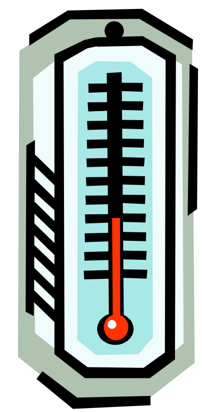
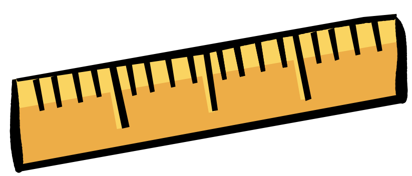
Types for Units-of-Measure: Theory and Practice

CEFP'09, Komarno, Slovakia



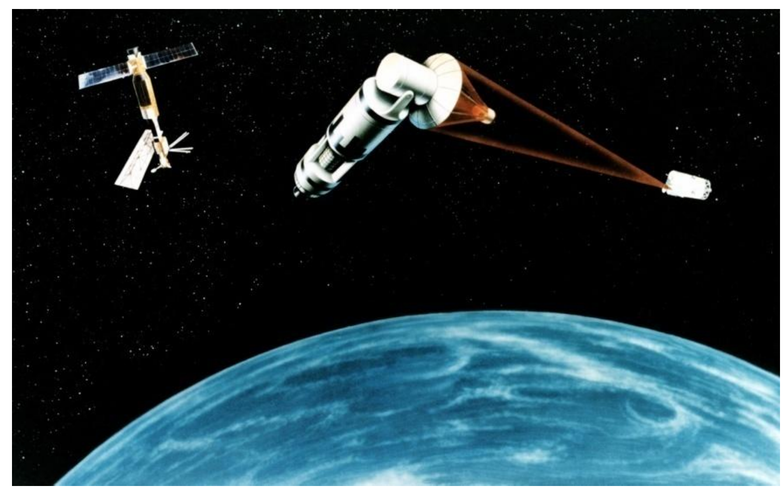
Andrew Kennedy
Microsoft Research, Cambridge



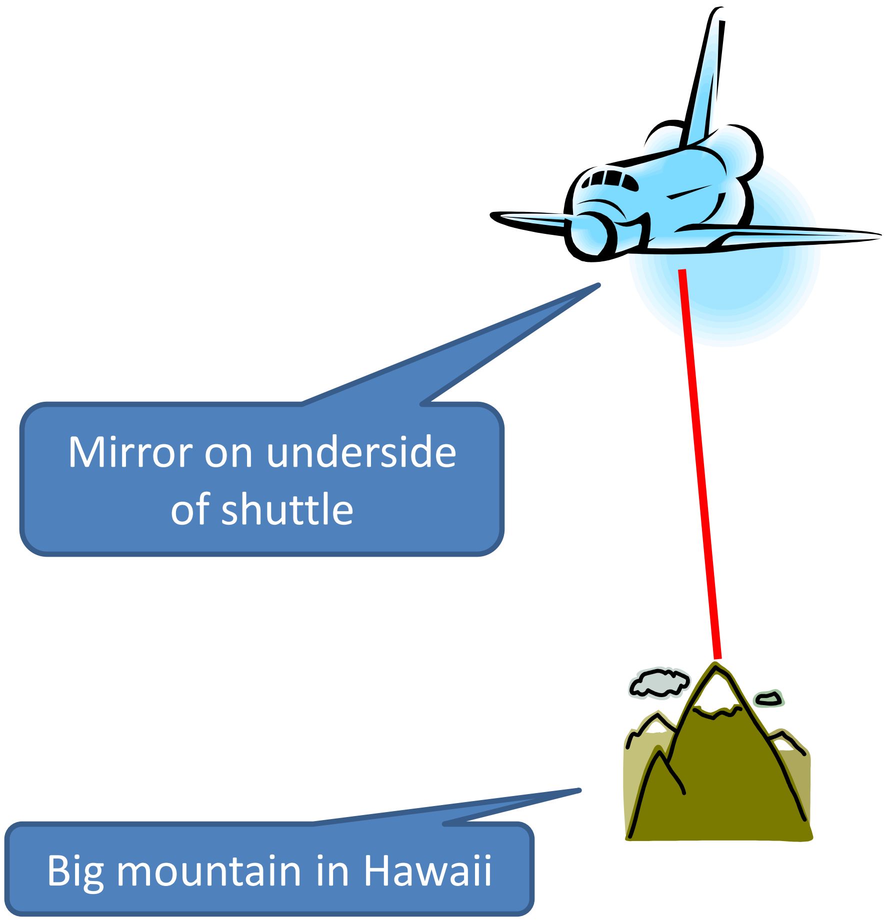
NASA “Star Wars” experiment, 1983

23rd March 1983. Ronald Reagan announces SDI (or “Star Wars”): ground-based and space-based systems to protect the US from attack by strategic nuclear ballistic missiles.

STRATEGIC DEFENSE INITIATIVE
DEPARTMENT OF DEFENSE

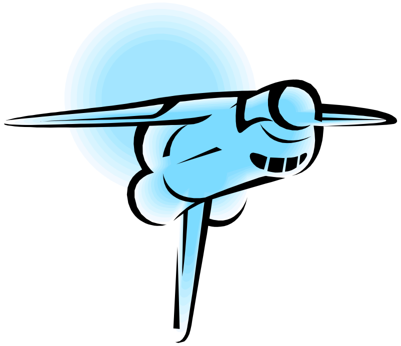


1985

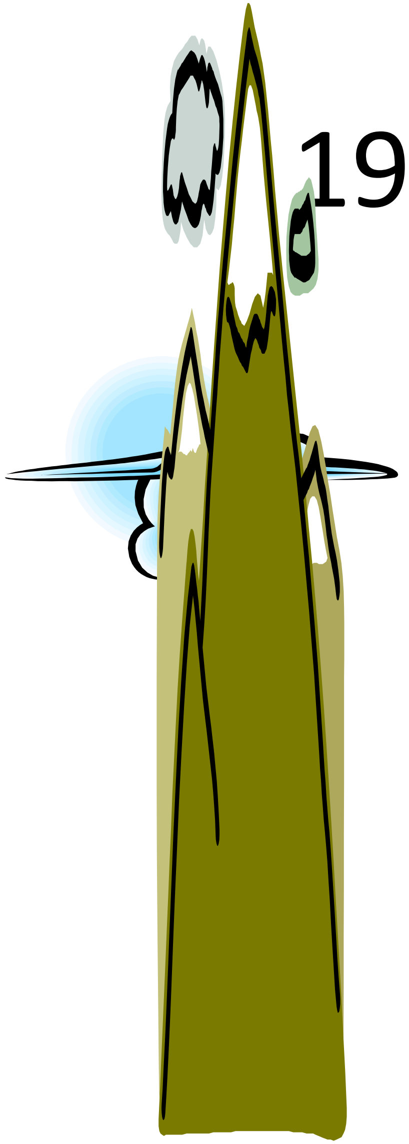


SDI experiment:
The plan

1985



SDI experiment:
The reality



1985

The reality

Attention All Units, Especially Miles and Feet!

