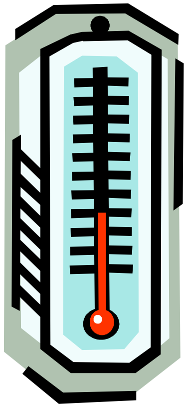
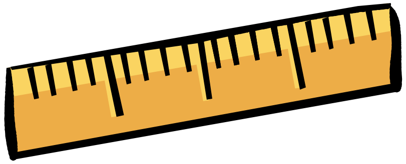
Formalizing an Extensional Semantics for Units-of-Measure

Andrew Kennedy
Microsoft Research
Cambridge





Introduction

- The F# programming language supports checking and inference of units-of-measure

```
let speedOfImpact : float<m/s> =
  sqrt (2.0 * gravityOnEarth * heightOfMyOfficeWindow)

val variance : float<'u> list -> float<'u ^ 2>

val areaUnderCurve :
   (float<'u> -> float<'v>) * float<'u> * float<'u> ->
   float<'u 'v>
```

- Type inference works well, with principal types and a practical algorithm
  - Come to my talk at the ML workshop (9am Sunday)
  - Visit http://blogs.msdn.com/andrewkennedy
  - Download F# Community Technology Preview from http://msdn.microsoft.com/fsharp

Introduction

- Today's talk: the *semantics* of units-of-measure
  - What does it mean for unit-incorrect programs *to go wrong*?
  - How do unit-polymorphic functions *behave*?
  - What is the analogue of classical results from dimensional analysis?
- And: *formalize* the semantics in Coq.

Going wrong, intensionally

- "Well-typed programs don't go wrong" (Milner, 1978)
  - They don't dump core or throw MissingMethodException
  - Originally formalized by adding a **wrong** value to the semantics (e.g. applying an integer as if it were a function reduces to **wrong**) and then showing that well-typed expressions don't reduce 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")
- 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!

Going wrong, extensionally

- Claim: the essence of unit correctness is *invariance of program behaviour under scaling*. E.g.

```
let good(x:float<kg>, y:float<kg>)
 = if x<y then print “!”
in good(1.0<kg>, 2.0<kg>)
```

```
let bad(x:float<kg>, y:float<s>)
 = if x<y then print “!”
in bad(1.0<kg>, 2.0<s>)
```

Convert kg into lb

```
let good(x:float<lb>, y:float<lb>)
 = if x<y then print “!”
in good(2.2<lb>, 4.4<lb>)
```

```
let bad(x:float<lb>, y:float<s>)
 = if x<y then print “!”
in bad(2.2<lb>, 2.0<s>)
```

