity g and the period of oscillation t:

$$f(m, l, \theta, g, t) = 0.$$

Now m has dimensions [M], l has dimensions [L], θ is dimensionless, g has dimensions [LT^{-2}] and t has dimensions [T]. Then by dimensional invariance, for any positive M, L and T the following relation holds:

$$f(Mm, Ll, \theta, LT^{-2}g, Tt) = 0.$$

Assuming that the variables m, l and g are positive, let

$$M = 1/m,$$
$$L = 1/l,$$
$$\text{and} \quad T = \sqrt{g/l}.$$

Then the relation can be rewritten

$$f(1, 1, \theta, 1, t\sqrt{g/l}) = 0$$

so removing the constant arguments this is a function $f'(\theta, t\sqrt{g/l})$ whose arguments are dimensionless power-products. We can rearrange this to obtain a function ϕ such that the period of oscillation is given by

$$t = \sqrt{l/g}\,\phi(\theta).$$

Inspired by this example, suppose that we have some expression e in Λ_δ with

$$\vdash e : \forall \mathrm{M}.\forall \mathrm{L}.\forall \mathrm{T}.\mathsf{real}\ \mathrm{M} \to \mathsf{real}\ \mathrm{L} \to \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathrm{L}\cdot\mathrm{T}^{-2} \to \mathsf{real}\ \mathrm{T} \to \mathsf{real}\ \mathbf{1}.$$

Then by using an argument similar to the example (based on our dimensional invariance theorem) we can deduce that the following equivalence holds for positive argument values. Again for simplicity we omit type information:

$$e\ (m)\ (l)\ (\theta)\ (g)\ (t) \approx 0 \ \text{ if and only if } \ e\ (1)\ (1)\ (\theta)\ (1)\ (t * \sqrt{g/l}) \approx 0.$$

So we can construct an expression e' analogous to f' above with the typing

$$\vdash e' : \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}$$

such that

$$e\ (m)\ (l)\ (\theta)\ (g)\ (t) \approx 0 \ \text{ if and only if } \ e'\ (\theta)\ (t * \sqrt{g/l}) \approx 0.$$

Note, however, that this *assumes* the existence of a square root function, which we have already shown cannot be defined in the language. Thus the relation above is rather limited in its application. More definitive results are the subject of further research.

The classical Pi Theorem covers only first-order functions, but the idea works equally well for higher-order functions in Λ_δ. This suggests a more general result which applies to expressions of *any* type, though it is not clear how types with nested quantifiers would be handled.

Example (Differentiation). Consider the differentiation function again:

$$\vdash \mathit{diff} : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2) \to (\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \cdot d_1^{-1}).$$

In Section 7.1 we showed that the following equivalence holds for any $k_1, k_2 \in \mathbb{Q}^+$. This time we show all dimensions explicitly:

$$\mathit{diff}\ \delta_1\ \delta_2\ h\ f\ x \;\approx\; \frac{k_2}{k_1} * \mathit{diff}\ \delta_1\ \delta_2\ \left(\frac{h}{k_1}\right) \left(\lambda x\colon \mathsf{real}\ \delta_1. \frac{f(x * k_1)}{k_2}\right) \left(\frac{x}{k_1}\right).$$

Assume that x, h and f are positive. Then we can set $k_1 = h$ and $k_2 = f(h)$ to obtain

$$\mathit{diff}\ \delta_1\ \delta_2\ h\ f\ x \;\approx\; \frac{f(h)}{h} * \mathit{diff}' \left(\lambda x\colon \mathsf{real}\ \mathbf{1}. \frac{f(x * h)}{f(h)}\right) \left(\frac{x}{h}\right)$$

where $\mathit{diff}' = \mathit{diff}\ \mathbf{1}\ \mathbf{1}\ 1$ and has typing

$$\vdash \mathit{diff}' : (\mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}) \to (\mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}).$$

These reductions are *almost* examples of *type isomorphisms*. We say that a type τ_1 is *isomorphic* to another type τ_2 (written $\tau_1 \cong \tau_2$) if there are expressions e_1 and e_2 with types $\tau_1 \to \tau_2$ and $\tau_2 \to \tau_1$ such that $e_1(e_2 x) \approx x$ and $e_2(e_1 x) \approx x$. In other words, there exist two expressions which together define an isomorphism between the domains of τ_1 and τ_2. For example,

$$\tau_1 \to \tau_2 \to \tau_3 \;\cong\; \tau_2 \to \tau_1 \to \tau_3$$

by the expressions

$$\begin{aligned} e_1 &\stackrel{\text{def}}{=} \lambda z\colon \tau_1 \to \tau_2 \to \tau_3.\ \lambda x\colon \tau_2.\ \lambda y\colon \tau_1.\ z\ y\ x \\ \text{and} \quad e_2 &\stackrel{\text{def}}{=} \lambda z\colon \tau_2 \to \tau_1 \to \tau_3.\ \lambda x\colon \tau_1.\ \lambda y\colon \tau_2.\ z\ y\ x. \end{aligned}$$

Now consider the type $\tau_1 = \forall d.\mathsf{real}\ d \to \mathsf{real}\ d \to \mathsf{real}\ d$. By the Pi Theorem it would first appear that any expression of type τ_1 is 'equivalent' in some sense to an expression of type $\tau_2 = \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}$. Therefore in addition to standard isomorphisms between types (studied for example by Di Cosmo [11]), one would hope to add

$$\forall d.\mathsf{real}\ d \to \mathsf{real}\ d \to \mathsf{real}\ d \;\cong\; \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}.$$

Suitable coercions are given by

$$\begin{aligned} e_1 &\stackrel{\text{def}}{=} \lambda z\colon (\forall d \mathsf{real}\ d \to \mathsf{real}\ d \to \mathsf{real}\ d).\ \lambda x\colon \mathsf{real}\ \mathbf{1}.\ z\ \mathbf{1}\ 1\ x \\ \text{and} \quad e_2 &\stackrel{\text{def}}{=} \lambda z\colon (\mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}).\ \Lambda d.\ \lambda x\colon \mathsf{real}\ d.\ \lambda y\colon \mathsf{real}\ d.\ (z\ (y/x)) * x. \end{aligned}$$

However, as we saw earlier this only works if argument values are restricted to be positive. Intuitively there are more functions of type $\forall d.\mathsf{real}\ d \to \mathsf{real}\ d \to \mathsf{real}\ d$ than of type $\mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}$ because they can use the *signs* of the two arguments to determine completely different courses of action, in contrast to functions of

type $\mathsf{real}\,\mathbf{1} \to \mathsf{real}\,\mathbf{1}$ which can only switch on the sign of a single argument. It follows that we do not have a true isomorphism.

One kind of type isomorphism which does exist is associated with a change of basis in the sense described in Chapter 2. Our earlier notion of type scheme equivalence ($\cong_D$ defined on page 27) is then just one example of a type isomorphism ($\cong$ defined above):

$$\forall d_1 \ldots d_m.\tau \cong_D \forall d'_1 \ldots d'_n.\tau' \text{ implies } \forall d_1 \ldots d_m.\tau \cong \forall d'_1 \ldots d'_n.\tau'.$$

If the type scheme equivalence holds then by Lemma 3.10 there is an invertible substitution U such that $U(\tau) =_D \tau'$ and $U^{-1}(\tau') =_D \tau$. Using this substitution we can define expressions e and e' which perform the appropriate coercions between expressions of equivalent polymorphic types. First let $\delta_1 = U(d_1), \ldots, \delta_m = U(d_m)$ and $\delta'_1 = U^{-1}(d'_1), \ldots, \delta'_n = U^{-1}(d'_n)$. Then

$$\begin{aligned} e &\stackrel{\text{def}}{=} \lambda z \colon (\forall d_1 \ldots d_m.\tau).\, \Lambda d'_1 \ldots d'_n.\, z\, \delta_1 \cdots \delta_m \\ \text{and } e' &\stackrel{\text{def}}{=} \lambda z \colon (\forall d'_1 \ldots d'_n.\tau').\, \Lambda d_1 \ldots d_m.\, z\, \delta'_1 \cdots \delta'_n. \end{aligned}$$

7.4 A more abstract model

Through the use of the dimensional invariance theorem we have shown how certain elements of a ‘dimensionless’ domain $[\![\tau]\!]^\star$ can be ruled out as possible meanings for expressions with the type τ. In this section we ‘put the dimensions back into the semantics’ by using the scaling relation to quotient the semantics in an appropriate way. The resulting semantics is more abstract and distinguishes less expressions.

Reasoning using dimensional invariance

Consider the following observationally-equivalent expressions seen earlier in Section 6.3:

$$\begin{aligned} e_1 &\stackrel{\text{def}}{=} \lambda y \colon (\forall d.\mathsf{real}\, d \to \mathsf{real}\, d).\, y\, \mathbf{1}\, 2 \\ \text{and } e_2 &\stackrel{\text{def}}{=} \lambda y \colon (\forall d.\mathsf{real}\, d \to \mathsf{real}\, d).\, 2 * y\, \mathbf{1}\, 1. \end{aligned}$$

Although these expressions have different denotations in the naive, dimensionless semantics given by $[\![\cdot]\!]^\star$, we can use the scaling relation and dimensional invariance theorem to prove that they are equivalent. First, though, we need a lemma which shows that observational equivalence can be characterised by something simpler than all program contexts. For functional languages Milner proved a result which he called the Context Lemma [41]. It states that two expressions of closed, functional type can only be distinguished observationally by applying them to some argument value. We can prove a similar result for our language Λ_δ and extend it to cover expressions of polymorphic type.

Lemma 7.3 (Context Lemma). *If* $\vdash e_1 : \tau_1 \to \tau_2$ *and* $\vdash e_2 : \tau_1 \to \tau_2$ *then* $e_1 \approx e_2$ *if and only if* $e_1\, e \approx e_2\, e$ *for all expressions* e *of type* τ_1*. Also, if* $\vdash e_1 : \forall d.\tau$ *and* $\vdash e_2 : \forall d.\tau$ *then* $e_1 \approx e_2$ *if and only if* $e_1\, \delta \approx e_2\, \delta$ *for all dimension expressions* δ*.*

Applying this to the expressions e_1 and e_2 given above, we need to show that for any expression e of type $\forall d.\mathsf{real}\ d \to \mathsf{real}\ d$ it is the case that $e_1\, e \approx e_2\, e$. Now by dimensional invariance, if $\vdash e : \forall d.\mathsf{real}\ d \to \mathsf{real}\ d$ and $[\![e]\!]^\star(\emptyset) = w$ then $\mathcal{R}^{\emptyset}_{\forall d.\mathsf{real}\ d \to \mathsf{real}\ d}(w, w)$. Also,

$$
\begin{aligned}
[\![e_1\, e]\!]^\star(\emptyset) &= [\![y\ \mathbf{1}\ 2]\!]^\star\{y \mapsto w\}\\
[\![e_2\, e]\!]^\star(\emptyset) &= [\![2 * y\ \mathbf{1}\ 1]\!]^\star\{y \mapsto w\}.
\end{aligned}
$$

Expanding out the definition of the scaling relation it follows that $[\![e_1\, e]\!]^\star(\emptyset) = [\![e_2\, e]\!]^\star(\emptyset)$ and hence by adequacy $e_1\, e \approx e_2\, e$ as required.

Quotienting the semantics

Inspired by the use of the scaling relation in the reasoning above, we now discuss some preliminary work on constructing a more abstract semantics for dimension types.