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. The San Francisco Chronicle article noted that "the laser experiment was designed to see if a low-energy laser could be used to track a high-speed target about 200 miles above the earth. By its failure yesterday, NASA unwittingly proved what the Air Force already knew -- that the laser would work only on a 'cooperative target' -- and is not likely to be useful as a tracking device for enemy missiles." [This statement appeared in the S.F. Chronicle on 20 June, excerpted from the L.A. Times; the NY Times article on that date provided some controversy on the interpretation of the significance of the problem.] The experiment was then repeated successfully on 21 June (using nautical miles). The important point is not whether this experiment proves or disproves the viability of Star Wars, but rather that here is just one more example of an unanticipated problem in a human-computer interface that had not been detected prior to its first attempted actual use.

NASA Mars Climate Orbiter, 1999

CNN.com

sci-tech> space> story page

exploringmars

in-depth specials

MAIN PAGE
WORLD
U.S.
LOCAL
POLITICS
WEATHER
BUSINESS
SPORTS
TECHNOLOGY
SPACE
HEALTH
ENTERTAINMENT
BOOKS
TRAVEL
FOOD
ARTS & STYLE
NATURE
IN-DEPTH
ANALYSIS
myCNN

Headline News brief
news quiz
daily almanac

MULTIMEDIA:
video
video archive
audio
multimedia showcase
more services

E-MAIL:
Subscribe to one of our news e-mail lists.
Enter your address:

Metric mishap caused loss of NASA orbiter

September 30, 1999
Web posted at: 4:21 p.m. EDT (2021 GMT)


(NASA)

NASA's Climate Orbiter was lost September 23, 1999

In this story:

Metric system used by NASA for many years

Error points to nation's conversion lag

RELATED STORIES, SITES

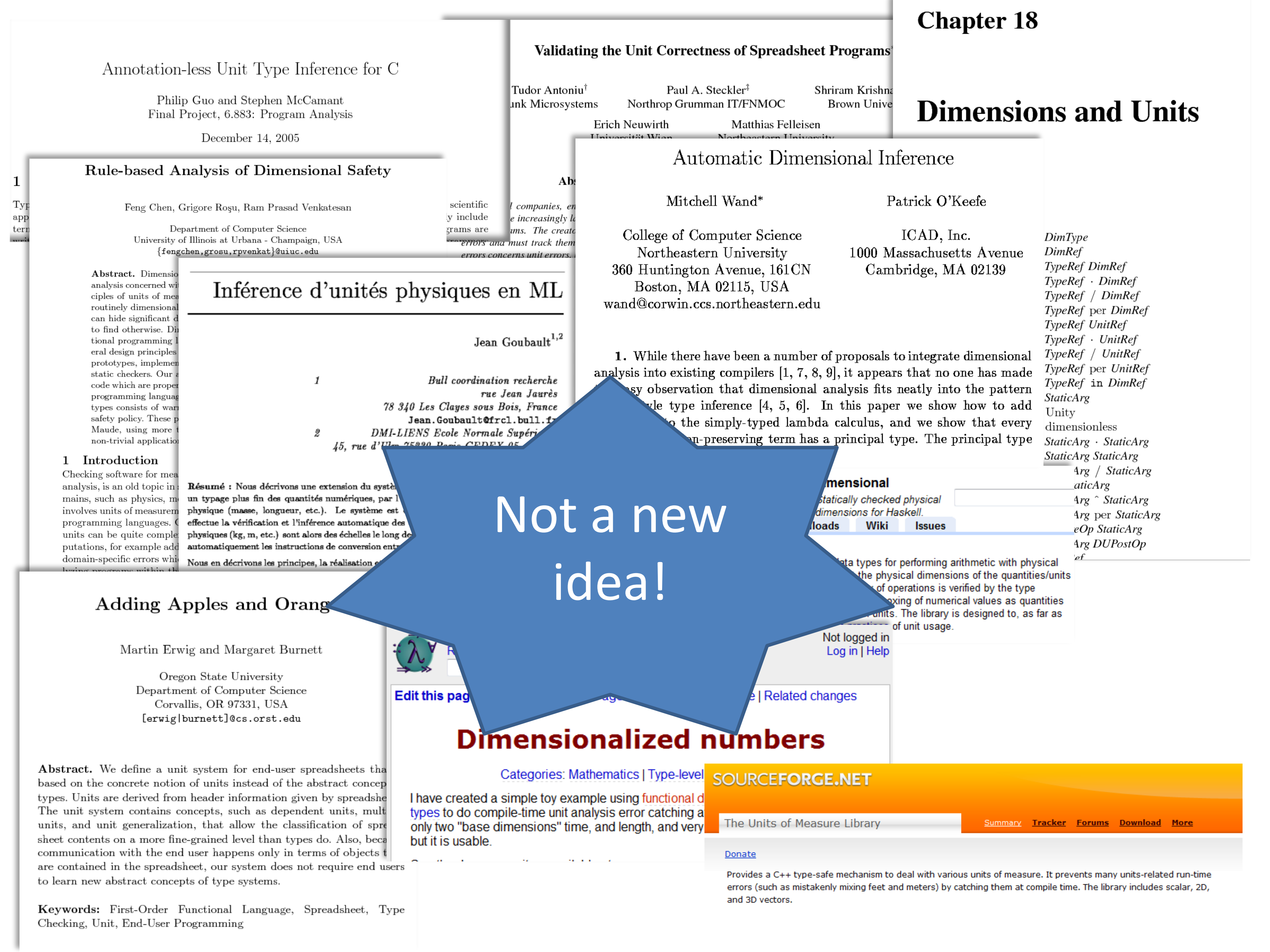
By Robin Lloyd
CNN Interactive Senior Writer

(CNN) -- NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation, according to a review finding released Thursday.

The units mismatch prevented navigation information from transferring between the Mars Climate Orbiter spacecraft team in at Lockheed Martin in Denver and the flight team at NASA's Jet Propulsion Laboratory in Pasadena, California.

Solution

- Check units at development time, by
  - Static analysis, *or*
  - Type checking


Not a new idea!
Annotation-less Unit Type Inference for C
Philip Guo and Stephen McCamant
Final Project, 6.883: Program Analysis
December 14, 2005
Validating the Unit Correctness of Spreadsheet Programs
Paul A. Steckler
Northrop Grumman IT/FNMOC
Erich Neuwirth
Matthias Felleisen
Chapter 18
Dimensions and Units
Rule-based Analysis of Dimensional Safety
Feng Chen, Grigore Roşu, Ram Prasad Venkatesan
Department of Computer Science
University of Illinois at Urbana - Champaign, USA
{fengchen,grosu,rpvenkat}@uiuc.edu
1 Introduction
Inférence d'unités physiques en ML
Jean Goubault
Bull coordination recherche
rue Jean Jaurès
78 340 Les Clayes sous Bois, France
Automatic Dimensional Inference
Mitchell Wand*
Patrick O'Keefe
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@corwin.ccs.northeastern.edu
ICAD, Inc.
1000 Massachusetts Avenue
Cambridge, MA 02139
DimType
DimRef
TypeRef DimRef
TypeRef · DimRef
TypeRef / DimRef
TypeRef per DimRef
TypeRef UnitRef
TypeRef · UnitRef
TypeRef / UnitRef
TypeRef per UnitRef
TypeRef in DimRef
StaticArg
Unity
dimensionless
StaticArg · StaticArg
StaticArg StaticArg
Statically checked physical dimensions for Haskell.
Wiki
Issues
Adding Apples and Oranges
Martin Erwig and Margaret Burnett
Oregon State University
Department of Computer Science
Corvallis, OR 97331, USA
[erwig|burnett]@cs.orst.edu
Keywords: First-Order Functional Language, Spreadsheet, Type Checking, Unit, End-User Programming
Not logged in
Log in | Help
Related changes
Dimensionalized numbers
SOURCEFORGE.NET
The Units of Measure Library
Summary
Tracker
Forums
Download
More
Donate
Provides a C++ type-safe mechanism to deal with various units of measure. It prevents many units-related run-time errors (such as mistakenly mixing feet and meters) by catching them at compile time. The library includes scalar, 2D, and 3D vectors.