- Compare: invariance of physical laws under scaling

Polymorphism, extensionally

- How do we know that we can safely assign a type

  `foo : float<’u> -> float<’u^2>`

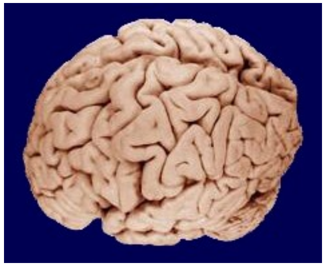
  ?
- If `foo` is implemented in F#, then safety follows from soundness of typing rules
- But what if it’s implemented by

```
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

Polymorphism, extensionally

- Claim: the essence of unit-polymorphism is *invariance under scaling*. For

  $$\texttt{foo} : \forall u. \mathbf{num}\, u \to \mathbf{num}\, u^2$$

  this amounts to the property

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

  for any positive "scale factor" k.
- This is an example of a "free theorem". Compare

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

  and the theorem

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

Extensional semantics of units

- Semantics is based on *scaling invariance*
  - Compare polymorphism as *representation independence*
  - Similar technology: parameterized binary logical relations
- See

  Relational Parametricity and Units of Measure
  Andrew Kennedy, POPL 1997

  for original work, based on a System-F-like language.
- Aim now: formalize in Coq, generalize results
  - No terms yet. Instead, purely semantic results over Coq functions
  - For crisper results, we assume an abstract base domain of *positive* values forming a multiplicative Abelian group (e.g. $\mathbb{R}^+$ or $\mathbb{Q}^+$)

Results

Theorems for Free. Another example: if

$$\models d : \forall uv.\mathbf{num}\,u \to (\mathbf{num}\,u \to \mathbf{num}\,v) \to (\mathbf{num}\,u \to \mathbf{num}\,v \cdot u^{-1})$$

then

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

Type isomorphisms. For example

$$\forall u.\mathbf{num}\,u \to \mathbf{num}\,u \;\cong\; \mathbf{num}\,\mathbf{1}$$

To see why, consider what functions have the left-hand type.

This is one an instance of the more general "Pi Theorem".

Syntax: units and types

- Unit expressions have grammar $\mu ::= u \mid \mathbf{1} \mid \mu \cdot \mu \mid \mu^{-1}$
- Axiomatize equational theory $=_U$ on units (Abelian group) :

identity	$\mathbf{1} \cdot \mu =_U \mu$	inverse	$\mu \cdot \mu^{-1} =_U \mathbf{1}$
assoc	$(\mu_1 \cdot \mu_2) \cdot \mu_3 =_U \mu_1 \cdot (\mu_2 \cdot \mu_3)$	comm	$\mu_1 \cdot \mu_2 =_U \mu_2 \cdot \mu_1$

- Type expressions have grammar

$$\tau ::= \textsf{num}\,\mu \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \forall u.\tau \mid \textsf{unit} \mid \textsf{void}$$

- No base units (e.g. kg, m, s)! Just quantify at top-level

Mechanizing Abelian groups

- Package operations and axioms in a record:

```
Record AbGroup := mkGroup {
  carrier :> Set;
  prod : carrier → carrier → carrier;
  inv : carrier → carrier;
  one : carrier;
  assoc : ∀ x y z, prod x (prod y z) = prod (prod x y) z;
  comm : ∀ x y, prod x y = prod y x;
  id_r : ∀ x, prod x one = x;
  inv_r : ∀ x, prod x (inv x) = one }.
```

- Similarly for group homomorphisms:

```
Record Hom (G:AbGroup) (H : AbGroup) := mkHom {
  hom :> carrier G → carrier H;
  preserves : ∀ x y : G, hom(prod x y) = prod (hom x) (hom y) }.
```

Units, in Coq

- To mechanize in Coq, we could define syntax inductively:

$$\begin{aligned}&\texttt{Inductive}\ \mathit{Unt} :=\\ &\mid \mathit{UntVar} : \mathit{nat} \to \mathit{Unt}\ \mid \mathit{UntOne} : \mathit{Unt}\\ &\mid \mathit{UntProd} : \mathit{Unt} \to \mathit{Unt} \to \mathit{Unt}\ \mid \mathit{UntInv} : \mathit{Unt} \to \mathit{Unt}.\end{aligned}$$

- But then we'd need to quotient by $=_U$. So instead:

$$\texttt{Definition}\ \mathit{Unt} := \mathit{nat} \to Z.$$

- Unit equivalence is then just extensional equality on functions. We can define operators and *prove* the Abelian group laws:

$$\texttt{Axiom}\ \mathit{UntExtensional} : \forall\ \mu_1\ \mu_2 : \mathit{Unt}, (\forall\ i, \mu_1\ i = \mu_2\ i) \to \mu_1 = \mu_2.$$

$$\begin{aligned}&\texttt{Definition}\ \mathit{UntProd}\ (\mu_1\ \mu_2 : \mathit{Unt}) := \texttt{fun}\ v \Rightarrow \mu_1(v) + \mu_2(v).\\ &\texttt{Notation}\ \text{"u} * \text{v"} := (\mathit{UntProd}\ u\ v).\end{aligned}$$

$$\texttt{Lemma}\ \mathit{UntProd\_comm} : \forall\ \mu_1\ \mu_2, \mu_1 * \mu_2 = \mu_2 * \mu_1.$$

$$\vdots$$

$$\begin{aligned}&\texttt{Canonical Structure}\ \mathit{UntGroup} :=\\ &\quad \mathit{AbGrp.mkGroup}\ \mathit{Unt}\ \mathit{UntProd}\ \mathit{UntInv}\ \mathit{UntNone}\\ &\qquad\qquad \mathit{UntProd\_assoc}\ \mathit{UntProd\_comm}\ \mathit{UntProd\_id\_r}\ \mathit{UntProd\_inv\_r}.\end{aligned}$$

Substitutions, in Coq

- A substitution is just a homomorphism:

```
Definition Subst := Hom UntGroup UntGroup.
```

- We can define e.g. singleton substitutions, identity, etc. We can also easily define the de Bruijn “shift” operator as a homomorphism:

```
Definition shift (μ:Unt):Unt :=
  fun i ⇒ match i with O ⇒ 0 | S j ⇒ μ j end.

