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

```fsharp
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.
“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.

```plaintext
let good(x: float<kg>, y: float<kg>) = if x<y then print "!
let bad(x: float<kg>, y: float<s>) = if x<y then print "!

in good(1.0<kg>, 2.0<kg>)
in bad(1.0<kg>, 2.0<s>)

let good(x: float<lb>, y: float<lb>) = if x<y then print "!
let bad(x: float<lb>, y: float<s>) = if x<y then print "!

in good(2.2<lb>, 4.4<lb>)
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
  
  \[
  \text{foo} : \text{float}'u \rightarrow \text{float}'u^2
  \]

- If \text{foo} is implemented in F#, then safety follows from soundness of typing rules

- But what if it’s implemented by

\begin{verbatim}
fmul       st(1),st
fmul       st(1),st
fld        DWORD PTR [esp]
fxch       st(1)
fmulp      st(2),st
fsub       st,st(1)
\end{verbatim}

Machine code  FPGA  analogue computer  human computer
Polymorphism, extensionally

• Claim: the essence of unit-polymorphism is \textit{invariance under scaling}. For

\[
\text{foo} : \forall u. \text{num } u \rightarrow \text{num } u^2
\]

this amounts to the property

\[
\forall x, \text{foo}(k \times x) = k^2 \times \text{foo}(x)
\]

for any positive “scale factor” \(k\).

• This is an example of a “free theorem”. Compare

\[
\text{bar} : \forall \alpha. \alpha \rightarrow \alpha \times \alpha
\]

and the theorem

\[
\forall x, \text{bar}(f(x)) = \langle f, f \rangle(\text{bar}(x))
\]
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

\[ \models d : \forall uv. \text{num} u \rightarrow (\text{num} u \rightarrow \text{num} v) \rightarrow (\text{num} u \rightarrow \text{num} v \cdot u^{-1}) \]

then

\[ \forall k_1, k_2, d \ h \ f \ x = \frac{k_2}{k_1} \cdot d \left( \frac{h}{k_1} \right) \left( \lambda x. \frac{f(x \cdot k_1)}{k_2} \right) \left( \frac{x}{k_1} \right) \]

Type isomorphisms. For example

\[ \forall u. \text{num} u \rightarrow \text{num} u \cong \text{num} 1 \]

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
  $\mu ::= u \mid 1 \mid \mu \cdot \mu \mid \mu^{-1}$

- Axiomatize equational theory $=_U$ on units (Abelian group):
  
  identity $\ 1 \cdot \mu =_U \mu$
  inverse $\ \mu \cdot \mu^{-1} =_U 1$
  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
  $\tau ::= \text{num } \mu \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \forall u.\tau \mid \text{unit} \mid \text{void}$

- 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:

\[
\text{Inductive } \text{Unt} := \\
| \text{UntVar} : \text{nat} \to \text{Unt} | \text{UntOne} : \text{Unt} \\
| \text{UntProd} : \text{Unt} \to \text{Unt} \to \text{Unt} | \text{UntInv} : \text{Unt} \to \text{Unt}.
\]

• But then we’d need to quotient by \(=_{\text{U}}\). So instead:

\[
\text{Definition } \text{Unt} := \text{nat} \to \text{Z}.
\]

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

\[
\text{Axiom } \text{UntExtensional} : \forall \mu_1 \mu_2 : \text{Unt}, (\forall i, \mu_1 i = \mu_2 i) \to \mu_1 = \mu_2.
\]

\[
\text{Definition } \text{UntProd} (\mu_1 \mu_2 : \text{Unt}) := \text{fun } v \Rightarrow \mu_1(v) + \mu_2(v).
\]

\[
\text{Notation } \text{”u \ast v” := (UntProd u v).}
\]

\[
\text{Lemma } \text{UntProd\_comm} : \forall \mu_1 \mu_2, \mu_1 \ast \mu_2 = \mu_2 \ast \mu_1.
\]

\[
\text{Canonical Structure } \text{UntGroup} :=
\text{AbGrp.mkGroup Unt UntProd UntInv UntNone}
\text{UntProd\_assoc UntProd\_comm UntProd\_id\_r UntProd\_inv\_r}.
\]
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 Bruijn

\[
\text{Inductive } \text{Ty} := \\
\quad \text{Num} : \text{Un} \to \text{Ty} \\
\quad \text{Arrow} : \text{Ty} \to \text{Ty} \to \text{Ty} \\
\quad \text{Prod} : \text{Ty} \to \text{Ty} \to \text{Ty} \\
\quad \text{Sum} : \text{Ty} \to \text{Ty} \to \text{Ty} \\
\quad \text{Unit} : \text{Ty} \\
\quad \text{Void} : \text{Ty} \\
\quad \text{All} : \text{Ty} \to \text{Ty}.
\]
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.