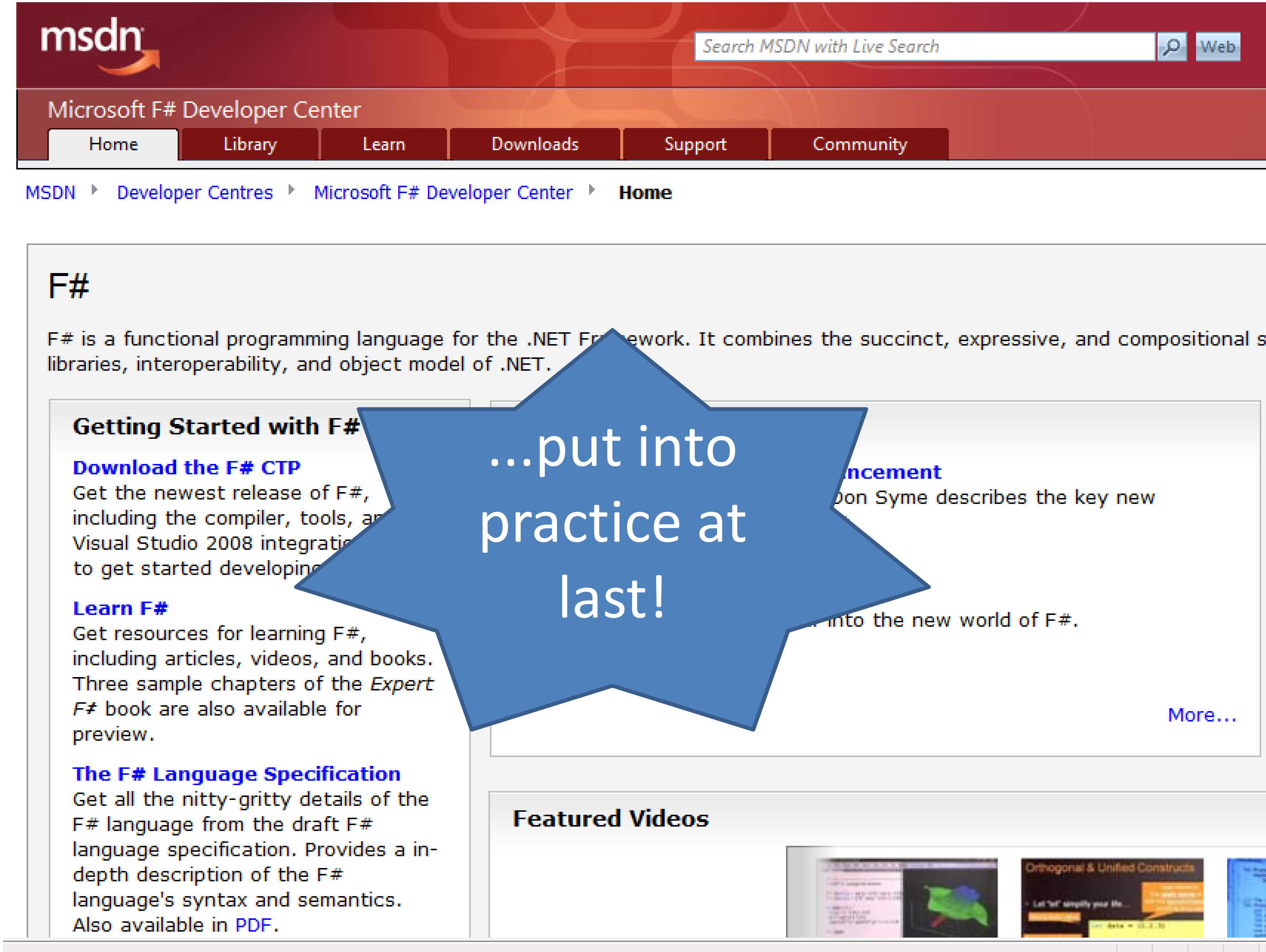
Programming Languages
and
Dimensions

Andrew John Kennedy
St. Catharine's College

Last century...

A dissertation submitted to the University of Cambridge
towards the degree of Doctor of Philosophy

November 1995



...put into practice at last!

Refined types

- Conventional type systems for languages such as Java, C#, ML and Haskell catch many common programming errors
  - Invoking a method that doesn't exist
  - Passing the wrong number of arguments
  - Writing to a read-only field
- So-called *refined* type systems layer additional information onto the underlying types
  - Size-of-array, to catch out-of-bounds access
  - Effect information, to limit scope of side effects
  - Other simple invariants (e.g. balanced-ness of trees)
  - Units-of-measure, to catch unit and dimension errors

Overview

- **Lecture 1: Practice**
  - Gentle tour through units-of-measure in F#
  - Using Visual Studio 2008, or from fsi
  - Demos: physics, Xbox game
- **Lectures 2 and 3: Theory**
  - The type system and type inference algorithm
  - Semantics of units; link to classical dimensional analysis

Units-of-measure design

- ✓ Minimally invasive
  - Type inference, in the spirit of ML & Haskell
    - Annotate literals with units, let inference do the rest
    - But overloading must be resolved
- ✓ Familiar notation, as used by scientists and engineers
- ✓ No run-time cost: units are not carried at runtime
- ✓ Extensible: not just for floats!
- ✗ No support for *dimensions* (classes of units, such as *mass*)
- ✗ No *automatic* unit conversions (but programmer can define them)

Feature Tour in Visual Studio 2008

Summary (1)

Declaring base units

```
[<Measure>] type kg
```

Declaring derived units

```
[<Measure>] type N = kg m/s^2
```

Constants with units

```
let gravity = 9.808<m/s^2>
```

Types with units

```
let newtonsLaw (m:float<kg>) (a:float<m/s^2>) : float<N> = m*a
```

Unit conversions

```
let metresToFeet (l:float<m>) = l * 3.28084<ft/m>
```

Interop

```
let t = 0.001<s> * float stopwatch.ElapsedMilliseconds
```

Dimensionless quantities

```
let calcAngle (arc:float<m>) (radius:float<m>) : float = arc/radius
```

Summary (2)

Unit-polymorphic functions

```
let sqr (x:float<_>) = x*x
```

Polymorphic types

```
let reciprocal : float<'u> -> float<'u^-1> = fun x -> 1.0/x
```

Polymorphic zero

```
let sumSquares xs = List.fold (fun acc x -> sqr x + acc) 0.0<_> xs
```

Application area 1: statistics

Input: list of numbers $[a_1; \ldots; a_n]$

Arithmetic mean $$\mu = \frac{1}{n}\sum_{i=1}^{n} a_i$$

Standard deviation $$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(a_i - \mu)^2$$

Geometric mean $$g = \left(\prod_{i=1}^{n} a_i\right)^{\frac{1}{n}}$$

Unit-polymorphic types?