Consider two expressions e_1, e_2 with typings $\mathcal{V}; \Gamma \vdash e_1 : \tau$ and $\mathcal{V}; \Gamma \vdash e_2 : \tau$. In the new semantics we want to assign identical meanings to e_1 and e_2 if their meanings $[\![e_1]\!]^\star$ and $[\![e_2]\!]^\star$ in the original semantics preserve the scaling relation for *any* choice of scaling environment $\psi \in \mathcal{V} \to \mathbb{R}^+$. Let $\mathcal{R}^{\mathcal{V};\Gamma}_\tau$ be a scaling relation for functions in $[\![\Gamma]\!]^\star \to [\![\tau]\!]^\star_\perp$ (where $\mathrm{fv}(\tau) \cup \mathrm{fv}(\Gamma) \subseteq \mathcal{V}$) defined by:

$$
\mathcal{R}^{\mathcal{V};\Gamma}_\tau(f, f') \quad \stackrel{\text{def}}{\iff} \quad \begin{array}{l} \text{for all } \psi \in \mathcal{V} \to \mathbb{R}^+ \text{ and } \rho, \rho' \in [\![\Gamma]\!]^\star, \\ \mathcal{R}^\psi_\Gamma(\rho, \rho') \Rightarrow \mathcal{R}^\psi_{\tau_\perp}(f(\rho), f'(\rho')). \end{array}
$$

Using this new notation, dimensional invariance (Theorem 6.11) can be restated as follows: if $\mathcal{V}; \Gamma \vdash e : \tau$ then $\mathcal{R}^{\mathcal{V};\Gamma}_\tau([\![e]\!]^\star, [\![e]\!]^\star)$.

To define the new semantics we quotient the original semantics by a *partial equivalence relation.* A PER is a binary relation R that is transitive and symmetric. Its *domain* is defined by $\mathrm{dom}(R) = \{\ x \mid (x, x) \in R\ \}$. A PER is so-called because it forms a proper equivalence relation over its domain, but outside of its domain no elements are in the relation. Ideally we would like $\mathcal{R}^{\mathcal{V};\Gamma}_\tau$ to be a PER, so that values in the new semantics are simply equivalence classes of values in the old semantics under $\mathcal{R}^{\mathcal{V};\Gamma}_\tau$. It is certainly true that $\mathcal{R}^{\mathcal{V};\Gamma}_\tau$ is symmetric.

Lemma 7.4. *If* $\mathcal{R}^{\mathcal{V};\Gamma}_\tau(f, f')$ *then* $\mathcal{R}^{\mathcal{V};\Gamma}_\tau(f', f)$*.*

Proof. For a scaling environment $\psi \in \mathcal{V} \to \mathbb{R}^+$ define ψ^{-1} by:

$$
\psi^{-1}(d) \stackrel{\text{def}}{=} 1/\psi(d).
$$

It is clear that $\psi^{-1}(\delta) = 1/\psi(\delta)$ for any dimension δ. It is also easy to show by induction that

$$
\mathcal{R}^\psi_\tau(w, w') \text{ if and only if } \mathcal{R}^{\psi^{-1}}_\tau(w', w)
$$

and

$$\mathcal{R}^{\psi}_{\Gamma}(\rho, \rho') \text{ if and only if } \mathcal{R}^{\psi^{-1}}_{\Gamma}(\rho', \rho).$$

The symmetry of $\mathcal{R}^{\mathcal{V};\Gamma}_{\tau}$ is a direct corollary. □

Unfortunately transitivity is much harder to show. The obvious approach is to start by defining the pointwise product of two scaling environments ψ_1 and ψ_2:

$$(\psi_1 \cdot \psi_2)(d) \stackrel{\text{def}}{=} \psi_1(d) \cdot \psi_2(d).$$

It is clear that $(\psi_1 \cdot \psi_2)(\delta) = \psi_1(\delta) \cdot \psi_2(\delta)$ for any dimension δ. We might then expect that

$$\text{if } \mathcal{R}^{\psi_1}_{\tau}(w, w') \text{ and } \mathcal{R}^{\psi_2}_{\tau}(w', w'') \text{ then } \mathcal{R}^{\psi_1 \cdot \psi_2}_{\tau}(w, w'').$$

Unfortunately this does not hold. A counterexample is given by $\tau = \mathsf{real}\, d \to \mathsf{real}\, \mathbf{1}$, $\psi_1 = \psi_2 = \{d \mapsto \sqrt{2}\}$, and $w = w' = w'' = \lambda r.r \in \mathbb{Q} \to \mathbb{Q}_\perp$. So far other attempts to prove transitivity have failed. To avoid the issue, we can just take the transitive closure of the relation, denoted by $tr(\mathcal{R}^{\mathcal{V};\Gamma}_{\tau})$. The practical import of this technique is that in order to prove that two expressions are equivalent it might be necessary to relate their meanings by $\mathcal{R}^{\mathcal{V};\Gamma}_{\tau}$ via some intermediate values.

The equivalence class of a function $f \in [\![\Gamma]\!]^\star \to [\![\tau]\!]^\star_\perp$ (where $\mathrm{fv}(\tau) \cup \mathrm{fv}(\Gamma) \subseteq \mathcal{V}$) is denoted $[f]^{\mathcal{V};\Gamma}_{\tau}$ and defined as follows:

$$[f]^{\mathcal{V};\Gamma}_{\tau} \stackrel{\text{def}}{=} \left\{ f' \in [\![\Gamma]\!]^\star \to [\![\tau]\!]^\star_\perp \;\middle|\; tr(\mathcal{R}^{\mathcal{V};\Gamma}_{\tau})(f, f') \right\}.$$

Then the abstract semantics of an expression e with typing $\mathcal{V}; \Gamma \vdash e : \tau$ is given by

$$[\![e]\!] \stackrel{\text{def}}{=} [[\![e]\!]^\star]^{\mathcal{V};\Gamma}_{\tau}.$$

Hence the meaning $[\![e]\!]$ is an equivalence class in the quotient

$$([\![\Gamma]\!]^\star \to [\![\tau]\!]^\star_\perp)/tr(\mathcal{R}^{\mathcal{V};\Gamma}_{\tau}).$$

How abstract is the semantics?

The observational equivalence proved manually above shows that the new semantics based on dimensional invariance is more abstract than the dimensionless semantics which provides the underlying domains. But just how abstract is it? First, it is worth observing that the decision to quantify scale factors over the reals rather than the rationals accounts for an additional level of abstraction. Consider the following expressions:

$$e_1 \stackrel{\text{def}}{=} \lambda y \colon (\forall d.\mathsf{real}\, d^2 \to \mathsf{real}\, d).\, y\, \mathbf{1}\, 2$$

$$\text{and } e_2 \stackrel{\text{def}}{=} \lambda y \colon (\forall d.\mathsf{real}\, d^2 \to \mathsf{real}\, d).\, y\, \mathbf{1}\, 4$$

It is clear from the discussion in Section 7.2 that if the scaling relation was allowed to quantify only over the rationals then there would be rational square root functions related by $\mathcal{R}^{\emptyset}_{\tau}$ for $\tau = \forall d.\mathsf{real}\ d^2 \to \mathsf{real}\ d$. These could be used to distinguish e_1 and e_2. With scale factors drawn from the reals, though, we have shown that the functions preserved by the scaling relation cannot even distinguish between different arguments which have the same sign. Hence in our more abstract semantics e_1 and e_2 are indistinguishable: $[\![e_1]\!] = [\![e_2]\!]$.

The new semantics could be said to be *parametric*, in the sense that it captures the parametric polymorphism of the language. What is the *best* we can expect of this parametric semantics? It cannot be fully abstract, because its underlying model is just the standard cpo model of PCF which is known not to be fully abstract. Ideally we want something that is 'no worse'; that is, the only distinctions that the semantics makes are those that would have been made anyway in models of PCF. To formalise this, let the proposition $\mathit{dist}(e_1, e_2)$ hold if e_1 and e_2 are incorrectly distinguished by the denotational semantics, that is,

$$\mathit{dist}(e_1, e_2) \stackrel{\text{def}}{=} [\![e_1]\!] \neq [\![e_2]\!] \wedge e_1 \approx e_2.$$

Then we say that the semantics is *relatively fully abstract* if for any expressions e_1 and e_2 such that $\mathcal{V}; \Gamma \vdash e_1 : \tau$ and $\mathcal{V}; \Gamma \vdash e_2 : \tau$, then

$$\mathit{dist}(e_1, e_2) \Rightarrow \mathit{dist}({e_1}^{\star}, {e_2}^{\star}).$$

Informally, this says that if two expressions e_1 and e_2 are distinguished incorrectly by the denotational semantics, then they would also be distinguished even if dimensions were ignored. Thus the only expressions which are distinguished incorrectly are the sort that standard models of PCF would distinguish, *i.e.* those which are distinguished by the presence of a parallel-or function in the model.

Another way of saying the same thing is that *if* the underlying semantics $[\![\cdot]\!]^{\star}$ were fully abstract with respect to dimensionless expressions, *then* the parametric semantics $[\![\cdot]\!]$ would be fully abstract also. This is analogous to the idea that a program logic can be *relatively complete* with respect to its underlying logic of assertions.

I do not know whether the new semantics is relatively fully abstract: this is an area for future research.

Chapter 8

Conclusion

In this final chapter we describe some possibilities for further research into the syntax and semantics of dimension types and related type systems. We then summarise the results of the dissertation.

8.1 Further work

Types and type inference

The type system described in Chapter 2 and its inference algorithm presented in Chapter 3 were based on the conventional notion of *type scheme*, first introduced by Damas and Milner [13]. However, the subtleties of working in the non-regular equational theory of Abelian groups led to some quite unconventional problems and their solutions: the distinction between essential and non-essential free variables, and the need for an invertible substitution which was applied before generalising a type with respect to a type assignment. One possible avenue of research is to abandon type schemes and define an equivalent *first-order* system. This is the style favoured by Henglein [23] and by Kfoury *et al.* [31] in their study of polymorphic recursion for ML. In their formulation, the context in a typing judgment consists of a type assignment containing only simple types (no type schemes) and a list of types which have been assigned to lambda-bound variables. Interestingly, this is very reminiscent of the original system proposed by Milner [42].

It is possible that a type inference algorithm based on a first-order formulation of the type system would be more efficient than the current one based on type schemes. Indeed, efficiency of dimension inference is an area not explored in this thesis. The worst-case complexity of dimension inference cannot be any worse than ordinary ML type inference, given that Abelian group unification can be done in polynomial-time. However, it is probable that for the average case dimension inference is worse, and this may be worth investigating.

Many possible enhancements to dimension types were discussed at the end of Chapter 3. Of these, polymorphic recursion has received some attention through Rittri's work [57, 58] but this is by no means complete. Odersky and Läufer have recently shown how one can extend the ML type system with explicit annotations

that give the language the expressiveness of System F. It would be interesting to investigate supporting dimension types in such a system.

More generally there is potential for studying ML-style type inference under an arbitrary equational theory. Rémy has done this for regular theories [55] but for non-regular theories such as that of dimensions the problem is open. The obvious conjecture is that unique most general types exist whenever unification in the equational theory is unitary.

The overloading of arithmetic operators was the only real difficulty encountered when implementing a dimension type extension to Standard ML. There seem to be two opposing trends in the treatment of overloading. One is to abandon it entirely: this appears to be the direction that ML is taking (for example in ML-2000). The other is to generalise the idea of equality types from Standard ML to a system of *type classes*: these are the most innovative aspect of the language Haskell [25, 67]. A type class declaration prescribes certain operations which must be available on any type belonging to that class. For example, the class *Eq* of equality types requires that for every type τ in the class there is an operation '=' with the type $\tau \to \tau \to \mathsf{bool}$. Furthermore, there is a programmer-defined ordering relation on type classes which supports the idea of *inheritance*. A class C is a *subclass* of another class C' if it provides at least the operations of C'. A very natural extension of this idea would be the definition of a type class (called *Dim*, say) whose type instances admit a dimension parameter. In other words, *Dim* is a class of type operators. Any type belonging to this class must provide the standard arithmetic operations, perhaps including a comparison operator. This idea is along the same lines as Chen, Hudak and Odersky's *parametric type classes* [10], where types belonging to a type class may be parameterised on another *type*; this is particularly useful in representing container classes such as lists and bags. Here is some tentative syntax for the definition of *Dim*:

$$
\begin{array}{l}
\mathsf{class}\ Dim\ T = \\
\{ \\
\quad \mathtt{+,-} \quad : \forall d.T(d) \to T(d) \to T(d) \\
\quad \mathtt{*} \quad : \forall d_1.\forall d_2.T(d_1) \to T(d_2) \to T(d_1 \cdot d_2) \\
\quad \mathtt{/} \quad : \forall d_1.\forall d_2.T(d_1) \to T(d_2) \to T(d_1 \cdot d_2^{-1}) \\
\quad \mathtt{<} \quad : \forall d.T(d) \to T(d) \to \mathsf{bool} \\
\}
\end{array}
$$

The type of rational values (perhaps $\mathsf{frac}\ \delta$) would be an instance of *Dim*. A subclass of *Dim* would add extra operations such as sin and log; the type $\mathsf{real}\ \delta$ of reals and the type $\mathsf{complex}\ \delta$ of complex numbers would be instances of this class.