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.

Notation "x * y" := (BaseProd x y).
Notation "1" := (BaseOne).
Notation "/ x" := (BaseInv x).
Notation "x / y" := (BaseProd x (BaseInv y)).
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 $\psi$ assigns to each unit variable $u$ a scale factor from Base

• Extend $\psi$ to unit expressions homomorphically i.e.

$$
\psi(\mu_1 \cdot \mu_2) = \psi(\mu_1) \times \psi(\mu_2) \\
\psi(\mu^{-1}) = 1/\psi(\mu) \\
\psi(1) = 1
$$

• In Coq, just

Definition $SEnv := Hom \ UntGroup \ BaseGroup$. 
Parametric logical relation

Definition $SEnvExtends (\psi':SEnv) (\psi:SEnv) := \forall \mu, \psi' (\text{shift}\ \mu) = \psi(\mu)$.

Fixpoint $sem (\psi:SEnv) (\tau : Ty) : usem \tau \rightarrow usem \tau \rightarrow \text{Prop} :=$

match $\tau$ with
| $\text{Num}\ \mu \Rightarrow \text{fun} \ x \ y \Rightarrow y = \psi(\mu) * x$  
| $\text{Arrow} \ \tau_1 \ \tau_2 \Rightarrow \text{RelArrow} (sem \ \psi \ \tau_1) (sem \ \psi \ \tau_2)$  
| $\text{Prod} \ \tau_1 \ \tau_2 \Rightarrow \text{RelProd} (sem \ \psi \ \tau_1) (sem \ \psi \ \tau_2)$  
| $\text{Sum} \ \tau_1 \ \tau_2 \Rightarrow \text{RelSum} (sem \ \psi \ \tau_1) (sem \ \psi \ \tau_2)$  
| $\text{Unit} \Rightarrow \text{fun} \ \_ \ \_ \Rightarrow \text{True}$  
| $\text{Void} \Rightarrow \text{fun} \ \_ \ \_ \Rightarrow \text{False}$  
| $\text{All} \ \tau \Rightarrow \text{fun} \ x \ y \Rightarrow \forall \ \psi', \ SEnvExtends \ \psi' \ \psi \rightarrow sem \ \psi' \ \tau \ x \ y$
end.
Using the relation

• Think of $\text{sem} \; \psi \; \tau \; f \; g$ as “$f$ is equivalent to $g$ at type $\tau$ under scaling $\psi$”

• For open types, we write
  • $\models f \sim g : \tau$ if for any $\psi$, $\text{sem} \; \psi \; \tau \; f \; g$
    (“$f$ is semantically equivalent to $g$ at type $\tau$”)
  • $\models f : \tau$ if for any $\psi$, $\text{sem} \; \psi \; \tau \; f \; f$
    (“$f$ semantically has type $\tau$”)

— It’s straightforward to show that base operations have the appropriate types semantically i.e.

\[
\begin{align*}
\models \text{BaseInv} : \forall u. \text{num} \; u \rightarrow \text{num} \; u^{-1} \\
\models \text{BaseProd} : \forall u_1.\forall u_2. \text{num} \; u_1 \rightarrow \text{num} \; u_2 \rightarrow \text{num} \; u_1 \cdot u_2 \\
\models \text{BaseOne} : \text{num} \; 1
\end{align*}
\]
Isomorphisms

- Define type isomorphism semantically:

\[
\text{Definition } iso \, \tau_1 \, \tau_2 := \exists \, i, \exists \, j, \\
\quad \vdash \ i \sim \ i : \text{Arrow } \tau_1 \, \tau_2 \land \\
\quad \vdash \ j \sim \ j : \text{Arrow } \tau_2 \, \tau_1 \land \\
\quad \vdash (\text{fun } x \Rightarrow j(i(x))) \sim (\text{fun } x \Rightarrow x) : \text{Arrow } \tau_1 \, \tau_1 \land \\
\quad \vdash (\text{fun } y \Rightarrow i(j(y))) \sim (\text{fun } y \Rightarrow y) : \text{Arrow } \tau_2 \, \tau_2.
\]

Notation ” \tau_1 \cong \tau_2 ” := (iso \, \tau_1 \, \tau_2) (at level 70).