Application area 2: calculus

- Lots of higher-order functions (called "operators" by mathematicians) e.g.

  differentiate : $(\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$

- These should have units! e.g.

  differentiate : $(\mathbb{R}_u \to \mathbb{R}_v) \to (\mathbb{R}_u \to \mathbb{R}_{v/u})$

Application area 2: calculus

Of course in practice, we use numerical methods:

Differentiation

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

Integration

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

Root-finding

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

Summary (3)

Unit-parameterized types

```
type complex< [<Measure>] 'u> = { re:float<'u>; im:float<'u> }
```

Overloaded static members

```
type vector2< [<Measure>] 'u> = { x:float<'u>; y:float<'u> } with
  static member (+) (a:vector2<'u>, b) = { x = a.x+b.x; y = a.y+b.y }
```

Polymorphic recursion in types

```
type derivs< [<Measure>] 'u, [<Measure>] 'v> =
| Nil
| Cons of (float<'u> -> float<'v>) * derivs<'u, 'v/'u>
```

Polymorphic recursion in functions

```
let rec makeDerivs< [<Measure>] 'u, [<Measure>] 'v>
   (n:int)
   (h:float<'u>)
   (f:float<'u> -> float<'v>) : derivs<'u,'v> =
   if n=0 then Nil else Cons(f, makeDerivs (n-1) h (diff h f))
```

Are units useful?

- We hope so!
  - They really do catch unit errors (e.g. Standard deviation vs variance in machine learning algorithms)
  - They *inform* the programmer, and “correct” types help catch errors e.g.

```
let doublesqr x = sqr x + x
```

```
val doublesqr : float -> float
```

```
let doublesqr x = sqr x + sqr x
```

```
val doublesqr : float<'u> -> float<'u ^ 2>
```

- Lots of “non-standard” applications
  - Finance (units: USD/yr, etc.)
  - Graphics (units: pixels, pt, etc.)
  - Games (units: as in physics!)
  - Search (units: hits/page, etc.)

"I have never understood why physical units didn't get into the typing systems. There is a resistance to richer typing that I think we should challenge." Bill Gates, 2006

Questions?

Solutions to exercises from Lecture 1

Exercise 2.

Q: Why is only zero polymorphic?

A: If other constants were polymorphic we could “cheat” the system e.g. write

```
let cast (x:float<'u>) : float<'v> = 1.0<'v/'u> * x
```

Q: What other special values are polymorphic?

A: Positive and negative infinity.

Solutions to exercises from Lecture 1

Exercise 4.

Implement geometric mean with a polymorphic type.

```
let gmeanAux xs = List.reduce (*) xs ** reciplen xs

let scaleBy (y:float<_>) xs = List.map (fun x -> x*y) xs
let gmean xs = List.hd xs * gmeanAux (scaleBy (1.0/List.hd xs) xs)
```

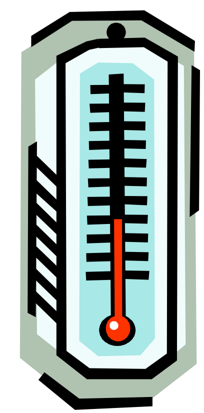
```
val gmean : float<'u> list -> float<'u>

Full name: Tutorial8.gmean
```

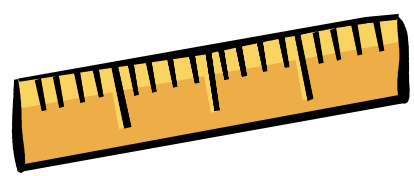
Types for Units-of-Measure: Theory and Practice

Lecture 2: Types and Type Inference

Andrew Kennedy

Microsoft Research, Cambridge





Polymorphic type inference

- Type systems of SML, Caml, Haskell, F# are all based on type inference for let polymorphism
  - Old technology! *A theory of type polymorphism in programming*, Robin Milner, 1978.
  - Polymorphic types (type *schemes*) are introduced by let bindings, lambda bindings are non-polymorphic

```
let pair =
  let id = fun y -> y in (id 5, id true)
```

val id : ('a -> 'a)

```
let pair =
  let applyFun f = (f 5, f true) in applyFun (fun y -> y)
```

This expression has type bool but is here used with type int

Polymorphic type inference, cont.

- Hundreds of papers have extended this system
  1. To support polymorphism for λ e.g. ML^F, HMF, FPH, giving ML the expressiveness of System F
  2. To add features such as GADTs, $\exists$
  3. To support polymorphism over other entities e.g. records ("row polymorphism") or effects
- Units-of-measure are an example of 3.

Units as types?

- Can't we just code up units-of-measure as types? E.g. Acceleration is just

```
acc : float<UProd<m,UInv<UProd<s,s>>>>
```

- No! This doesn't respect properties of units e.g.

```
let totalAcc = 2.0<m s^-2> + 3.0<s^-2 m>
```

Need commutativity to make units match

```
let distance = 2.0<m> + 3.0<m s^-1> * 4.0<s>
```

Need inverses and identity to make units match

Grammar for units

Base units e.g. kg

No units (dimensionless)

Unit expressions

Inverse

$$u, v, w ::= b \mid \alpha \mid \mathtt{1} \mid u \mathbin{\mathtt{*}} v \mid u\,\mathtt{\hat{}\,\text{-}1}$$

Unit variables e.g. ‘u

Product of units

Unit quotient

$$u/v \quad = \quad u \mathbin{\mathtt{*}} v\,\mathtt{\hat{}\,\text{-}1}$$

Integer powers of units

$$u\,\mathtt{\hat{}}\,n \quad = \quad \begin{cases} u \mathbin{\mathtt{*}} u\,\mathtt{\hat{}}(n-1) & \text{if } n > 0, \\ \mathtt{1} & \text{if } n = 0, \\ u\,\mathtt{\hat{}\,\text{-}1} \mathbin{\mathtt{*}} u\,\mathtt{\hat{}}(n+1) & \text{if } n < 0. \end{cases}$$

Equations for units

Equivalence relation

$$\frac{}{u =_U u}\,(\text{refl}) \qquad \frac{u =_U v}{v =_U u}\,(\text{sym}) \qquad \frac{u =_U v \quad v =_U w}{u =_U w}\,(\text{trans})$$

Congruence

$$\frac{u =_U v}{u\texttt{\^{}-1} =_U v\texttt{\^{}-1}}\,(\text{cong1}) \qquad \frac{u =_U v \quad u' =_U v'}{u * u' =_U v * v'}\,(\text{cong2})$$

Abelian group axioms

$$\frac{}{u * \texttt{1} =_U u}\,(\text{id}) \qquad \frac{}{(u * v) * w =_U u * (v * w)}\,(\text{assoc})$$

$$\frac{}{u * v =_U v * u}\,(\text{comm}) \qquad \frac{}{u * u\texttt{\^{}-1} =_U \texttt{1}}\,(\text{inv})$$

Equational theories

- $=_U$ is an example of an *equational theory*
- Other examples:
  - AC (just associativity and commutativity)
  - AC1 (add identity, to get commutative monoids)
  - ACI (add idempotence)
  - BR (boolean rings)
- For units we have AG, the theory of Abelian groups

The case of the vanishing variable

- Write *vars*(*u*) for the set of variables syntactically occurring in unit expression *u* e.g.

$$vars((\alpha \ \texttt{*}\ \beta)\ \texttt{*}\ (\texttt{kg}\ \texttt{*}\ \beta\texttt{\^{}-1})) = \{\alpha, \beta\}$$

- Our theory (AG) is *non-regular*, meaning that

$$u =_U v \not\Rightarrow vars(u) = vars(v)$$

- This is the source of many challenges!
  - For example, we have to be careful when saying "α not free in ..."