This discussion hints at a whole hierarchy of type classes whose instances are dimensioned types. An obvious question is: can integers have dimensions? After all, many quantities clearly have units (for instance, disc storage density in bytes per sector) but would not usually be represented by real numbers. However, consider the 'div' operation on integers, defined by

$$x \,\mathsf{div}\, y = \lfloor x/y \rfloor.$$

This could be given the polymorphic dimension type

$$\mathsf{div} : \forall d_1.\forall d_2.\mathsf{int}\, d_1 \to \mathsf{int}\, d_2 \to \mathsf{int}\, d_1 \cdot d_2^{-1}.$$

But this type does not correctly describe the behaviour of div under scaling of its arguments: it is not dimensionally invariant. Therefore if we want a proper semantics for dimensioned integers, and a dimensional invariance result in the sense described in Chapter 6, then we are forced to reject this type for div and allow only dimensionless arguments. We are left with just addition, subtraction and multiplication as dimensioned arithmetic operations, none of which uses dimension inverses. So perhaps the correct equational theory for dimensions in this case is not that of Abelian groups, but of commutative monoids instead, with axioms just for associativity, commutativity and identity of dimensions.

This thesis has concentrated on dimension-checking and dimension-inference for programming languages and has not covered the situation of multiple systems of units within the same dimension. Different units can be seen as different *views*, or implementations, of the same underlying abstract data type. Thatte has shown how a type inference algorithm can be modified to insert coercions between different views automatically [62]. He calls these maps *coercive type isomorphisms*, but because in his system they are defined by the programmer there is no guarantee that they are true isomorphisms. For the special case of unit conversions there *is* such a guarantee whose most general form is the dimensional invariance theorem described in Chapter 6. If units are taken as primary, and $\mathsf{real}\,\delta$ interpreted as "reals with unit of measure δ", then it should be straightforward to extend the typing rules of ML_δ with information concerning unit conversions, and then modify the dimension unification and inference algorithms to insert unit conversions automatically. The best way to formalise this would be to combine it with the translation into the explicitly-typed language Λ_δ.

As well as multiple units, programmers might want to use different sets of base dimensions, for example to use the dimension *force* as an alternative to the combination of SI base dimensions $[\mathrm{MLT}^{-2}]$. At its simplest, this just requires a *dimension synonym* construct in the style of the type synonym construct provided by Standard ML. Then a synonym for force would be declared by the following:

```
dimension Force = M * L * T ^ ~2
```

The problem of a unique form to display polymorphic dimension types was addressed in Section 3.3. The same issue crops up here: if there are a number of ways to describe a dimension type in terms of various base dimensions and synonyms, what is the 'best' way to present such a type to the programmer? For example, pressure (force per unit area) has the SI dimensions $[\mathrm{ML}^{-1}\mathrm{T}^{-2}]$ but is more simply expressed as $[\mathrm{Force}{\cdot}\mathrm{L}^{-2}]$. Of course, this issue of type representation occurs with ordinary type synonyms too.

More imaginatively, the possibility of multiple units and alternative sets of base dimensions could be integrated into the module system of a language, as

Goubault has suggested for ML [18]. In Standard ML, types may be *shared* between modules if special declarations called *sharing constraints* make clear which types are to be treated as the same. Goubault describes how an analogous dimension sharing construct could specify the relationship between alternate sets of base dimensions, and at the same time define unit conversions between different units used for these base dimensions. For example, one module might use the SI base dimensions of mass, length and time, measured in kilograms, metres and seconds, whereas another uses force, velocity and acceleration measured in pounds, feet per second and feet per second per second.

Semantics

There is scope for further work in understanding the denotational semantics of dimension types. In particular, it would be very satisfying to obtain a *relatively* fully abstract model, in the sense that all distinctions made by the semantics are due to the underlying cpo-based model. In essence the idea is to separate the issues of sequentiality and parametricity in the language. The former is a well-known hard problem (the 'full abstraction problem for PCF') and the partial equivalence relation described in Section 7.4 is a step towards solving the latter. Although this work is not complete, there are reasons to be optimistic. This is because dimension polymorphism is *predicative*: the quantification in a dimension type $\forall d.\tau$ is over something 'smaller' than $\forall d.\tau$. In contrast, parametric models of System F are much harder to find because type polymorphism is *impredicative*: the quantification in a type $\forall t.\tau$ is over all *types*, and this includes the type $\forall t.\tau$ itself.

In Chapter 7 we presented two examples of types which are inhabited up to equivalence only by a small number of 'trivial' terms. It should be possible to obtain a better characterisation of what is meant by 'trivial' and of those types which are non-trivially inhabited. There may be some kind of Curry-Howard correspondence between propositions and dimension types: that is to say, a type is non-trivially inhabited if and only if there is a proof of a certain proposition which corresponds to that type.

We saw informally how the Pi Theorem from dimensional analysis can be applied to first-order types in Λ_δ, and an example was given of a similar result holding for the higher-order type of the differentiation function. One area of future research is the formulation of a Pi Theorem for types of any order; also, the situation for negative and zero arguments to functions should be investigated.

It seems likely that parametricity could be applied fruitfully to other predicative type systems based on an equational theory. For example, Rémy's system for polymorphic typing of records uses an equational theory and predicative quantification over *rows*. There are also potential applications in providing a better understanding of some type-based program analyses. For example, Tofte's analysis of memory allocation in functional programs [65] employs quantification over *regions*; again, this is predicative. In general, we have some equational theory E which induces an equivalence between types $=_E$. Then a logical relation $\mathcal{R}_\tau$ is defined over the domains of values in a denotational semantics, in the style

of the scaling relation of Chapter 6. We certainly want this relation to preserve the equivalence, that is:

$$\tau_1 =_E \tau_2 \;\Rightarrow\; \mathcal{R}_{\tau_1} = \mathcal{R}_{\tau_2}.$$

For the dimension type system, the equivalence relation on types was built up from an equivalence on the dimension components of base types, that is $\mathsf{real}\,\delta$. By ensuring that the scaling relation was preserved under this equivalence at base type the rest then came 'for free'. Our scaling environment ψ can be viewed as a homomorphism between elements of the free Abelian group of dimensions and the Abelian group of unit conversions on the values of base type (the rationals). It seems likely that for type systems based on other equational theories we should look to this kind of homomorphic relation for parametricity results.

It is possible to extend the explicitly-typed language Λ_δ to give it a higher-order type system with a notion of *kinds*, in the same way that System F has been extended to produce System F_ω. To see why this might be useful, consider the following ML code:

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

Under ordinary ML type inference, `twice` is assigned the type scheme

$$\forall t.(t \to t) \to (t \to t).$$

Unfortunately, this does not capture all possible uses of the function. In particular, when it is applied to the function `sqr`, which in our system would be assigned the polymorphic dimension type $\forall d.\mathsf{real}\; d \to \mathsf{real}\; d^2$, the resulting function `fourth` can only be assigned a dimensionless type instead of the desired $\forall d.\mathsf{real}\; d \to \mathsf{real}\; d^4$. This is because the argument passed to `twice` is used at two instances: at the type $\forall d.\mathsf{real}\; d \to \mathsf{real}\; d^2$ and at $\forall d.\mathsf{real}\; d^2 \to \mathsf{real}\; d^4$. Crucially, these instances are related to each other by the dimension *operator* $F = \lambda d.d^2$. The first has type $\forall d.\mathsf{real}\; d \to \mathsf{real}\; (F(d))$ and the second has type $\forall d.\mathsf{real}\; (F(d)) \to \mathsf{real}\; (F(F(d)))$. Hence if `twice` was allowed to quantify over dimension *operators* then a suitable typing could be given to it which would permit `twice sqr` to be assigned the type $\forall d.\mathsf{real}\; d \to \mathsf{real}\; d^4$ as desired.

We call the new language with dimension operators $\Lambda_{\delta\omega}$. In addition to the kind of ground dimension expressions, there is a kind of dimension operators (functions from dimensions to dimensions), and of all higher-order operators

too. Kinds, dimensions and types would then have the following syntax:

$$
\begin{array}{rcll}
K & ::= & \mathsf{dim} & \textit{kind of dimensions} \\
 & | & K_1 \rightarrow K_2 & \textit{operator kinds} \\
 \\
\delta & ::= & d & \textit{dimension variables} \\
 & | & \mathbf{1} & \textit{unit dimension} \\
 & | & \delta_1 \cdot \delta_2 & \textit{dimension product} \\
 & | & \delta^{-1} & \textit{dimension inverse} \\
 & | & \lambda d\colon K.\delta & \textit{operator abstraction} \\
 & | & \delta_1\ \delta_2 & \textit{operator application} \\
 \\
\tau & ::= & \mathsf{bool} & \textit{booleans} \\
 & | & \mathsf{real}\ \delta & \textit{dimensioned reals} \\
 & | & \tau_1 \rightarrow \tau_2 & \textit{function types} \\
 & | & \forall d\colon K.\tau & \textit{dimension quantification}
\end{array}
$$

Dimensions now form a very small typed lambda calculus, with application of dimension operators and operator abstraction. Types differ only in that the kind of a quantified dimension variable must be given; similarly, expressions are identical to those of Λ_δ except that the dimension abstraction construct $\Lambda d\colon K.e$ specifies a kind. We will not present the typing rules but the reader familiar with System F_ω should be able to reconstruct them without difficulty. Just observe that the problematic `twice sqr` can be expressed in $\Lambda_{\delta\omega}$:

$$
\begin{array}{rcl}
\textit{twice} & : & \forall F\colon \mathsf{dim} \rightarrow \mathsf{dim}. \\
 & & (\forall d\colon \mathsf{dim}.\ \mathsf{real}\ d \rightarrow \mathsf{real}\ F\ d) \\
 & & (\forall d\colon \mathsf{dim}.\ \mathsf{real}\ d \rightarrow \mathsf{real}\ F(F\ d)) \\
\textit{twice} & \stackrel{\text{def}}{=} & \Lambda F\colon \mathsf{dim} \rightarrow \mathsf{dim}. \\
 & & \lambda f\colon (\forall d\colon \mathsf{dim}.\ \mathsf{real}\ d \rightarrow \mathsf{real}\ F\ d). \\
 & & \Lambda d\colon \mathsf{dim}.\ \lambda x\colon \mathsf{real}\ d.\ f\ (F\ d)\ (f\ d\ x) \\
 \\
\textit{sqr} & : & \forall d\colon \mathsf{dim}.\ \mathsf{real}\ d \rightarrow \mathsf{real}\ d^2 \\
\textit{sqr} & \stackrel{\text{def}}{=} & \Lambda d\colon \mathsf{dim}.\ \lambda x\colon \mathsf{real}\ d.\ x * x \\
 \\
\textit{fourth} & : & \forall d\colon \mathsf{dim}.\ \mathsf{real}\ d \rightarrow \mathsf{real}\ d^4 \\
\textit{fourth} & \stackrel{\text{def}}{=} & \textit{twice}\ (\lambda d\colon \mathsf{dim}.\ d^2)\ \textit{sqr}
\end{array}
$$

This demonstrates the expressiveness of $\Lambda_{\delta\omega}$ but perhaps just for the purposes of dimensions it is a 'type system too far'. However, the ideas are likely be applicable to similar type systems based on equational theories which have some kind of predicative quantification over 'data representations'. Indeed there are similarities between $\Lambda_{\delta\omega}$ and the type system used by Harper and Morrisett to optimize compilation of ML programs [21]. The predicativity of his system stems from the restriction that kinds range over type constructors of *simple types* only.