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

\[
\begin{align*}
\tau_1 \times \tau_2 & \cong \tau_2 \times \tau_1 \\
(\tau_1 \times \tau_2) \to \tau_3 & \cong \tau_1 \to \tau_2 \to \tau_3
\end{align*}
\]
Primitive isomorphisms

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

\[ \tau_1 \to \cdots \to \tau_i \to \cdots \to \tau_j \to \cdots \to \tau_n \to \tau \quad \text{C1} \]

\[ \text{num } \mu \to \tau \quad \text{num } \mu^{-1} \to \tau \quad \text{C2} \]

\[ \text{num } \mu_0 \to \text{num } \mu \to \tau \quad \text{num } \mu_0 \cdot \mu^z \to \text{num } \mu \to \tau \quad \text{C3} \]

\[ \forall u_1 \cdots \forall u_n \tau \Rightarrow \forall u_1 \cdots \forall u_n \tau[u_i \mapsto u_j, u_j \mapsto u_i] \quad \text{R1} \]

\[ \forall u.\tau \Rightarrow \forall u.\tau[u \mapsto u^{-1}] \quad \text{R2} \]

\[ \forall u_0.\forall u.\tau \Rightarrow \forall u_0.\forall u.\tau[u_0 \mapsto u_0 \cdot u^z] \quad \text{R3} \]

\[ \forall u.\text{num } u^z \to \text{num } (u^z \cdot \mu) \Rightarrow \text{num } \mu \quad (u \text{ not free in } \mu) \quad \text{D} \]

• These can be composed to build isomorphisms such as

\[ \forall u.\text{num } u \to \text{num } u \to \text{num } u \cong \text{num } 1 \to \text{num } 1 \]
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 \( \theta \) 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

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

\[ \tau = \forall u_1, \ldots, u_m.\text{num } \mu_1 \rightarrow \cdots \rightarrow \text{num } \mu_n \rightarrow \text{num } \mu_0. \]

Let \( A \) be \( m \times n \) matrix of exponents of variables in \( \mu_1, \ldots, \mu_n \). Let \( B \) be \( m \)-vector of exponents in \( \mu_0 \). If \( AX = B \) is solvable, then

\[ \tau \cong \text{num 1} \rightarrow \cdots \rightarrow \text{num 1} \rightarrow \text{num 1} \]

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:

\[
\text{Lemma } SEnvSubstSem : \\
\forall \tau, \forall s \psi, \forall x y : \text{usem } \tau, \\
\text{sem } (\psi \circ s) \tau x y \leftrightarrow \text{sem } \psi (\text{subst}_t y s \tau) x y.
\]

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

\[
\text{usem } \tau = \text{usem } (\text{subst}_t y s \tau)
\]

• Solution: explicit equality coercions.

\[
\text{Definition usem}_{\text{subst}} : \forall \tau, \forall (s: \text{Subst}), \text{usem } \tau = \text{usem } (\text{subst}_t y s \tau).
\]

\[
\text{induction } \tau.:::
\]

\[
\text{Defined.}
\]

\[
\text{Definition up } s \tau (x: \text{usem } \tau) := \\
\quad (\text{eq} \text{\_rect } \_ (\text{fun } X : \text{Set} \Rightarrow X) x \_ (\text{usem}_{\text{subst}} \tau s)).
\]

\[
\text{Lemma } SEnvSubstSem : \\
\forall \tau, \forall s \psi, \forall x y : \text{usem } \tau, \\
\text{sem } (\psi \circ s) \tau x y \leftrightarrow \text{sem } \psi (\text{subst}_t y s \tau) (\text{up } x) (\text{up } y).
\]
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