Deciding equations

How to check if equation

$$u =_U v$$

is valid?

1. Put unit expressions u and v into *normal form*:

Non-zero exponents

$$\alpha_1{}^{x_1} * \cdots * \alpha_m{}^{x_m} * b_1{}^{y_1} * \cdots * b_n{}^{y_n}$$

Variables and base units
ordered alphabetically

2. Check equality syntactically.

Normal form example

- Unit expression:

$$(\alpha * \beta) * ((\mathtt{kg} * \beta\mathtt{\hat{}-1}) * \alpha)$$

- Normal form:

$$\alpha^2 * \mathtt{kg}$$

Solving equations

- Deciding equations gives us type *checking*.
- For type *inference*, we need to *solve* equations.

```
> let area = 20.0<m^2>;;

val area : float<m ^ 2> = 20.0

> let f (y:float<_>) = area + y*y;;

val f : float<m> -> float<m ^ 2>
```

- Here, the compiler generates a fresh unit variable α for the units of y, then solves the equation

$$\alpha\texttt{\^{}}2 =_U \texttt{m\^{}2}$$

Multiple solutions

- In general, there may be many ways to solve e.g.

$$\alpha * \beta =_U \mathtt{m\hat{}2}$$

- This has (at least) three *ground* solutions

$$\{\alpha := \mathtt{m}, \beta := \mathtt{m}\} \quad \{\alpha := \mathtt{m\hat{}2}, \beta := \mathtt{1}\} \quad \{\alpha := \mathtt{1}, \beta := \mathtt{m\hat{}2}\}$$

- But all solutions are subsumed by a non-ground, `parametric solution':

$$\{\alpha := \beta\mathtt{\hat{}\text{-}1} * \mathtt{m\hat{}2}\}$$

Equational unification

- Solving equations with respect to an equational theory E is called *equational unification.*
  - Given two terms t and u, find substitution S such that $S(t) =_E S(u)$
- Syntactic unification is the basis of ML type inference.
  - *principal types* property stems from the fact that if two terms are unifiable then there exists a single *most general unifier* that subsumes all others
- Not all equational theories enjoy this property. Many theories require multiple substitutions to express all solutions.

A good book:
"Term Rewriting and *All That*" by
Baader and Nipkow

AG unification

- For units, a unifier of two unit expressions u_1 and u_2 is a substitution S on unit variables such that S(u_1)$=_U$ S(u_2)
- Fortunately, Abelian Group unification is
  - *unitary* (single most general unifiers exist with respect to the equational theory), and
  - *decidable* (algorithm is a variation of Gaussian elimination)
- First, notice that

$$u =_U v \text{ if and only if } u \texttt{ * } v\texttt{^-1} =_U \texttt{1}$$

- So we can reduce the problem to unifying a unit expression against 1.

Unification algorithm

$Unify(u, v) = UnifyOne(u * v\texttt{\^{}-1})$

$UnifyOne(u) =$

let $u = \alpha_1^{x_1} * \cdots * \alpha_m^{x_m} * b_1^{y_1} * \cdots * b_n^{y_n}$ where $|x_1| \leqslant |x_2|, \cdots, |x_m|$

in

if $m = 0$ and $n = 0$ then id

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

if $m = 1$ and $x_1 \mid y_i$ for all i then $\{\alpha_1 \mapsto b_1^{-y_1/x_1} * \cdots * b_m^{-y_n/x_1}\}$

if $m = 1$ otherwise then fail

else $S_2 \circ S_1$ where

$S_1 = \{\alpha_1 \mapsto \alpha_1 * \alpha_2^{-\lfloor x_2/x_1 \rfloor} * \cdots * \alpha_m^{-\lfloor x_m/x_1 \rfloor} * b_1^{-\lfloor y_1/x_1 \rfloor} * \cdots * b_n^{-\lfloor y_n/x_1 \rfloor}\}$

$S_2 = UnifyOne(S_1(u))$

Unification in action

$$\alpha^3 * \beta^2 =_U \mathrm{kg}^6$$

$\Downarrow$ rewrite

$$\alpha^3 * \beta^2 * \mathrm{kg}^{-6} =_U 1$$

$\Downarrow$ apply $\{\beta := \beta * \alpha^{-1} * \mathrm{kg}^3\}$

$$\alpha * \beta^2 =_U 1$$

$\Downarrow$ apply $\{\alpha := \alpha * \beta^{-2}\}$

$$\alpha =_U 1$$

$\Downarrow$ apply $\{\alpha := 1\}$

$$1 =_U 1$$

Success!

Correctness of Unification

- We can prove the following:

$$\text{(Soundness) If } \mathit{Unify}(u, v) = S \text{ then } S(u) =_U S(v).$$
$$\text{(Completeness) If } S(u) =_U S(v) \text{ then } \mathit{Unify}(u, v) \preceq_U S.$$

"is more general than"

Grammar for types

Type variables

Function types

$$\tau ::= \alpha \mid \texttt{float<}u\texttt{>} \mid \tau \ \texttt{->} \ \tau$$

Type expressions

Unit-parameterized floats

Equations for types

- Obvious extension from units, such that

$$\texttt{float<}u\texttt{>} =_U \texttt{float<}v\texttt{>} \text{ iff } u =_U v$$

Unification for types

$$
\begin{array}{rcl}
TUnify(\alpha, \alpha) & = & id \\
TUnify(\alpha, \tau) = TUnify(\tau, \alpha) & = & \begin{cases} \textsf{fail} & \text{if } \alpha \text{ in } \tau \\ \{\alpha := \tau\} & \text{otherwise.} \end{cases} \\
TUnify(\texttt{float<}u\texttt{>}, \texttt{float<}v\texttt{>}) & = & Unify(u, v) \\
TUnify(\tau_1 \texttt{ -> } \tau_2, \tau_3 \texttt{ -> } \tau_4) & = & S_2 \circ S_1 \\
 & & \text{where } S_1 = TUnify(\tau_1, \tau_3) \\
 & & \text{and } S_2 = TUnify(S_1(\tau_2), S_1(\tau_4))
\end{array}
$$

Just ordinary unification with unification for units plugged in!

Type schemes

- Formally, a type scheme is a type in which (some) unit variables are quantified:

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n . \tau$$

- A type scheme *instantiates* to a type by replacing its quantified variables by unit expressions:

$$\forall \alpha_1, \ldots, \alpha_n . \tau \preceq \tau' \text{ if } \tau' = \{\alpha_1 := u_1, \ldots, \alpha_n := u_n\}\tau \text{ for some } u_1, \ldots, u_n$$

Type scheme instantiation, cont.