Lemma shift_prod : ∀ μ1 μ2:Unt, shift (μ1 * μ2) = shift μ1 * shift μ2.
Proof.
  intros μ1 μ2. unfold shift. apply UntExtensional. intro j.
  case j; compute; auto.
Qed.

Definition shiftAsSubst : Subst.
exact (mkHom UntGroup UntGroup shift shift_prod).
Defined.
```

Types, in Coq

- We define types inductively
- Bound variable in $\forall$ is encoded using de Brujn

$$\begin{array}{l}
\texttt{Inductive}\ Ty := \\
\quad \mid Num : Unt \to Ty \\
\quad \mid Arrow : Ty \to Ty \to Ty \\
\quad \mid Prod : Ty \to Ty \to Ty \\
\quad \mid Sum : Ty \to Ty \to Ty \\
\quad \mid Unit : Ty \\
\quad \mid Void : Ty \\
\quad \mid All : Ty \to Ty.
\end{array}$$

The base domain

- We assume a numeric domain. We could be concrete, e.g. use Coq’s Q (rationals) or R (reals)
  - But results are crisper if we restrict to positive values
- Instead, we assume enough axioms to get our results: just that we have a non-trivial (multiplicative) Abelian group

```
Parameter Base : Set.

Parameter BaseProd :
   Base → Base → Base.
Parameter BaseOne : Base.
Parameter BaseInv : Base → Base.

Notation ”x ∗ y” := (BaseProd x y).
Notation ”1” := (BaseOne).
Notation ”/ x” := (BaseInv x).
Notation ”x / y” := (BaseProd x (BaseInv y)).
```

```
Axiom BaseProd_id_r : ∀ x, x ∗ 1 = x.
Axiom BaseProd_assoc : ∀ x y z, x ∗ (y ∗ z) = (x ∗ y) ∗ z.
Axiom BaseProd_inverse_r : ∀ x, x ∗ / x = 1.
Axiom BaseProd_comm : ∀ x y, x ∗ y = y ∗ x.
Axiom BaseNonTrivial : ∃ x:Base, x ≠ 1.
```

The underlying semantics

```
Fixpoint usem τ :=
  (match τ with
  | Num μ ⇒ Base
  | Arrow τ1 τ2 ⇒ usem τ1 → usem τ2
  | Prod τ1 τ2 ⇒ usem τ1 × usem τ2
  | Sum τ1 τ2 ⇒ usem τ1 + usem τ2
  | Unit ⇒ unit
  | Void ⇒ False
  | All τ ⇒ usem τ
  end)%type.
```

Units ignored

Shallow embedding in Coq types

Units ignored

Scaling environments

- A scaling environment ψ assigns to each unit variable *u* a scale factor from Base
- Extend ψ to unit expressions homomorphically i.e.

$$\begin{array}{rcl} \psi(\mu_1 \cdot \mu_2) & = & \psi(\mu_1) \times \psi(\mu_2) \\ \psi(\mu^{-1}) & = & 1/\psi(\mu) \\ \psi(\mathbf{1}) & = & 1 \end{array}$$

- In Coq, just

$$\texttt{Definition}\ \mathit{SEnv} := \mathit{Hom}\ \mathit{UntGroup}\ \mathit{BaseGroup}.$$

Parametric logical relation

```
Definition SEnvExtends (ψ':SEnv) (ψ:SEnv) := ∀ μ, ψ' (shift μ) = ψ(μ).
```

ψ' extends ψ with assignment for variable 0

Binary relation over underlying semantics