ML	ML_δ
Class of types τ	Classes of types τ and dimensions δ
Syntactic equivalence $=$	Semantic equivalence $=_D$
Type quantification $\forall t.\tau$	Dimension quantification $\forall d.\tau$ also
Alpha-equivalence of type schemes $\Rightarrow$ obvious canonical form	Non-trivial equivalence of type schemes $\cong_D$ $\Rightarrow$ non-trivial Hermite normal form
Free variables in types and schemes	Essential free variables in types and schemes
Generalisation easy (just $\mathrm{ftv}(\Gamma) \setminus \mathrm{ftv}(\tau)$)	Generalisation hard (change of basis first)
Inference by syntactic unification	Inference by semantic unification over $=_D$
Syntactic unification is unitary $\Rightarrow$ single most general type	Unification over $=_D$ is unitary $\Rightarrow$ single most general type
Well-typed programs don't go wrong $\Rightarrow$ erase types at run-time	Well-dimensioned programs don't go wrong $\Rightarrow$ erase dimensions at run-time
Polymorphic recursion undecidable	Dimension-polymorphic recursion not known

Figure 8.1: Comparison of ML and ML_δ

Finally, the inspiration for all this work on semantics was the well-known fact that physical laws are invariant under changes in the system of units. But in general they are also invariant under changes in the *coordinate system* given by a translation or rotation of the axes. Mathematically, this is the theory of *tensors*. It would make an interesting challenge to devise a type system which supported this idea.

8.2 Summary

In conclusion, we summarise the relationship between conventional type systems and the dimension type systems described here. Figure 8.1 highlights the similarities and differences between ordinary ML and the extension which we called ML_δ. Figure 8.2 does the same for System F and the explicitly-typed language Λ_δ.

In our investigation into programming languages and dimensions we have described the following:

- A generalisation of the Damas-Milner type system which supports dimension types and dimension polymorphism.
- An inference algorithm which combines type inference with dimension inference, deducing a most general type which has a natural canonical representation expressing type polymorphism and dimension polymorphism.

System F	Λ_δ
Impredicative type polymorphism $\forall t.\tau$	Predicative dimension polymorphism $\forall d.\delta$
Type abstraction $\Lambda t.e$	Dimension abstraction $\Lambda d.e$
Type application $e\ \tau$	Dimension application $e\ \delta$
Parametricity	Dimensional invariance
Theorems for free	Scaling theorems for free
Type inhabitation (no term in $\forall t_1.\forall t_2.t_1 \to t_2$)	Type inhabitation (trivial terms in $\forall d.\mathsf{real}\ d^2 \to \mathsf{real}\ d$)
Type isomorphisms	Pi Theorem from dimensional analysis

Figure 8.2: Comparison of System F and Λ_δ

- The practicality of integrating dimensions into a general-purpose programming language: Standard ML.
- The formal operational semantics of a programming language with dimensions which showed that dimension errors cannot occur during the evaluation of a well-typed program, and hence that dimensions do not need to be carried around at run-time.
- The denotational semantics of the same language and its relation to the operational semantics.
- A theorem that captured the essence of polymorphic dimension types—invariance under scaling—and was used to prove interesting observational equivalences between expressions in the language.

This research provides a foundation for the application of dimensions to programming by programming language designers and implementers. Then programmers can enjoy the benefits of the prevention of dimension errors in programs.

Appendix A

Example derivations

In this appendix we present three sample typing derivations. The first two (shown in Figures A.1 and A.2) are for an expression in the language ML_δ of Chapter 2:

$$\lambda y.\mathsf{let}\ sqr = \lambda x.x * x\ \mathsf{in}\ sqr\ 3.14 * sqr\ y$$

This is assigned the type

$$\mathsf{real}\ d \to \mathsf{real}\ d^2.$$

The typing of this expression uses most of the rules. In particular, notice how the built-in multiplication operation and user-defined squaring function are used at different dimensions through the application of the (dspec) rule, and the polymorphism in *sqr* is introduced through a combination of (let) and (dgen). In the syntax-directed derivation, specialisation happens in rule (var′) and generalisation in rule (let′).

The third derivation is of the translation of the program above into the following expression in the explicitly-typed language of Chapter 5:

$$\begin{array}{l}\lambda y\colon \mathsf{real}\ d.\ (\lambda sqr\colon (\forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2).\ sqr\ \mathbf{1}\ 3.14 * sqr\ d\ y) \\ \qquad (\Lambda d_1.\lambda x\colon \mathsf{real}\ d_1.\ x * x)\end{array}$$

Derivation $\mathcal{D}_1$

$$\Gamma_1 = \{* : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2,\ y : \mathsf{real}\ d\}$$

$$\dfrac{}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash * : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2}\ (\mathrm{var})$$

$$\dfrac{\Gamma_1[x : \mathsf{real}\ d_1] \vdash * : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash * : \mathsf{real}\ d_1 \to \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}\ (\mathrm{dspec}^2)
\qquad
\dfrac{}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x : \mathsf{real}\ d_1}\ (\mathrm{var})$$

$$\dfrac{\Gamma_1[x : \mathsf{real}\ d_1] \vdash * : \mathsf{real}\ d_1 \to \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2 \qquad \Gamma_1[x : \mathsf{real}\ d_1] \vdash x : \mathsf{real}\ d_1}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x\, *\ : \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}\ (\mathrm{app})
\qquad
\dfrac{}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x : \mathsf{real}\ d_1}\ (\mathrm{var})$$

$$\dfrac{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x\, *\ : \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2 \qquad \Gamma_1[x : \mathsf{real}\ d_1] \vdash x : \mathsf{real}\ d_1}{\dfrac{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x * x : \mathsf{real}\ d_1^2}{\Gamma_1 \vdash \lambda x.x * x : \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}\ (\mathrm{abs})}\ (\mathrm{app})$$

Derivation $\mathcal{D}_2$

$$\Gamma_2 = \{* : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2,\ y : \mathsf{real}\ d,\ sqr : \forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2\}$$

$$\dfrac{\dfrac{}{\Gamma_2 \vdash * : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2}\ (\mathrm{var})}{\Gamma_2 \vdash * : \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ d^2 \to \mathsf{real}\ d^2}\ (\mathrm{dspec}^2)$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash sqr : \forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}\ (\mathrm{var})}{\Gamma_2 \vdash sqr : \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ \mathbf{1}}\ (\mathrm{dspec}) \qquad \dfrac{}{\Gamma_2 \vdash 3.14 : \mathsf{real}\ \mathbf{1}}\ (\mathrm{const})}{\Gamma_2 \vdash sqr\ 3.14 : \mathsf{real}\ \mathbf{1}}\ (\mathrm{app})$$

$$\dfrac{\Gamma_2 \vdash * : \mathsf{real}\ \mathbf{1} \to \mathsf{real}\ d^2 \to \mathsf{real}\ d^2 \qquad \Gamma_2 \vdash sqr\ 3.14 : \mathsf{real}\ \mathbf{1}}{\Gamma_2 \vdash sqr\ 3.14\, *\ : \mathsf{real}\ d^2 \to \mathsf{real}\ d^2}\ (\mathrm{app})$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash sqr : \forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}\ (\mathrm{var})}{\Gamma_2 \vdash sqr : \mathsf{real}\ d \to \mathsf{real}\ d^2}\ (\mathrm{dspec}) \qquad \dfrac{}{\Gamma_2 \vdash y : \mathsf{real}\ d}\ (\mathrm{var})}{\Gamma_2 \vdash sqr\ y : \mathsf{real}\ d^2}\ (\mathrm{app})$$

$$\dfrac{\Gamma_2 \vdash sqr\ 3.14\, *\ : \mathsf{real}\ d^2 \to \mathsf{real}\ d^2 \qquad \Gamma_2 \vdash sqr\ y : \mathsf{real}\ d^2}{\Gamma_2 \vdash sqr\ 3.14 * sqr\ y : \mathsf{real}\ d^2}\ (\mathrm{app})$$

Final derivation using $\mathcal{D}_1$ and $\mathcal{D}_2$

$$\Gamma = \{* : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \to \mathsf{real}\ d_2 \to \mathsf{real}\ d_1 \cdot d_2\}$$

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\Gamma[y : \mathsf{real}\ d] \vdash \lambda x.x * x : \mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}}{\Gamma[y : \mathsf{real}\ d] \vdash \lambda x.x * x : \forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2}\ (\mathrm{dgen}) \qquad \overset{\mathcal{D}_2}{\Gamma[y : \mathsf{real}\ d, sqr : \forall d_1.\mathsf{real}\ d_1 \to \mathsf{real}\ d_1^2] \vdash sqr\ 3.14 * sqr\ y : \mathsf{real}\ d^2}}{\dfrac{\Gamma[y : \mathsf{real}\ d] \vdash \mathsf{let}\ sqr = \lambda x.x * x\ \mathsf{in}\ sqr\ 3.14 * sqr\ y : \mathsf{real}\ d^2}{\Gamma \vdash \lambda y.\mathsf{let}\ sqr = \lambda x.x * x\ \mathsf{in}\ sqr\ 3.14 * sqr\ y : \mathsf{real}\ d \to \mathsf{real}\ d^2}\ (\mathrm{abs})}\ (\mathrm{let})$$

Figure A.1: A non-syntax-directed derivation in ML_δ

Derivation $\mathcal{D}'_1$

$$\Gamma_1 = \{* : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_2 \rightarrow \mathsf{real}\ d_1 \cdot d_2,\ y : \mathsf{real}\ d\}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash * : \mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1^2}\,(\text{var}') \quad \dfrac{}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x : \mathsf{real}\ d_1}\,(\text{var})}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x * \ : \mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1^2}\,(\text{app}) \quad \dfrac{}{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x : \mathsf{real}\ d_1}\,(\text{var})}{\dfrac{\Gamma_1[x : \mathsf{real}\ d_1] \vdash x * x : \mathsf{real}\ d_1^2}{\Gamma_1 \vdash \lambda x.x * x : \mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1^2}\,(\text{abs})}\,(\text{app})$$

Derivation $\mathcal{D}'_2$

$$\Gamma_2 = \{* : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_2 \rightarrow \mathsf{real}\ d_1 \cdot d_2,\ y : \mathsf{real}\ d,\ sqr : \forall d_1.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1^2\}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash * : \mathsf{real}\ \mathbf{1} \rightarrow \mathsf{real}\ d^2 \rightarrow \mathsf{real}\ d^2}\,(\text{var}') \quad \dfrac{\dfrac{}{\Gamma_2 \vdash sqr : \mathsf{real}\ \mathbf{1} \rightarrow \mathsf{real}\ \mathbf{1}}\,(\text{var}') \quad \dfrac{}{\Gamma_2 \vdash 3.14 : \mathsf{real}\ \mathbf{1}}\,(\text{const})}{\Gamma_2 \vdash sqr\ 3.14 : \mathsf{real}\ \mathbf{1}}\,(\text{app})}{\Gamma_2 \vdash sqr\ 3.14 * \ : \mathsf{real}\ d^2 \rightarrow \mathsf{real}\ d^2}\,(\text{app}) \quad \dfrac{\dfrac{}{\Gamma_2 \vdash sqr : \mathsf{real}\ d \rightarrow \mathsf{real}\ d^2}\,(\text{var}') \quad \dfrac{}{\Gamma_2 \vdash y : \mathsf{real}\ d}\,(\text{var})}{\Gamma_2 \vdash sqr\ y : \mathsf{real}\ d^2}\,(\text{app})}{\Gamma_2 \vdash sqr\ 3.14 * sqr\ y : \mathsf{real}\ d^2}\,(\text{app})$$

Final derivation using $\mathcal{D}'_1$ and $\mathcal{D}'_2$

$$\Gamma = \{* : \forall d_1.\forall d_2.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_2 \rightarrow \mathsf{real}\ d_1 \cdot d_2\}$$

$$\dfrac{\dfrac{\overset{\mathcal{D}'_1}{\Gamma[y : \mathsf{real}\ d] \vdash \lambda x.x * x : \mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1^2} \quad \overset{\mathcal{D}'_2}{\Gamma[y : \mathsf{real}\ d, sqr : \forall d_1.\mathsf{real}\ d_1 \rightarrow \mathsf{real}\ d_1^2] \vdash sqr\ 3.14 * sqr\ y : \mathsf{real}\ d^2}}{\Gamma[y : \mathsf{real}\ d] \vdash \mathsf{let}\ sqr = \lambda x.x * x\ \mathsf{in}\ sqr\ 3.14 * sqr\ y : \mathsf{real}\ d^2}\,(\text{let}')}{\Gamma \vdash \lambda y.\mathsf{let}\ sqr = \lambda x.x * x\ \mathsf{in}\ sqr\ 3.14 * sqr\ y : \mathsf{real}\ d \rightarrow \mathsf{real}\ d^2}\,(\text{abs})$$