- We write

$$\sigma \preceq_U \tau \text{ if } \sigma \preceq_U \tau' \text{ and } \tau' =_U \tau \text{ for some } \tau'.$$

- Surprising example:

$$\forall \alpha.\texttt{float<}\alpha \texttt{ * kg> -> float<}\alpha \texttt{ * kg>} \preceq_U \texttt{float<1> -> float<1>}$$

Type system

- Essentially the same as ML, with one new rule:

$$\frac{V;\Gamma \vdash e : \tau_1}{V;\Gamma \vdash e : \tau_2}\ \tau_1 =_U \tau_2$$

- This just says that typing respects "rules of units"
- Rule for variables just instantiates the type scheme of the variable:

$$\frac{}{V;\Gamma, x{:}\sigma \vdash x : \tau}\ \sigma \preceq \tau$$

Type Inference Algorithm

- Can we just plug in our new unification algorithm into usual ML inference algorithm?
- *Not quite*. We get soundness, but not completeness – i.e. some legal programs are rejected.
  - This is because just using "free unit variables" in the rule for let is not sufficient.
  - Can be fixed by "normalizing" the type environment before generalizing unit variables. For details, see my thesis.

Correctness of Inference Algorithm

- Suppose algorithm *Infer*(*e*) produces a type scheme for expression *e*. We can prove the following:

$$\text{(Soundness) If } \mathit{Infer}(e) \preceq_U \tau \text{ then } \vdash e : \tau$$
$$\text{(Completeness) If } \vdash e : \tau \text{ then } \mathit{Infer}(e) \preceq_U \tau.$$

Type Scheme Equivalence

- Two type schemes are equivalent if they instantiate to the same set of types, up to the equational theory:

$$\sigma_1 \cong_U \sigma_2 \text{ iff } (\forall \tau. \sigma_1 \preceq_U \tau \Leftrightarrow \sigma_2 \preceq_U \tau)$$

- For vanilla ML, this just amounts to renaming quantified type variables or removing redundant quantifiers.
- For F# with units, there are many non-trivial equivalences. E.g.

$$/ : \forall \alpha\beta.\texttt{float<}\alpha\texttt{>} \rightarrow \texttt{float<}\beta\texttt{>} \rightarrow \texttt{float<}\alpha \texttt{ * } \beta\texttt{\^{}-1>}$$

$$/ : \forall \alpha\beta\gamma.\texttt{float<}\gamma \texttt{ * } \alpha\texttt{>} \rightarrow \texttt{float<}\beta\texttt{>} \rightarrow \texttt{float<}\gamma \texttt{ * } \alpha \texttt{ * } \beta\texttt{\^{}-1>}$$

$$/ : \forall \alpha\beta.\texttt{float<}\alpha\texttt{\^{}-1>} \rightarrow \texttt{float<}\beta\texttt{\^{}-1>} \rightarrow \texttt{float<}\alpha\texttt{\^{}-1 * } \beta\texttt{>}$$

$$/ : \forall \alpha\beta.\texttt{float<}\alpha \texttt{ * } \beta\texttt{>} \rightarrow \texttt{float<}\alpha\texttt{>} \rightarrow \texttt{float<}\beta\texttt{>}$$

$$/ : \forall \alpha\beta.\texttt{float<}\alpha\texttt{>} \rightarrow \texttt{float<}\beta\texttt{\^{}-1>} \rightarrow \texttt{float<}\alpha \texttt{ * } \beta\texttt{>}$$

$$/ : \forall \alpha\beta.\texttt{float<}\gamma \texttt{ * } \alpha\texttt{>} \rightarrow \texttt{float<}\gamma \texttt{ * } \beta\texttt{>} \rightarrow \texttt{float<}\alpha \texttt{ * } \beta\texttt{\^{}-1>}$$

Simplifying type schemes

- We can show that two type schemes are equivalent iff there is an invertible substitution on the bound variables that maps between them (this is a “change of basis”)
- *Idea*: compute such a substitution that puts a type scheme in some kind of preferred “normal form” for printing. Desirable properties:
  - No redundant bound or free variables (so number of variables = number of “degrees of freedom”)
  - Minimize size of exponents
  - Use positive exponents if possible
  - Unique up to renaming
- Such a form does exist, and corresponds to Hermite Normal Form from algebra
  - Pleasant side-effect: deterministic ordering on variables in type

Simplification in action

$$\forall\alpha\beta.\texttt{float<}\gamma * \alpha\texttt{>} \rightarrow \texttt{float<}\gamma * \beta\texttt{\^{}-1>} \rightarrow \texttt{float<}\alpha * \beta\texttt{>}$$

$$\Downarrow \{\alpha := \alpha * \gamma\texttt{\^{}-1}\}$$

$$\forall\alpha\beta.\texttt{float<}\alpha\texttt{>} \rightarrow \texttt{float<}\gamma * \beta\texttt{\^{}-1>} \rightarrow \texttt{float<}\gamma\texttt{\^{}-1} * \alpha * \beta\texttt{>}$$

$$\Downarrow \{\beta := \beta\texttt{\^{}-1}\}$$

$$\forall\alpha\beta.\texttt{float<}\alpha\texttt{>} \rightarrow \texttt{float<}\gamma * \beta\texttt{>} \rightarrow \texttt{float<}\gamma\texttt{\^{}-1} * \alpha * \beta\texttt{\^{}-1>}$$

$$\Downarrow \{\beta := \beta * \gamma\texttt{\^{}-1}\}$$

$$\forall\alpha\beta.\texttt{float<}\alpha\texttt{>} \rightarrow \texttt{float<}\beta\texttt{>} \rightarrow \texttt{float<}\alpha * \beta\texttt{\^{}-1>}$$

Technical summary

- Grammar for units
- Equational theory of units (AG) with
  - decidable equality
  - decidable and unitary unification
- Change of basis algorithm, used for
  - type scheme simplification
  - generalization (not discussed today)
- Main Result: *principal types*

Executive summary

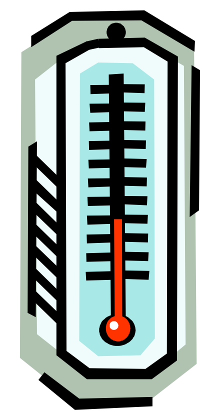
- Units-of-measure types occupy a “sweet spot” in the space of type systems
  - Type system is easy to understand for novices (just high-school “rules of units”)
  - Types have a simple form (e.g. no constraints, bounds)
  - Types don’t intrude (there is rarely any need for annotation)
  - Behind the scenes, inference is non-trivial but practical

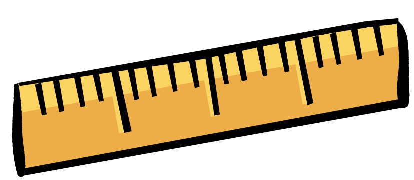
Questions?

Types for Units-of-Measure: Theory and Practice

Lecture 3: Semantics of Units

Andrew Kennedy
Microsoft Research, Cambridge





Type safety

- "Well-typed programs don't go wrong" (Milner, 1978)
  - They don't dump core or throw MissingMethodException
  - Formalized by adding a **wrong** value to the semantics (e.g. "applying" an integer to a value evaluates to **wrong**) and then showing that well-typed expressions don't evaluate to **wrong**
  - These days usually formalized as *syntactic type soundness*:
    - *Preservation*: if e:τ and e reduces in some number of steps to e', then e':τ, and
    - *Progress*: if e:τ then either e is a final value (constant, lambda, etc) or e reduces to some e' (i.e. it doesn't "get stuck")

Units going wrong?

- What "goes wrong" if a program contains a unit error?
  - Nothing!
  - Unless runtime values are instrumented with their units-of-measure. But that would be cheating (runtime values don't have units)!
  - We need a different notion of "going wrong"
- In Nature, units do not go wrong! Instead, physical laws are *invariant under changes to the unit system*.
- So in Programming, the *real* essence of unit correctness is the invariance of program behaviour under change to units.