```
Fixpoint sem (ψ:SEnv) (τ : Ty) : usem τ → usem τ → Prop :=
  match τ with
  | Num μ ⇒ fun x y ⇒ y = ψ(μ) * x
  | Arrow τ1 τ2 ⇒ RelArrow (sem ψ τ1) (sem ψ τ2)
  | Prod τ1 τ2 ⇒ RelProd (sem ψ τ1) (sem ψ τ2)
  | Sum τ1 τ2 ⇒ RelSum (sem ψ τ1) (sem ψ τ2)
  | Unit ⇒ fun _ _ ⇒ True
  | Void ⇒ fun _ _ ⇒ False
  | All τ ⇒ fun x y ⇒ ∀ ψ', SEnvExtends ψ' ψ → sem ψ' τ x y
  end.
```

x “scales” to *y*

Standard relational operators

Quantify over all extensions of ψ

Using the relation

- Think of *sem* $\psi\ \tau\ f\ g$ as "*f* is equivalent to *g* at type τ under scaling ψ"
- For open types, we write
  - $\models$ f $\sim$ g : τ if for any ψ, sem ψ τ f g
    ("f is semantically equivalent to g at type τ")
  - $\models$ f : τ if for any ψ, sem ψ τ f f
    ("f semantically has type τ")

– It's straightforward to show that base operations have the appropiate types semantically i.e.

$$\models BaseInv : \forall u.\mathsf{num}\, u \rightarrow \mathsf{num}\, u^{-1}$$
$$\models BaseProd : \forall u_1.\forall u_2.\mathsf{num}\, u_1 \rightarrow \mathsf{num}\, u_2 \rightarrow \mathsf{num}\, u_1 \cdot u_2$$
$$\models BaseOne : \mathsf{num}\, \mathbf{1}$$

Isomorphisms

- Define type isomorphism semantically:

$$\begin{array}{l}
\texttt{Definition}\ \mathit{iso}\ \tau_1\ \tau_2 := \exists\, i,\ \exists\, j, \\
\quad \models i \sim i : \mathit{Arrow}\ \tau_1\ \tau_2\ \wedge \\
\quad \models j \sim j : \mathit{Arrow}\ \tau_2\ \tau_1\ \wedge \\
\quad \models (\texttt{fun}\ x \Rightarrow j(i(x))) \sim (\texttt{fun}\ x \Rightarrow x) : \mathit{Arrow}\ \tau_1\ \tau_1\ \wedge \\
\quad \models (\texttt{fun}\ y \Rightarrow i(j(y))) \sim (\texttt{fun}\ y \Rightarrow y) : \mathit{Arrow}\ \tau_2\ \tau_2.
\end{array}$$

$$\texttt{Notation}\ \text{''}\ \tau_1 \cong \tau_2\ \text{''} := (\mathit{iso}\ \tau_1\ \tau_2)\ (\texttt{at}\ \mathit{level}\ 70).$$

- Straightforward to prove that ≅ is a congruence, and some non-unit-specific isomorphisms e.g.

$$\begin{array}{rcl}
\tau_1 \times \tau_2 & \cong & \tau_2 \times \tau_1 \\
(\tau_1 \times \tau_2) \rightarrow \tau_3 & \cong & \tau_1 \rightarrow \tau_2 \rightarrow \tau_3
\end{array}$$

Primitive isomorphisms

- Can then prove some primitive unit-specific isomorphisms e.g.

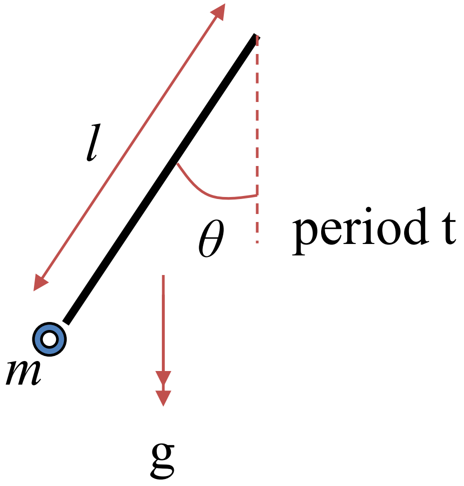
$$
\begin{array}{rcll}
\tau_1 \to \cdots \to \tau_i \to \cdots \tau_j \to \cdots \to \tau_n \to \tau & \cong & \tau_1 \to \cdots \to \tau_j \to \cdots \tau_i \to \cdots \to \tau_n \to \tau & \mathrm{C1} \\
\mathsf{num}\,\mu \to \tau & \cong & \mathsf{num}\,\mu^{-1} \to \tau & \mathrm{C2} \\
\mathsf{num}\,\mu_0 \to \mathsf{num}\,\mu \to \tau & \cong & \mathsf{num}\,\mu_0 \cdot \mu^z \to \mathsf{num}\,\mu \to \tau & \mathrm{C3} \\
 & & & \\
\forall u_1 \cdots \forall u_n \tau & \cong & \forall u_1 \cdots \forall u_n \tau[u_i \mapsto u_j, u_j \mapsto u_i] & \mathrm{R1} \\
\forall u.\tau & \cong & \forall u.\tau[u \mapsto u^{-1}] & \mathrm{R2} \\
\forall u_0.\forall u.\tau & \cong & \forall u_0.\forall u.\tau[u_0 \mapsto u_0 \cdot u^z] & \mathrm{R3} \\
 & & & \\
\forall u.\mathsf{num}\,u^z \to \mathsf{num}\,(u^{y*z} \cdot \mu) & \cong & \mathsf{num}\,\mu \qquad (u \text{ not free in } \mu) & \mathrm{D}
\end{array}
$$

- These can be composed to build isomorphisms such as

$$\forall u.\mathsf{num}\,u \to \mathsf{num}\,u \to \mathsf{num}\,u \cong \mathsf{num}\,\mathbf{1} \to \mathsf{num}\,\mathbf{1}$$

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

- Assume some relation $f(m, l, g, \theta, t) = 0$
- Then by dimensional 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.

Pi Theorem, for first-order types

- Suppose

$$\tau = \forall u_1, \ldots, u_m.\mathsf{num}\,\mu_1 \to \cdots \to \mathsf{num}\,\mu_n \to \mathsf{num}\,\mu_0.$$

Let *A* be *m*×*n* matrix of exponents of variables in μ_1,…,μ_n. Let B be *m*-vector of exponents in μ_0. If AX=B is solvable, then

$$\tau \cong \mathsf{num}\,\mathbf{1} \to \overset{n-r}{\cdots} \to \mathsf{num}\,\mathbf{1} \to \mathsf{num}\,\mathbf{1}$$

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!

Experience of Coq mechanization

- Nice
  - Definition of logical relation just as on paper!
  - If extensionality is assumed, working with functions instead of syntax works very well
  - Canonical Structures used to good effect
  - Setoid feature supports proofs of isomorphisms by *rewriting*
- Nasty
  - An Abelian group tactic would be nice (ring and field are standard)
  - Substitution lemma for logical relations awkward (needs equality coercions)
  - Unfolding of canonical structures by tactic "simpl" is a pain

Problem: Substitution Lemma

- First attempt in Coq:

```
Lemma SEnvSubstSem :
  ∀ τ,   ∀ s ψ,   ∀ x y:usem τ,
  sem (ψ o s) τ x y ↔ sem ψ (subst_ty s τ) x y.
```

- This doesn’t even type-check! Type-checker needs to know

$$usem\,\tau = usem(subst\_ty\;s\,\tau)$$

- Solution: explicit equality coercions.

```
Definition usem_subst : ∀ τ, ∀ (s:Subst), usem τ = usem (subst_ty s τ).
  induction τ....
Defined.

Definition up s τ (x:usem τ) :=
   (eq_rect _ (fun X : Set ⇒ X) x _ (usem_subst τ s)).
Lemma SEnvSubstSem :

  ∀ τ,   ∀ s ψ,   ∀ x y:usem τ,
  sem (ψ o s) τ x y ↔ sem ψ (subst_ty s τ) (up x) (up y).
```

Work in progress

- Formalizing the proof of the Pi Theorem
  - Cf fundamental theorem of finitely generated Abelian groups
- Terms
  - Already shown that semantics is preserved by typing rules
- Formalizing proofs of non-definability
  - Needs a more generous notion of scaling environment (homomorphisms from subgroups of Unt) that model *exactly* the primitive operations in the term language
- Generalization of Pi Theorem to higher-order functions
- Generalization to other domains with similar “invariance under transformation” properties