Figure A.2: A syntax-directed derivation in ML_δ

Derivation $\mathcal{D}_1''$

$$\dfrac{\dfrac{\dfrac{}{\{d;d_1\};\{y:\mathsf{real}\ d,\ x:\mathsf{real}\ d_1\}\vdash x:\mathsf{real}\ d_1}\,(\mathrm{var}) \qquad \dfrac{}{\{d;d_1\};\{y:\mathsf{real}\ d,\ x:\mathsf{real}\ d_1\}\vdash x:\mathsf{real}\ d_1}\,(\mathrm{var})}{\{d;d_1\};\{y:\mathsf{real}\ d,\ x:\mathsf{real}\ d_1\}\vdash x*x:\mathsf{real}\ d_1^2}\,(\mathrm{mul})}{\{d,d_1\};\{y:\mathsf{real}\ d\}\vdash \lambda x\colon\mathsf{real}\ d_1.\ x*x:\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2}\,(\mathrm{abs})$$

Derivation $\mathcal{D}_2''$

$$\Gamma=\{y:\mathsf{real}\ d,\ sqr:\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{d\};\Gamma\vdash sqr:\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ {d_1}^2}\,(\mathrm{var})}{\{d\};\Gamma\vdash sqr\ \mathbf{1}:\mathsf{real}\ \mathbf{1}\to\mathsf{real}\ \mathbf{1}}\,(\mathrm{dspec}) \qquad \dfrac{}{\{d\};\Gamma\vdash 3.14:\mathsf{real}\ \mathbf{1}}\,(\mathrm{const})}{\{d\};\Gamma\vdash sqr\ \mathbf{1}\ 3.14:\mathsf{real}\ \mathbf{1}}\,(\mathrm{app}) \qquad \dfrac{\dfrac{\dfrac{}{\{d\};\Gamma\vdash sqr:\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ {d_1}^2}\,(\mathrm{var})}{\{d\};\Gamma\vdash sqr\ d:\mathsf{real}\ d\to\mathsf{real}\ d^2}\,(\mathrm{dspec}) \qquad \dfrac{}{\{d\};\Gamma\vdash y:\mathsf{real}\ d}\,(\mathrm{var})}{\{d\};\Gamma\vdash sqr\ d\ y:\mathsf{real}\ d^2}\,(\mathrm{app})}{\{d\};\Gamma\vdash sqr\ \mathbf{1}\ 3.14*sqr\ d\ y:\mathsf{real}\ d^2}\,(\mathrm{mul})$$

Final derivation using $\mathcal{D}_1''$ and $\mathcal{D}_2''$

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_2''\\ \{d\};\{y:\mathsf{real}\ d,sqr:\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2\}\vdash sqr\ \mathbf{1}\ 3.14*sqr\ d\ y:\mathsf{real}\ d^2\end{matrix}}{\{d\};\{y:\mathsf{real}\ d\vdash \lambda sqr\colon\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2.\ sqr\ \mathbf{1}\ 3.14*sqr\ d\ y:(\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2)\to\mathsf{real}\ d^2}\,(\mathrm{abs}) \qquad \dfrac{\begin{matrix}\mathcal{D}_1''\\ \{d,d_1\};\{y:\mathsf{real}\ d\}\vdash\lambda x\colon\mathsf{real}\ d_1.\ x*x:\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2\end{matrix}}{\{d\};\{y:\mathsf{real}\ d\}\vdash\Lambda d_1.\lambda x\colon\mathsf{real}\ d_1.\ x*x:\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2}\,(\mathrm{dgen})}{\dfrac{\{d\};\{y:\mathsf{real}\ d\}\vdash(\lambda sqr\colon\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2.\ sqr\ \mathbf{1}\ 3.14*sqr\ d\ y)\ (\Lambda d_1.\lambda x\colon\mathsf{real}\ d_1.\ x*x):\mathsf{real}\ d^2}{\{d\};\emptyset\vdash\lambda y\colon d.\ (\lambda sqr\colon\forall d_1.\mathsf{real}\ d_1\to\mathsf{real}\ d_1^2.\ sqr\ \mathbf{1}\ 3.14*sqr\ d\ y)\ (\Lambda d_1.\lambda x\colon\mathsf{real}\ d_1.\ x*x):\mathsf{real}\ d\to\mathsf{real}\ d^2}\,(\mathrm{abs})}\,(\mathrm{app})$$

Figure A.3: A derivation in Λ_δ

Appendix B

An algebraic view of dimension types

In Chapters 2 and 3, a dimension type system and its inference algorithm were presented in the syntactic style traditionally used for describing such systems. The presence of equations in the type system made for a certain degree of complexity, including a number of notions of ordering and equivalence. In this short appendix we take a more algebraic approach, treating dimensions and dimension types as elements of vector spaces over the integers, or $\mathbb{Z}$-*modules*. More concretely, if only a finite set of dimension variables is considered, then dimensions can simply be interpreted as vectors of integers, dimension types as rectangular matrices of integers, and substitutions as square matrices.

Most of the results in the main part of the thesis were obtained without the benefit of module theory, but it is likely to illuminate future investigations. The best reference for module theory is the textbook by Adkins and Weintraub [1]; for the theory of integral matrices only, see Newman [46].

For ease of presentation, only the dimension fragment of the language ML_δ of Chapter 2 is considered: type variables and type schemes which quantify over types are omitted from the interpretation. We also do not distinguish dimension variables and base dimensions.

B.1 Module theory

We begin by defining the algebraic notion of a *module* and associated *module homomorphism*. Other module-theoretic concepts will be introduced when required.

Let R be an arbitrary ring with identity. Then a *left R-module*, or *left module over R*, is an abelian group M together with a scalar multiplication operation

$$\cdot : R \times M \to M$$

that satisfies the following four axioms, with x, y standing for elements of M and

λ, μ standing for elements of R.

$$\begin{aligned} &\lambda(x + y) = \lambda x + \lambda y; \\ &(\lambda + \mu)x = \lambda x + \mu y; \\ &(\lambda\mu)x = \lambda(\mu x); \\ &1_R x = x. \end{aligned}$$

The analogous definition of a *right R-module* has scalar multiplication on the right instead. If R is a commutative ring then either definition can be taken, and as we will only be considering modules over the integers we will not make the distinction. An F-module V for some field F is the more familiar *vector space* over F.

Let R be a ring and let M, N be R-modules. A function $f : M \to N$ is an R-module homomorphism if

$$\begin{aligned} &f(x + y) = f(x) + f(y) \text{ for all } x, y \in M, \text{ and} \\ &f(\lambda x) = \lambda f(x) \text{ for all } \lambda \in R \text{ and } x \in M. \end{aligned}$$

B.2 Dimensions

Dimension spaces

A *dimension space* $\mathcal{D}$ is a set of dimension expressions identified up to equivalence $=_D$ and closed under dimension product and dimension exponentiation. Formally,

$$\begin{aligned} \delta_1, \delta_2 \in \mathcal{D} &\Rightarrow \delta_1 \cdot \delta_2 \in \mathcal{D} \\ \text{and } \delta \in \mathcal{D} &\Rightarrow \delta^x \in \mathcal{D} \text{ for any } x \in \mathbb{Z}. \end{aligned}$$

Then $\mathcal{D}$ forms a $\mathbb{Z}$-module if dimension product is interpreted as 'addition' and exponentiation as 'scalar multiplication'.

The dimension space *generated by* a set of dimension expressions $\{\delta_1, \dots, \delta_n\}$ is simply the closure of the set under product and exponentiation, and is denoted by $\mathrm{Dims}(\{\delta_1, \dots, \delta_n\})$. Then we define the *rank* of a dimension space $\mathcal{D}$ to be the minimal number of generators of $\mathcal{D}$. For example, the rank of $\mathcal{D} = \mathrm{Dims}(\{d_1^2, d_1^3, d_2^4\})$ is 2, because $\mathcal{D}$ can also be generated by the set $\{d_1, d_2^4\}$.

Dimensions as vectors of integers

By the above definition $\mathrm{Dims}(\mathcal{V})$ is the space of all dimension expressions whose free dimension variables are picked from the set $\mathcal{V}$. If the set $\mathcal{V}$ of dimension variables is finite, it is easier to think of elements of $\mathrm{Dims}(\mathcal{V})$ as vectors of integers. Let $\mathcal{V} = \{d_1, \dots, d_m\}$. Then a dimension $\delta \in \mathrm{Dims}(\mathcal{V})$ can be written as an m-vector whose elements are the exponents of the dimension variables in δ. So if

$$\mathrm{nf}(\delta) = d_1^{x_1} \cdots d_m^{x_m}$$

then we write

$$\underline{\delta} \stackrel{\text{def}}{=} \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}.$$

The (essential) free variables in a dimension δ are of course just the variables corresponding to the non-zero elements of the vector $\underline{\delta}$.

Linear independence and bases

Let $\mathcal{D}$ be a dimension space. Then $\delta_1, \ldots, \delta_n \in \mathcal{D}$ are *linearly dependent* if there are some $x_1, \ldots, x_n \in \mathbb{Z}$, not all zero, such that

$$\delta_1^{x_1} \cdots \delta_n^{x_n} = \mathbf{1}.$$

A set of dimensions that is not linearly dependent is said to be linearly *independent*. Notice that in contrast to the situation for vector spaces, if a set is linearly dependent it does not necessarily follow that any element can be expressed as a linear combination of the others. Consider, for example, the linearly dependent dimensions d^2 and d^3.

If a set $\{\delta_1, \ldots, \delta_r\}$ is linearly independent and generates a dimension space $\mathcal{D}$, then it is called a *basis* of $\mathcal{D}$. Then every $\delta \in \mathcal{D}$ can be written uniquely as a linear combination of elements of the basis. That is,

$$\delta = \delta_1^{x_1} \cdots \delta_r^{x_r}$$

for $x_1, \ldots, x_r \in \mathbb{Z}$. Every basis of a dimension space $\mathcal{D}$ contains $\text{rank}(\mathcal{D})$ elements; conversely, every set with $\text{rank}(\mathcal{D})$ elements which generates $\mathcal{D}$ is a basis of $\mathcal{D}$. Clearly $\mathcal{V}$ is a basis of $\text{Dims}(\mathcal{V})$.

If $\mathcal{D}'$ is a subset of the dimension space $\mathcal{D}$ and is itself a dimension space, then $\mathcal{D}'$ is said to be a *subspace* of $\mathcal{D}$, written $\mathcal{D}' \subseteq \mathcal{D}$.

Dimension space homomorphisms and substitutions

Let $\mathcal{D}, \mathcal{D}'$ be dimension spaces. We can recast the standard definition of module homomorphism as follows. A function $f : \mathcal{D} \to \mathcal{D}'$ is a dimension space *homomorphism* if

$$\begin{array}{l} f(\delta_1 \cdot \delta_2) = f(\delta_1) \cdot f(\delta_2) \text{ for all } \delta_1, \delta_2 \in \mathcal{D}, \text{ and} \\ f(\delta^x) = (f(\delta))^x \text{ for all } \delta \in \mathcal{D} \text{ and } x \in \mathbb{Z}. \end{array}$$

We will write $\text{Hom}(\mathcal{D}, \mathcal{D}')$ for the set of all dimension homomorphisms between $\mathcal{D}$ and $\mathcal{D}'$; if $\mathcal{D} = \mathcal{D}'$ then $\text{End}(\mathcal{D}) = \text{Hom}(\mathcal{D}, \mathcal{D})$, the set of all *endomorphisms* of $\mathcal{D}$.