Units going right

```
let checkin(baggage:float<lb>, allowance:float<lb>)
 = if baggage > allowance then printf “Bags exceed limit”

checkin(88.0<lb>, 44.0<lb>)
```

Metricate

```
let checkin(baggage:float<kg>, allowance:float<kg>)
 = if baggage > allowance then printf “Bags exceed limit”

checkin(40.0<kg>, 20.0<kg>)
```

Same behaviour:
passenger is turned away!

Units going wrong

```
let checkin(baggage:float<lb>, allowance:float<cm>)
 = if baggage > allowance then printf “Bags exceed limit”

checkin(88.0<lb>, 55.0<cm>)
```

Metricate

```
let checkin(baggage:float<kg>, allowance:float<cm>)
 = if baggage > allowance then printf “Bags exceed limit”

checkin(40.0<lb>, 55.0<cm>)
```

Different behaviour!

Polymorphic units going wrong?

- Suppose we have a function

```
foo : float<'u> -> float<'u^2>
```

- What does it mean for this function to “go wrong”? We surely know it when we see it:

```
let foo (x:float<'u>) = x*x*x
```

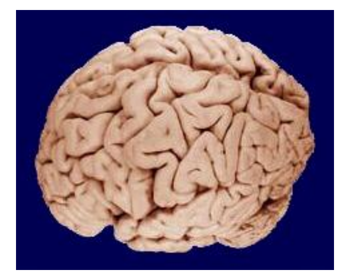
- But what if it’s implemented by

```
fmul     st(1),st
fmul     st(1),st
fld      DWORD PTR [esp]
fxch     st(1)
fmulp    st(2),st
fsub     st,st(1)
```

Machine code

FPGA

analogue computer



human computer

Polymorphic units going right

- Again: the essence of unit correctness is *invariance under scaling*. For

  $$\texttt{foo} : \forall\alpha.\texttt{float<}\alpha\texttt{> -> float<}\alpha^2\texttt{>}$$

  this amounts to the property

  $$\forall x, \texttt{foo}(k * x) = k^2 * \texttt{foo}(x)$$

  for any positive “scale factor” k.

- Suppose that we discovered that

  $$\texttt{foo}(2) = 8 \qquad \texttt{foo}(4) = 64$$

  Then we would know that foo’s type is “lying”!

Representation Independence

- Invariance under scaling is an example of *representation independence.*
  - We can change the data representation without changing the behaviour of a program
  - Applied to polymorphic functions, this is known as *parametricity* (Reynolds, 1983)
- Example for ordinary polymorphism: if

$$\mathtt{bar} : \forall \alpha . \alpha \to \alpha \times \alpha$$

then for any "change of representation" function *f*,

$$\forall x, \mathtt{bar}(f(x)) = \langle f, f \rangle (\mathtt{bar}(x))$$

Parametricity for units

- First define a scaling environment ψ: a map from unit variables to positive scale factors. Extend to unit expressions:

$$\begin{array}{rcl} \psi(\mathtt{1}) & = & 1 \\ \psi(u \ast v) & = & \psi(u) \cdot \psi(v) \\ \psi(u\mathtt{\hat{}-1}) & = & 1/\psi(u) \end{array}$$

- Now define a binary "logical" relation over values, indexed by types and type schemes:

$$\begin{array}{rcl} x \sim^{\psi}_{\mathtt{float<}u\mathtt{>}} y & \Leftrightarrow & y = \psi(u) \ast x \\ f \sim^{\psi}_{\tau_1 \mathtt{->} \tau_2} g & \Leftrightarrow & \forall xy, x \sim^{\psi}_{\tau_1} y \Rightarrow f(x) \sim^{\psi}_{\tau_2} g(y) \\ x \sim_{\forall \overline{\alpha}.\tau} y & \Leftrightarrow & \forall \overline{k}, x \sim^{\{\overline{\alpha} \mapsto \overline{k}\}}_{\tau} y \end{array}$$

- Now we can prove the "fundamental theorem":

$$\vdash a : \sigma \quad \Rightarrow \quad a \sim_{\sigma} a$$

Scaling theorems for free

- First consequence of parametricity: given just the type of a function, we can obtain “theorems for free”

Example 1. If

$$f : \forall \alpha\beta.\texttt{float<}\alpha\texttt{> -> float<}\beta\texttt{> -> float<}\alpha \texttt{ * } \beta\texttt{\^{}-1>}$$

then

$$\forall k_1, k_2 > 0, f\ (k_1 * x)\ (k_2 * y) = (k_1/k_2) * f\ x\ y$$

Scaling theorems for free

Example 2. If

$$\texttt{diff} : \forall\alpha\beta.\texttt{float<}\alpha\texttt{>} \quad \texttt{->} \left(\texttt{float<}\alpha\texttt{>} \texttt{ -> } \texttt{float<}\beta\texttt{>}\right)$$
$$\texttt{->} \left(\texttt{float<}\alpha\texttt{>} \texttt{ -> } \texttt{float<}\beta * \alpha\texttt{\^{}-1>}\right)$$

then

$$\forall k_1, k_2 > 0, \texttt{diff}\ h\ f\ x = \frac{k_2}{k_1} * \texttt{diff}\left(\frac{h}{k_1}\right)\left(\lambda x.\,\frac{f(x * k_1)}{k_2}\right)\left(\frac{x}{k_1}\right)$$

Zero

- Why is zero polymorphic in its units? Answer: because it is invariant under scaling:

$$\forall k, k * 0 = 0$$

- This holds for no other values, so they cannot be polymorphic.

Definability

- Parametricity can also be used to show that some types are *uninhabited*, or at least contain only “boring” functions.
- Example for ordinary polymorphism: no functions have type

$$\forall \alpha\beta.\alpha \texttt{ -> } \beta$$

- For units, we can show that given only basic arithmetic (+, -, *, /, <) there are no interesting functions with type

$$\forall \alpha.\texttt{float<}\alpha^2\texttt{>} \texttt{ -> } \texttt{float<}\alpha\texttt{>}$$

- Exercise: intuitively, why is this? Hint: try using Newton’s method to compute square root, with polymorphic units.

Type isomorphisms

- We write $\tau_1 \cong \tau_2$
  if the types are *isomorphic*, meaning

$$\exists i : \tau_1 \to \tau_2, j : \tau_2 \to \tau_1 \text{ such that } j \circ i = id \text{ and } i \circ j = id$$

- Examples:

$$\texttt{int * bool} \cong \texttt{bool * int}$$

$$\texttt{int * bool -> unit * int} \cong \texttt{bool * int -> int}$$

$$\texttt{int} \cong \forall\alpha.(\alpha\ \texttt{-> int})\ \texttt{-> int}$$

$$\texttt{int * bool} \cong \forall\alpha.(\texttt{int -> bool ->}\ \alpha)\ \texttt{->}\ \alpha$$

Need parametricity
to prove these two!