Given a set of dimension variables $\mathcal{V}$, a dimension *substitution* S which involves only variables from $\mathcal{V}$ can be treated as a dimension endomorphism $S \in \text{End}(\text{Dims}(\mathcal{V}))$. At first it might appear that any dimension homomorphism can be represented by a dimension substitution. This is not the case: consider the homomorphism $f \in \text{Hom}(\text{Dims}(\{d^2\}), \text{Dims}(\{d\}))$ defined by

$$f(d^{2x}) = d^x.$$

Substitutions as square integer matrices

If $\mathcal{V}$ is finite and has m elements then a substitution $S \in \mathrm{End}(\mathrm{Dims}(\mathcal{V}))$ can be represented by an $m \times m$ matrix $\underline{S}$ whose i'th column corresponds to the dimension δ_i to be substituted for dimension variable d_i. If

$$S = \{\, d_1 \mapsto \delta_1, \ldots , d_m \mapsto \delta_m \,\}$$

then

$$\underline{S} \stackrel{\mathrm{def}}{=} \begin{pmatrix} | & & | \\ \underline{\delta_1} & \cdots & \underline{\delta_m} \\ | & & | \end{pmatrix}.$$

Then the application of a substitution S to a dimension δ is just the multiplication of a vector by a matrix:

$$\underline{S(\delta)} = \underline{S}\,\underline{\delta}.$$

The composition of two substitutions is just matrix multiplication:

$$\underline{S_1 \circ S_2} = \underline{S_1}\,\underline{S_2}.$$

The inverse of a substitution S, if it exists, is given by the inverse of its matrix representation $\underline{S}^{-1}$. Moreover, an integer matrix has an inverse if and only if its determinant is ± 1. Hence a substitution S has inverse given by

$$\underline{S^{-1}} = \underline{S}^{-1}$$

if and only if $\det(\underline{S}) = \pm 1$.

If a square integer matrix is invertible then it can be written as the product of a set of matrices which correspond to *elementary row operations*. These are the following:

1. Multiplication of a row by -1. As a substitution this takes the form

   $$\{d_i \mapsto d_i^{-1}\}.$$

2. Interchange of any two rows. This corresponds to a *renaming* of the form

   $$\{d_i \mapsto d_j,\, d_j \mapsto d_i\}, \quad i \neq j.$$

   Of course, an arbitrary permutation of dimension variables can be built up from a sequence of such substitutions.

3. Addition of the multiple of one row to another row. This corresponds to a substitution of the form

   $$\{d_i \mapsto d_i \cdot d_j^x\}, \quad i \neq j$$

   for some $x \in \mathbb{Z}$.

B.3 Types

Type spaces

First we define the 'product' of two types, defined simply as the pointwise product of corresponding dimension components. The types must be 'compatible', that is, they have the same type structure when the dimensions are omitted (in the notation of Chapter 5 this is just the dimension erasure $\tau^\star$).

$$\begin{aligned}\mathsf{bool} \cdot \mathsf{bool} &= \mathsf{bool} \\ \mathsf{real}\,\delta \cdot \mathsf{real}\,\delta' &= \mathsf{real}\,\delta \cdot \delta' \\ (\tau_1 \to \tau_2) \cdot (\tau_1' \to \tau_2') &= (\tau_1 \cdot \tau_1') \to (\tau_2 \cdot \tau_2').\end{aligned}$$

Similarly we can define an exponentiation operation on types:

$$\begin{aligned}\mathsf{bool}^x &= \mathsf{bool} \\ (\mathsf{real}\,\delta)^x &= \mathsf{real}\,\delta^x \\ (\tau_1 \to \tau_2)^x &= \tau_1^x \to \tau_2^x.\end{aligned}$$

Then a *type space* $\mathcal{T}$ is a set of type expressions identified up to equivalence $=_D$ and closed under product and exponentiation. Formally,

$$\begin{aligned}\tau_1, \tau_2 \in \mathcal{T} &\Rightarrow \tau_1 \cdot \tau_2 \in \mathcal{T} \\ \text{and } \tau \in \mathcal{T} &\Rightarrow \tau^x \in \mathcal{T} \text{ for any } x \in \mathbb{Z}.\end{aligned}$$

So $\mathcal{T}$ forms a $\mathbb{Z}$-module if type product is interpreted as 'addition' and exponentiation as 'scalar multiplication'.

Suppose that the elements of a type space $\mathcal{T}$ each have n dimension components drawn from dimension spaces $\mathcal{D}_1, \ldots, \mathcal{D}_n$. If the type structure is ignored, then $\mathcal{T}$ is the *direct sum* of $\mathcal{D}_1, \ldots, \mathcal{D}_n$: the cartesian product $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ treated as a $\mathbb{Z}$-module by the pointwise product and exponentiation of dimensions. This is usually denoted $\mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$.

Usually all dimension components in a type are taken from the same dimension space. Hence we will let $\mathrm{Types}(\mathcal{D}, \tau^\star)$ stand for the space of types whose dimension erasure is $\tau^\star$ with dimension components taken from $\mathcal{D}$. Then if the type structure is ignored this is just the direct sum $\mathcal{D} \oplus \cdots \oplus \mathcal{D}$ for n copies of $\mathcal{D}$, where n is the number of components in each type.

Types as integer matrices

By the above definition $\mathrm{Types}(\mathrm{Dims}(\mathcal{V}), \tau^\star)$ is the space of all types whose dimension erasure is $\tau^\star$ and whose free dimension variables are picked from the set $\mathcal{V}$. If the set $\mathcal{V}$ of dimension variables is finite, it is easier to think of the elements as rectangular matrices of integers. Let $\mathcal{V} = \{d_1, \ldots, d_m\}$ and let n be the number of dimension components in $\tau^\star$. Then the dimension components in a type $\tau \in \mathrm{Types}(\mathrm{Dims}(\mathcal{V}), \tau^\star)$ can be represented by an $m \times n$ matrix $\underline{\tau}$, defined

as follows by induction on the structure of the type:

$$\underline{\mathsf{bool}} \stackrel{\text{def}}{=} ()$$
$$\underline{\mathsf{real}\,\delta} \stackrel{\text{def}}{=} \underline{\delta}$$
$$\underline{\tau_1 \to \tau_2} \stackrel{\text{def}}{=} \underline{\tau_1}|\underline{\tau_2}$$

Here $()$ denotes the empty matrix and $A|B$ joins an $m \times n$ matrix A to an $m \times r$ matrix B to give an $m \times (n + r)$ matrix.

For example, assuming a two-element set of dimension variables, the type $\mathsf{real}\, d_1 \to \mathsf{real}\, d_2 \to \mathsf{real}\, d_1 \cdot d_2$ is represented by the dimensionless type $\mathsf{real}\,\mathbf{1} \to \mathsf{real}\,\mathbf{1} \to \mathsf{real}\,\mathbf{1}$ together with the matrix

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

The set of free variables in a type are those which correspond to non-zero rows in its matrix representation.

Now consider our notion of a *free variable reduced form* for types, defined on page 30. Matrix theoretically, this corresponds to *full row rank* form. Its rank (number of degrees of freedom in τ) is equal to the number of non-zero rows (number of free dimension variables in τ). For example, the dimension components of the type $\mathsf{real}\, d_1 \cdot d_2^{-1} \to \mathsf{real}\, d_1^{-2} \cdot d_2 \to \mathsf{real}\, d_1^{-1}$ can be represented by the full row rank matrix

$$\begin{pmatrix} 1 & -2 & -1 \\ -1 & 1 & 0 \end{pmatrix}.$$

In contrast, the type $\mathsf{real}\, d_1 \cdot d_2^2 \to \mathsf{real}\, d_1^2 \cdot d_2^4 \to \mathsf{real}\, d_1^{-1} \cdot d_2^{-2}$ has only one degree of freedom and hence its matrix representation has rank 1 and is not in full row rank form:

$$\begin{pmatrix} 1 & 2 & -1 \\ 2 & 4 & -2 \end{pmatrix}.$$

B.4 Type schemes

Closed type schemes

Suppose that $\tau \in \mathcal{T}$ for some type space $\mathcal{T}$. Then the set of all *instantiations* of τ is given by the smallest set containing τ which is closed under endomorphisms on $\mathcal{T}$:

$$\mathrm{Insts}(\tau) = \{\, f(\tau) \;\mid\; f \in \mathrm{End}(\mathcal{T}) \,\}.$$

It is clear that $\mathrm{Insts}(\tau)$ is itself a type space, so $\mathrm{Insts}(\tau) \subseteq \mathcal{T}$.

Let $\mathcal{T} = \mathrm{Types}(\mathrm{Dims}(\mathcal{V}), \tau^\star)$. If $\sigma = \forall \vec{d}.\tau_b$ is a closed type scheme with $\tau_b \in \mathcal{T}$ then $\mathrm{Insts}(\tau_b)$ is the space of all specialisations of σ with dimension

variables picked from $\mathcal{V}$. This is a very natural interpretation of σ. In particular, observe that for closed type schemes $\forall\vec{d_1}.\tau_1$ and $\forall\vec{d_2}.\tau_2$ with $\tau_1, \tau_2 \in \mathcal{T}$,

$$\forall\vec{d_1}.\tau_1 \preccurlyeq_D \forall\vec{d_2}.\tau_2 \;\Rightarrow\; \mathrm{Insts}(\tau_1) \supseteq \mathrm{Insts}(\tau_2).$$

Geometrically, one can think of $\mathrm{Insts}(\tau_b) \subseteq \mathcal{T}$ as a collection of points in a plane in the type space $\mathcal{T}$ which passes through the origin.

Returning to the syntax, from Lemma 3.10 we know that two closed type schemes $\forall\vec{d_1}.\tau_1$ and $\forall\vec{d_2}.\tau_2$ are equivalent if and only if there is an invertible substitution U such that $U(\tau_1) =_D \tau_2$. Recasting this in terms of matrices, there is an invertible matrix $\underline{U}$ such that $\underline{U}\underline{\tau_1} = \underline{\tau_2}$. If a matrix A can be transformed into another matrix B by multiplication on the left by an invertible matrix, then A and B are said to be *row-equivalent* (sometimes: *left equivalent*). Thus (for closed type schemes) our notion of type scheme equivalence corresponds to the notion of row equivalence from matrix theory. For each equivalence class induced by this equivalence relation there is a single representative with the form described on page 60—the Hermite Normal Form. Our simplification algorithm Simplify essentially calculates the invertible matrix necessary to bring a rectangular matrix into this form: if $\mathrm{Simplify}(\emptyset, \tau) = U$ then the matrix $\underline{U}\,\underline{\tau}$ is in Hermite Normal Form.

Quotient spaces

To deal with the situation of open type schemes we introduce the idea of a *quotient space*. Consider a type space $\mathcal{T}$ and a subspace $\mathcal{T}' \subseteq \mathcal{T}$. Then for some $\tau \in \mathcal{T}$ define the *left coset* $\tau \cdot \mathcal{T}'$ by

$$\tau \cdot \mathcal{T}' = \{\, \tau \cdot \tau' \mid \tau' \in \mathcal{T}' \,\}.$$

Now the collection of all such cosets of $\mathcal{T}'$ containing elements from $\mathcal{T}$ is given by

$$\mathcal{T}/\mathcal{T}' = \{\, \tau \cdot \mathcal{T}' \mid \tau \in \mathcal{T} \,\}.$$

This then forms a $\mathbb{Z}$-module called the *quotient space* $\mathcal{T}/\mathcal{T}'$ when product and exponentiation operations are defined for cosets by

$$\begin{aligned} (\tau_1 \cdot \mathcal{T}') \cdot (\tau_2 \cdot \mathcal{T}') &= (\tau_1 \cdot \tau_2) \cdot \mathcal{T}'; \\ (\tau \cdot \mathcal{T}')^x &= \tau^x \cdot \mathcal{T}'. \end{aligned}$$

A type space endomorphism $f : \mathcal{T} \to \mathcal{T}$ can be extended to give an endomorphism $f' : \mathcal{T}/\mathcal{T}' \to \mathcal{T}/\mathcal{T}'$ by the following definition:

$$f'(\tau \cdot \mathcal{T}') = f(\tau) \cdot \mathcal{T}'.$$

This is well-defined only if $\mathcal{T}'$ is *invariant* under f: that is, if $f(\tau) \in \mathcal{T}'$ whenever $\tau \in \mathcal{T}'$.

Open type schemes

We now discuss how the notion of a quotient space can be used to model open type schemes in which some variables are free. We allow the possibility of *non-essential* free variables as discussed on page 28. Suppose that we are given a type scheme $\sigma = \forall d_1 \ldots \forall d_m.\tau$ with

$$\mathrm{fdv}(\tau) = \{d_1, \ldots, d_m, d_{m+1}, \ldots, d_p, d_{p+1}, \ldots, d_n\} \subseteq \mathcal{V}.$$

Here $d_1, \ldots, d_m$ are the bound variables, $d_{m+1}, \ldots, d_p$ are the non-essential free variables, and $d_{p+1}, \ldots, d_n$ are the essential free variables, and $\tau \in \mathcal{T}$ with $\mathcal{T} = \mathrm{Types}(\mathrm{Dims}(\mathcal{V}), \tau^\star)$. Then τ can be decomposed into its bound part τ_b, non-essential free part τ_n and essential free part τ_f as follows:

$$\begin{aligned}
\tau_b &= \{d_{m+1} \mapsto \mathbf{1}, \ldots, d_n \mapsto \mathbf{1}\}\tau \\
\tau_n &= \{d_1 \mapsto \mathbf{1}, \ldots, d_m \mapsto \mathbf{1}, d_{p+1} \mapsto \mathbf{1}, \ldots, d_n \mapsto \mathbf{1}\}\tau \\
\tau_f &= \{d_1 \mapsto \mathbf{1}, \ldots, d_p \mapsto \mathbf{1}\}\tau.
\end{aligned}$$

Clearly $\tau = \tau_f \cdot \tau_b \cdot \tau_n$. Now a particular type scheme can be represented by the left coset

$$\tau_f \cdot \tau_n \cdot \mathrm{Insts}(\tau_b).$$

Geometrically, think of this as a plane parallel to $\mathrm{Insts}(\tau_b)$ which passes through the point $\tau_f \cdot \tau_n$. As τ_n is non-essential we can remove it: in fact, it is just an instance of the bound part of the type scheme ($\tau_n \in \mathrm{Insts}(\tau_b)$). Hence $\tau_f \cdot \tau_n \cdot \mathrm{Insts}(\tau_b) = \tau_f \cdot \mathrm{Insts}(\tau_b)$. Geometrically, the line through the origin passing through τ_f is perpendicular to the plane $\mathrm{Insts}(\tau_b)$, and the effect of τ_n is a translation which leaves $\mathrm{Insts}(\tau_b)$ alone.

Then we can form a *type scheme space* $\mathcal{T}/\,\mathrm{Insts}(\tau_b)$ containing all type schemes with τ_b as bound part. Geometrically, this quotient space is the collection of all planes parallel to the plane $\mathrm{Insts}(\tau_b)$ which represents a closed type scheme.

The representation of closed type schemes by $\mathrm{Insts}(\tau_b) \subseteq \mathcal{T}$ is closed under type space endomorphisms on $\mathcal{T}$; hence $\mathrm{Insts}(\tau_b)$ is an invariant subspace of $\mathcal{T}$ for any endomorphism $f : \mathcal{T} \to \mathcal{T}$. Then the extension of endomorphisms to quotient spaces described earlier explains correctly the extension of substitutions on types to type schemes:

$$S(\tau_f \cdot \mathrm{Insts}(\tau_b)) = S(\tau_f) \cdot \mathrm{Insts}(\tau_b).$$

Observe how S is applied only to the free part τ_f of the type scheme, as required.

Finally, we consider what the type scheme ordering means for open type schemes. Clearly $\sigma_1 \preccurlyeq_D \sigma_2$ can only hold when the free parts of σ_1 and σ_2 are the same. Also if the type space $\mathcal{T}_1$ represents the bound part of σ_1 and $\mathcal{T}_2$ represents the bound part of σ_2 so that $\sigma_1 \in \mathcal{T}/\mathcal{T}_1$ and $\sigma_2 \in \mathcal{T}/\mathcal{T}_2$ then $\mathcal{T}_2 \subseteq \mathcal{T}_1$. So the relation $\sigma_1 \preccurlyeq_D \sigma_2$ is given by the natural injective homomorphism $\iota : \mathcal{T}/\mathcal{T}_2 \to \mathcal{T}/\mathcal{T}_1$ defined by

$$\iota(\tau \cdot \mathcal{T}_2) = \tau \cdot \mathcal{T}_1.$$

Appendix C

Omitted proofs

In this appendix we present the proofs of soundness and completeness for the inference algorithm which were omitted from Chapter 3 (Theorems 3.3 and 3.4). Although these are standard results, more care is taken than usual over the generation of fresh variable names. To simplify the statement of completeness, let $\mathrm{Ty}(\mathcal{V})$ denote the set of all simple types whose free variables are in $\mathcal{V}$, let $\mathrm{Ass}(\mathcal{V})$ denote the set of all type assignments whose free variables are in $\mathcal{V}$, and let $\mathrm{Substs}(\mathcal{V}, \mathcal{V}')$ denote the set of all substitutions whose domains are included in $\mathcal{V}$ and whose range involves variables only from $\mathcal{V}'$.

Theorem (Soundness of Infer**).** *If* $\mathrm{Infer}(\mathcal{V}, \Gamma, e) = (\mathcal{V}', S, \tau)$ *then there is a typing derivation* $S(\Gamma) \vdash_{\mathrm{sd}} e : \tau$.

Proof. By induction on the structure of e.

- If e is a variable x or a constant r, then the derivation follows immediately.
- If the expression is an abstraction $\lambda x.e$ then

$$\begin{array}{l}\mathrm{Infer}(\mathcal{V}, \Gamma, \lambda x.e) = (\mathcal{V}', S|_{\mathcal{V}}, S(t) \to \tau) \\ \quad \text{where} \\ \qquad (\mathcal{V}', S, \tau) = \mathrm{Infer}(\mathcal{V} \cup \{t\}, \Gamma[x : t], e) \\ \qquad t \text{ is a type variable not in } \mathcal{V}.\end{array}$$

  By the induction hypothesis there must be a derivation of

$$S(\Gamma[x : t]) \vdash e : \tau$$

  so by an application of the (abs) rule we can obtain the following derivation:

$$\frac{S(\Gamma)[x : S(t)] \vdash e : \tau}{S(\Gamma) \vdash \lambda x.e : S(t) \to \tau}\ (\mathrm{abs}).$$

- If we have an application $e_1\,e_2$ then

$$\begin{array}{l}\text{Infer}(\mathcal{V},\Gamma,e_1\,e_2) = (\mathcal{V}_2 \cup \{t\}, (S_3 \circ S_2 \circ S_1)|_{\mathcal{V}}, S_3(t))\\ \quad\text{where}\\ \quad\quad (\mathcal{V}_1, S_1, \tau_1) = \text{Infer}(\mathcal{V},\Gamma,e_1)\\ \quad\quad (\mathcal{V}_2, S_2, \tau_2) = \text{Infer}(\mathcal{V}_1, S_1(\Gamma), e_2)\\ \quad\quad S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \to t)\\ \quad\quad t \text{ is a type variable not in } \mathcal{V}_2.\end{array}$$

By the induction hypothesis there are derivations of

$$\begin{array}{rl} & S_1(\Gamma) \vdash e_1 : \tau_1\\ \text{and} & S_2(S_1(\Gamma)) \vdash e_2 : \tau_2\end{array}$$

and from Lemma 2.14 (substitution on a derivation) we can deduce that these imply derivations of

$$\begin{array}{rl} & S_3(S_2(S_1(\Gamma))) \vdash e_1 : S_3(S_2(\tau_1))\\ \text{and} & S_3(S_2(S_1(\Gamma))) \vdash e_2 : S_3(\tau_2).\end{array}$$

Now we know from the soundness of Unify (Theorem 3.2) that

$$S_3(S_2(\tau_1)) =_D S_3(\tau_2) \to S_3(t)$$

so we can construct a typing derivation as follows:

$$\frac{S_3(S_2(S_1(\Gamma))) \vdash e_1 : S_3(\tau_2) \to S_3(t) \quad S_3(S_2(S_1(\Gamma))) \vdash e_2 : S_3(\tau_2)}{S_3(S_2(S_1(\Gamma))) \vdash e_1\,e_2 : S_3(t)}\ \text{(app)}$$

- If the expression is a let construct then

$$\begin{array}{l}\text{Infer}(\mathcal{V},\Gamma,\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = (\mathcal{V}_2, S_2 \circ S_1, \tau_2)\\ \quad\text{where}\\ \quad\quad (\mathcal{V}_1, S_1, \tau_1) = \text{Infer}(\mathcal{V},\Gamma,e_1)\\ \quad\quad (\mathcal{V}_2, S_2, \tau_2) = \text{Infer}(\mathcal{V}_1, S_1(\Gamma)[x:\sigma], e_2)\\ \quad\quad \sigma = \text{Gen}(S_1(\Gamma), \tau_1)\end{array}$$

By the induction hypothesis there are derivations of

$$S_1(\Gamma) \vdash e_1 : \tau_1 \tag{1}$$

$$\text{and } S_2(S_1(\Gamma)[x : \text{Gen}(S_1(\Gamma),\tau_1)]) \vdash e_2 : \tau_2. \tag{2}$$

By Lemma 2.10 there is a substitution R such that

$$R(S_1(\Gamma)) \cong_D S_2(S_1(\Gamma))$$

and

$$S_2(\text{Gen}(S_1(\Gamma),\tau_1)) \cong_D \text{Gen}(R(S_1(\Gamma)), R(\tau_1)).$$

Then we can apply Lemma 2.14 (substitution) to (1) to obtain a derivation of

$$R(S_1(\Gamma)) \vdash e_1 : R(\tau_1)$$

and rewrite (2) as

$$R(S_1(\Gamma))[x : \text{Gen}(R(S_1(\Gamma)), R(\tau_1))] \vdash e_2 : \tau_2.$$

So finally we can construct the following derivation:

$$\frac{R(S_1(\Gamma)) \vdash e_1 : R(\tau_1) \quad R(S_1(\Gamma))[x : \text{Gen}(R(S_1(\Gamma)), R(\tau_1))] \vdash e_2 : \tau_2}{R(S_1(\Gamma)) \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \text{(let}'\text{)}$$

Identifying type schemes up to equivalence, this is a derivation of

$$S_2(S_1(\Gamma)) \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2$$

as required.

□

Theorem (Completeness of Infer**).** *Suppose* $\Gamma \in \text{Ass}(\mathcal{V})$*,* $S \in \text{Substs}(\mathcal{V}, \mathcal{V}'')$ *and* $\tau \in \text{Ty}(\mathcal{V}'')$*. If* $S(\Gamma) \vdash_{\text{sd}} e : \tau$ *then* $\text{Infer}(\mathcal{V}, \Gamma, e) = (\mathcal{V}', S_0, \tau_0)$ *for some* $S_0 \in \text{Substs}(\mathcal{V}, \mathcal{V}')$*,* $\tau_0 \in \text{Ty}(\mathcal{V}')$ *such that*

$$S =_D R \circ S_0 \text{ and } \tau =_D R(\tau_0)$$

for some substitution $R \in \text{Substs}(\mathcal{V}', \mathcal{V}'')$*.*

Proof. By induction on the structure of e. We present every case in detail; it is easy to see that all uses of the induction hypothesis are valid by checking that the types, type assignments and substitutions belong to the appropriate sets.

- A variable x must have the typing derivation

$$\frac{}{S(\Gamma) \vdash x : \tau'} \text{(var}'\text{)}$$

with $S(\Gamma)(x) \preccurlyeq_D \tau'$. From the definition of the algorithm we have:

$$\begin{array}{l} \text{Infer}(\mathcal{V}, \Gamma, x) = (\mathcal{V} \cup \vec{v'}, I, \{\vec{v} \mapsto \vec{v'}\}\tau) \\ \quad \text{where} \\ \qquad \Gamma(x) \text{ is } \forall \vec{v}.\tau \\ \qquad \vec{v'} \text{ are type and dimension variables not in } \mathcal{V}. \end{array}$$

Now

$$\begin{aligned} S(\Gamma)(x) &= S(\forall \vec{v}.\tau) \\ &=_D \forall \vec{v'}.S(\{\vec{v} \mapsto \vec{v'}\}\tau) \\ &\preccurlyeq_D \tau'. \end{aligned}$$

From the definition of specialisation there must be some substitution R' with $\text{dom}(R') \subseteq \vec{v'}$ such that

$$R'(S(\{\vec{v} \mapsto \vec{v'}\}\tau)) =_D \tau'.$$

Then $R' \circ S$ is the substitution R required by the theorem.