A surprising isomorphism

- Assuming positive values only:

$$\forall\alpha.\texttt{float<}\alpha\texttt{>}\ \texttt{->}\ \texttt{float<}\alpha\texttt{>} \cong \texttt{float<1>}$$

Proof.

$$i : (\forall\alpha.\texttt{float<}\alpha\texttt{>}\ \texttt{->}\ \texttt{float<}\alpha\texttt{>}) \rightarrow \texttt{float<1>} = \lambda f.f(1)$$
$$j : \texttt{float<1>} \rightarrow (\forall\alpha.\texttt{float<}\alpha\texttt{>}\ \texttt{->}\ \texttt{float<}\alpha\texttt{>}) = \lambda x.\lambda y.y * x$$

$$\begin{array}{ll}
i \circ j & \\
= \lambda x.i(j(x)) & \text{(composition)} \\
= \lambda x.i(\lambda y.y * x) & \text{(applying } j\text{)} \\
= \lambda x.1.0 * x & \text{(applying } i\text{)} \\
= \lambda x.x & \text{(arithmetic)}
\end{array}$$

$$\begin{array}{ll}
j \circ i & \\
= \lambda f.j(i(f)) & \text{(composition)} \\
= \lambda f.j(f(1.0)) & \text{(applying } i\text{)} \\
= \lambda f.\lambda y.y * f(1.0) & \text{(applying } j\text{)} \\
= \lambda f.\lambda y.fy & \text{(scaling invariance)} \\
= \lambda f.f & \text{(eta)}
\end{array}$$

A surprising isomorphism

- Assuming positive values only:

$$\forall\alpha.\texttt{float<}\alpha\texttt{> -> float<}\alpha\texttt{>} \cong \texttt{float<1>}$$

Informally, consider what functions have type

$$\forall\alpha.\texttt{float<}\alpha\texttt{> -> float<}\alpha\texttt{>}$$

- They *must* be equivalent to

$$\lambda x.k \texttt{ * } x \text{ for some } k : \texttt{float<1>}$$

Another surprising isomorphism

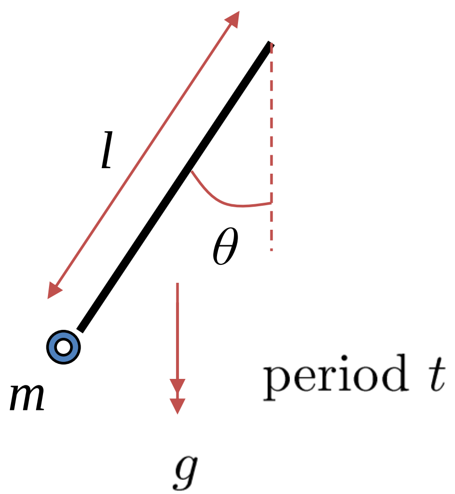
- Assuming positive values only:

$$\forall\alpha.\texttt{float<}\alpha\texttt{>}\ \texttt{->}\ \texttt{float<}\alpha\texttt{>}\ \texttt{->}\ \texttt{float<}\alpha\texttt{>} \cong \texttt{float<1>}\ \texttt{->}\ \texttt{float<1>}$$

Exercise: prove it!

Dimensional analysis

- Old idea (Buckingham): given some physical system with known variables but unknown equations, use the dimensions of the variables to determine the form of the equations. Example: a pendulum.



$$t = \sqrt{\frac{l}{g}}\phi(\theta) \text{ for some } \phi$$

Worked example

- Pendulum has five variables:

mass	m	M
length	l	L
gravity	g	LT^{-2}
angle	θ	none
time period	t	T

Think of M L T as (arbitrary) units for Mass Length and Time

- Assume some relation $f(m, l, g, \theta, t) = 0$
- Then by scaling invariance $f(Mm, Ll, LT^{-2}g, \theta, Tt) = 0$ for any "scale factors" M,L,T
- Let $M=1/m$, $L=1/l$, $T=1/t$, so $f(1,1,t^2g/l, \theta, 1) = 0$
- Assuming a functional relationship, we obtain

$$t = \sqrt{\frac{l}{g}}\phi(\theta) \text{ for some } \phi$$

Dimensional analysis, formally

Pi Theorem
Any dimensionally-invariant relation

$$f(x_1,\ldots,x_n)=0$$

for dimensioned variables $x_1,\ldots,x_n$ whose dimension exponents are given by an m by n matrix A is equivalent to some relation

$$g(P_1,\ldots,P_{n-r})=0$$

where r is the rank of A and $P_1,\ldots,P_{n-r}$ are dimensionless products of powers of $x_1,\ldots,x_n$.

Proof: Birkhoff.

Primitive isomorphisms

- We can classify isomorphisms:

$$
\begin{array}{rcll}
\tau_1 \to \cdots \to \tau_i \to \cdots \tau_j \to \cdots \to \tau_n \to \tau & \cong & \tau_1 \to \cdots \to \tau_j \to \cdots \tau_i \to \cdots \to \tau_n \to \tau & \mathrm{C1} \\
\texttt{float<}u\texttt{>} \to \tau & \cong & \texttt{float<}u\texttt{\^{}-1>} \to \tau & \mathrm{C2} \\
\texttt{float<}v\texttt{>} \to \texttt{float<}u\texttt{>} \to \tau & \cong & \texttt{float<}v \ast u^z\texttt{>} \to \texttt{float<}u\texttt{>} \to \tau & \mathrm{C3} \\
\\
\forall \alpha_1 \cdots \alpha_n.\tau & \cong & \forall \alpha_1 \cdots \alpha_n.\{\alpha_i := \alpha_j, \alpha_j := \alpha_i\}\tau & \mathrm{R1} \\
\forall \alpha.\tau & \cong & \forall \alpha.\{\alpha := \alpha\texttt{\^{}-1}\}\tau & \mathrm{R2} \\
\forall \beta\alpha.\tau & \cong & \forall \beta\alpha.\{\beta := \beta \ast \alpha^z\}\tau & \mathrm{R3} \\
\\
\forall \alpha.\texttt{float<}\alpha^z\texttt{>} \to \texttt{float<}\alpha^{y \cdot z} \ast u\texttt{>} & \cong & \texttt{float<}u\texttt{>} \qquad (\alpha \text{ not free in } u) & \mathrm{D}
\end{array}
$$

- These can be composed to build isomorphisms such as

$$\forall \alpha.\texttt{float<}\alpha\texttt{>} \to \texttt{float<}\alpha\texttt{>} \to \texttt{float<}\alpha\texttt{>} \cong \texttt{float<1>} \to \texttt{float<1>}$$

Pi Theorem, for first-order types

- Suppose

$$\tau = \forall\alpha_1,\ldots,\alpha_m.\texttt{float<}u_1\texttt{>} \to \cdots \to \texttt{float<}u_n\texttt{>} \to \texttt{float<}u_0\texttt{>}.$$

  Let A be $m \times n$ matrix of exponents of variables in $u_1,\ldots,u_n$. Let B be m-vector of exponents in u_0. If AX=B is solvable, then

$$\tau \cong \texttt{float<1>} \to \overset{n-r}{\cdots} \to \texttt{float<1>} \to \texttt{float<1>}$$

  where r is the rank of A.

- *Proof*. Iteratively apply primitive isomorphisms C1-C3 and R1-R3 that correspond to column and row operations on matrix A, producing the *Smith Normal Form* of A. Then apply r instances of isomorphism D and we're done!

Summary

- The semantics of units is all about “invariance under scaling”
  - Program behaviour is invariant under changes to base units
  - Polymorphic functions have “scaling properties” derived from their types
- Nice connection to classical results from dimensional analysis
- This “extensional” approach to safety can be applied in other domains too e.g. “high-level types for low-level programs”