- The case for a non-zero constant r is trivial.
- A constant zero must have been typed by the derivation

$$\frac{}{S(\Gamma) \vdash 0 : \mathsf{real}\ \delta}\ (\text{zero})$$

From the definition of the algorithm,

$$\begin{array}{l} \text{Infer}(\mathcal{V}, \Gamma, 0) = (\mathcal{V} \cup \{d\}, I, \mathsf{real}\ d) \\ \quad \text{where} \\ \qquad d \text{ is a dimension variable not in } \mathcal{V}. \end{array}$$

Then $\{d \mapsto \delta\} \circ S$ is the substitution R required by the theorem.

- An abstraction $\lambda x.e$ must have been typed by the derivation

$$\frac{S(\Gamma)[x : \tau_1] \vdash e : \tau_2}{S(\Gamma) \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\text{abs})$$

From the algorithm,

$$\begin{array}{l} \text{Infer}(\mathcal{V}, \Gamma, \lambda x.e) = (\mathcal{V}', S_0|_{\mathcal{V}}, S_0(t) \to \tau_0) \\ \quad \text{where} \\ \qquad (\mathcal{V}', S_0, \tau_0) = \text{Infer}(\mathcal{V} \cup \{t\}, \Gamma[x : t], e) \\ \qquad t \text{ is a type variable not in } \mathcal{V}. \end{array}$$

Now let $S' = \{t \mapsto \tau_1\} \circ S$ so that the premise can be rewritten as

$$S'(\Gamma[x : t]) \vdash e : \tau_2.$$

Then by the induction hypothesis there is a substitution R such that

$$S' =_D R \circ S_0 \quad (1)$$
$$\tau_2 =_D R(\tau_0). \quad (2)$$

Then R is the substitution we need, because equation (1) implies that

$$S =_D R \circ S_0|_{\mathcal{V}}$$

and

$$\begin{array}{rll} R(S_0(t) \to \tau_0) & =_D S'(t) \to R(\tau_0) & \text{by (1)} \\ & =_D \tau_1 \to \tau_2 & \text{by definition of } S' \text{ and (2)} \end{array}$$

as required.

- An application $e_1\, e_2$ must have been typed by the derivation:

$$\frac{S(\Gamma) \vdash e_1 : \tau_1 \to \tau_2 \qquad S(\Gamma) \vdash e_2 : \tau_1}{S(\Gamma) \vdash e_1\, e_2 : \tau_2}\ (\text{app})$$

From the algorithm,

$$\begin{aligned}&\text{Infer}(\mathcal{V}, \Gamma, e_1\, e_2) = (\mathcal{V}_2 \cup \{t\}, (S_3 \circ S_2 \circ S_1)|_{\mathcal{V}}, S_3(t))\\&\quad \text{where}\\&\qquad (\mathcal{V}_1, S_1, \tau_f) = \text{Infer}(\mathcal{V}, \Gamma, e_1)\\&\qquad (\mathcal{V}_2, S_2, \tau_a) = \text{Infer}(\mathcal{V}_1, S_1(\Gamma), e_2)\\&\qquad S_3 = \text{Unify}(S_2(\tau_f), \tau_a \to t)\\&\qquad t \text{ is a type variable not in } \mathcal{V}_2.\end{aligned}$$

Applying the induction hypothesis to the first premise gives a substitution R_1 such that

$$S =_D R_1 \circ S_1 \tag{1}$$

$$\tau_1 \to \tau_2 =_D R_1(\tau_f). \tag{2}$$

Using equation (1) the second premise can be rewritten as

$$R_1(S_1(\Gamma)) \vdash e_2 : \tau_1.$$

Applying the induction hypothesis to this gives a substitution R_2 such that

$$R_1 =_D R_2 \circ S_2 \tag{3}$$

$$\tau_1 =_D R_2(\tau_a). \tag{4}$$

Now let $R_2' = \{t \mapsto \tau_2\}$. Then

$$\begin{aligned}R_2'(R_2(S_2(\tau_f))) &=_D R_2'(R_1(\tau_f)) && \text{by (3)}\\&=_D \tau_1 \to \tau_2 && \text{by (2)}\end{aligned}$$

and

$$\begin{aligned}R_2'(R_2(\tau_a \to t)) &=_D R_2(\tau_a) \to \tau_2 && \text{because } t \notin \mathcal{V}_2\\&=_D \tau_1 \to \tau_2 && \text{by (4).}\end{aligned}$$

Hence by completeness of Unify (Theorem 3.2) there must be some substitution T such that

$$R_2' \circ R_2 =_D T \circ S_3.$$

Then

$$\begin{aligned}S &=_D R_1 \circ S_1 && \text{by (1)}\\&=_D R_2 \circ S_2 \circ S_1 && \text{by (3)}\\&=_D (R_2' \circ R_2 \circ S_2 \circ S_1)|_{\mathcal{V}}\\&=_D (T \circ S_3 \circ S_2 \circ S_1)|_{\mathcal{V}}\\&=_D T \circ (S_3 \circ S_2 \circ S_1)|_{\mathcal{V}}.\end{aligned}$$

This is the first part of the result we require. For the second part,

$$\begin{aligned}T(S_3(t)) &=_D R_2'(R_2(t))\\&=_D \tau_2.\end{aligned}$$

- A let construct must have been typed by the derivation:

$$\frac{S(\Gamma) \vdash e_1 : \tau_1 \qquad S(\Gamma)[x : \mathrm{Gen}(S(\Gamma), \tau_1)] \vdash e_2 : \tau_2}{S(\Gamma) \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\mathrm{let})$$

From the definition of Infer,

$$\begin{array}{l} \mathrm{Infer}(\mathcal{V}, \Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = (\mathcal{V}_2, S_2 \circ S_1, \tau_2') \\ \quad \text{where} \\ \qquad (\mathcal{V}_1, S_1, \tau_1') = \mathrm{Infer}(\mathcal{V}, \Gamma, e_1) \\ \qquad (\mathcal{V}_2, S_2, \tau_2') = \mathrm{Infer}(\mathcal{V}_1, S_1(\Gamma)[x : \mathrm{Gen}(S_1(\Gamma), \tau_1')], e_2) \end{array}$$

By the induction hypothesis applied to the first premise there is some substitution R_1 such that

$$S =_D R_1 \circ S_1 \tag{1}$$
$$\tau_1 =_D R_1(\tau_1'). \tag{2}$$

Now we can rewrite the second premise as:

$$R_1(S_1(\Gamma))[x : \mathrm{Gen}(R_1(S_1(\Gamma)), R_1(\tau_1'))] \vdash e_2 : \tau_2.$$

Let Γ' be the type assignment in this judgment. Then by Lemma 2.9 we know that

$$R_1(S_1(\Gamma)[x : \mathrm{Gen}(S_1(\Gamma), \tau_1')]) \preccurlyeq_D \Gamma'$$

and so by Lemma 2.13 (context generalisation) there is a derivation of

$$R_1(S_1(\Gamma)[x : \mathrm{Gen}(S_1(\Gamma), \tau_1')]) \vdash e_2 : \tau_2.$$

It follows by the induction hypothesis that there is a substitution R_2 such that

$$R_1 =_D R_2 \circ S_2 \tag{3}$$
$$\tau_2 =_D R_2(\tau_2'). \tag{4}$$

Then R_2 is the substitution we require, because

$$\begin{array}{rll} S & =_D R_1 \circ S_1 & \text{by (1)} \\ & =_D R_2 \circ S_2 \circ S_1 & \text{by (3).} \end{array}$$

□

Bibliography

[1] W. A. Adkins and S. H. Weintraub. *Algebra: An Approach via Module Theory.* Springer-Verlag, 1992.

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

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

[4] P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, University of Cambridge Computer Laboratory, August 1993. Technical Report 309.

[5] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit. Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[6] G. Birkhoff. *Hydrodynamics: A Study in Logic, Fact and Similitude.* Princeton University Press, Revised edition, 1960.

[7] T. S. Blyth. *Module Theory: An approach to linear algebra.* Oxford University Press, second edition, 1990.

[8] A. Boudet, J.-P. Jouannaud, and M. Schmidt-Schauss. Unification in Boolean rings and Abelian groups. *Journal of Symbolic Computation*, 8:449–477, 1989.

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

[10] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (extended abstract). In *ACM Conference on Lisp and Functional Programming.* ACM Press, June 1992.

[11] R. Di Cosmo. Type isomorphisms in a type-assignment framework. In *19th Annual ACM Symposium on Principles of Programming Languages*, pages 200–210, 1992.

[12] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Computer Science, University of Edinburgh, April 1985. Technical Report CST-33-85.

[13] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[14] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

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

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

[17] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[18] J. Goubault. Inférence d'unités physiques en ML. In P. Cointe, C. Queinnec, and B. Serpette, editors, *Journées Francophones des Langages Applicatifs, Noirmoutier*, pages 3–20. INRIA, Collection didactique, 1994.

[19] C. A. Gunter. *Semantics of Programming Langauges: Structures and Techniques*. The MIT Press, 1992.

[20] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[21] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. Technical Report CMU-CS-FOX-94-07, School of Computer Science, Cargnegie Mellon University, September 1994.

[22] I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. Technical Report 94-31, Software Verification Research Centre, University of Queensland, November 1994.

[23] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

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

[25] P. Hudak and P. L. Wadler. Report on the functional programming language Haskell. Technical report, Department of Computing Science, Glasgow University, April 1990.

[26] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1992.

[27] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. Technical Report 561, Université de Paris-Sud, 1990.

[28] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. Elsevier Science Publishers B. V. North Holland, 1988.

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

[30] A. J. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 1994.

[31] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

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

[33] D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky. *Foundations of Measurement*, volume I. Academic Press, 1971.

[34] D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky. *Foundations of Measurement*, volume III. Academic Press, 1990.

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

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

[37] X. Leroy. *Polymorphic typing of an algorithmic language*. PhD thesis, INRIA-Rocquencourt, October 1992. Technical Report 1778.

[38] X. Leroy. Unboxed objects and polymorphic typing. In *19th Annual ACM Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[39] J. D. Logan. *Applied Mathematics: A Contemporary Approach*. John Wiley and Sons, 1987.

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

[41] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[42] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[43] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.

[44] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, Amsterdam, 1990.

[45] A. Mycroft. Polymorphic types schemes and recursive definitions. In *Proc. International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[46] M. Newman. *Integral Matrices.* Academic Press, 1972.

[47] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[48] I. Niven and H. S. Zuckerman. *An Introduction to the Theory of Numbers.* John Wiley and Sons, 1980.

[49] M. Odersky and K. Läufer. Putting type annotations to work (preliminary). Presented at Newton Institute Workshop on Advances in Type Systems for Computing, Cambridge, England, August 1995.

[50] C.-H. L. Ong. Correspondence between operational and denotational semantics: the full abstraction problem for PCF. In S. Abramsky, D. B. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4. Oxford University Press, 1995.

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

[52] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[53] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in Pascal.* Cambridge University Press, 1989.

[54] D. Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, INRIA Rocquencourt, May 1991.

[55] D. Rémy. Extension of ML type system with a sorted equational theory on types. Technical Report 1766, INRIA Rocquencourt, October 1992.

[56] J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[57] M. Rittri. Semi-unification of two terms in Abelian groups. *Information Processing Letters*, 52(2):61–68, 1994.

[58] M. Rittri. Dimension inference under polymorphic recursion. In *Proceedings of the 8th Annual Conference on Functional Programming Languages and Computer Architecture*, June 1995.

[59] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, 1965.

[60] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3/4):207–274, 1989.

[61] V. Tannen. Tutorial: Languages for collection types. In *13th ACM Symposium on Principles of Database Systems*. ACM Press, May 1994.

[62] S. R. Thatte. Coercive type isomorphism. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 29–49. Springer-Verlag, 1991.

[63] S. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[64] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, University of Edinburgh, May 1988. Technical Report ECS-LFCS-88-54.

[65] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages*, 1994.

[66] P. Wadler. Theorems for free! In *Proceedings of the 4th International Symposium on Functional Programming Languages and Computer Architecture*, 1989.

[67] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[68] 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.

[69] G. Winksel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.

[70] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.