DEFINING A LANGUAGE

Robert Harper
I am honored to have had the privilege of working with Robin on the development of Standard ML. It is also a pleasure to thank my collaborators, Karl Crary, Derek Dreyer, Daniel Lee, Mark Lillibridge, David MacQueen, John Mitchell, Eugenio Moggi, Greg Morrisett, Frank Pfenning, Chris Stone, and Mads Tofte. And many colleagues in the field over the years.
Robin sought to consolidate disparate work on ML to formulate a common language to support research on automated reasoning and functional programming.

The result was the language **Standard ML**.

The design and implementation of Standard ML set new standards for the field and led to a wealth of further developments.
BOLD OBJECTIVES

A full-scale language with polymorphism, pattern matching, exceptions, higher-order functions, mutable references, abstract types, modules.

A precise definition that would admit analysis, inform implementation, and ensure portability.

An implementation based on the definition that would support application to mechanized proof.
A PROVOCATIVE QUESTION

What does it mean for a language to exist?

Just what sort of thing is a language?

When is a language well-defined?

What can we prove about a language?

Robin’s thesis: a language is a formal object amenable to rigorous analysis.
Answering such questions is the province of semantics, to which Robin’s work was devoted.

Generally speaking, one wishes to give a mathematical formulation of computational ideas, often using ideas from logic, algebra, and topology.

But such methods had never been tried at scale, and there was reason to doubt they would work.
Robin proposed an operational approach that stressed the symmetries between two aspects of a language:

**Statics**, which defines when programs are properly formed.

**Dynamics**, which defines the execution behavior of a program.

At the time *denotational* methods were more popular, but had more limited scope.
The statics consists of *typing judgements*

\[
\text{context} \vdash \text{expression} \Rightarrow \text{type}
\]

The dynamics consists *evaluation judgements*

\[
\text{environment} \vdash \text{expression} \Rightarrow \text{value}
\]

Both are given by *inductive definitions* in the form of *inference rules* like those used in formal logic.
\[
\frac{\Gamma, x \Rightarrow \tau \vdash x \Rightarrow \tau}{\Gamma \vdash e_1 \Rightarrow \text{int} \quad \Gamma \vdash e_2 \Rightarrow \text{int}} \quad \frac{\Gamma \vdash e_1 + e_2 \Rightarrow \text{int}}{\Gamma \vdash e_1 \Rightarrow \text{real} \rightarrow \text{int} \quad \Gamma \vdash e_2 \Rightarrow \text{real}} \quad \frac{\Gamma \vdash e_1(e_2) \Rightarrow \text{int}}{\Gamma \vdash e_1 \Rightarrow \text{int}}
\]
Type inference, which is of great practical importance, is expressed by non-determinism in the rules.

Expressions have many types (are polymorphic).

Just “guess” the appropriate type for a particular situation:

\[
\begin{align*}
\Gamma, x \Rightarrow \text{int} & \vdash x \Rightarrow \text{int} \\
\Gamma \vdash \lambda x.x \Rightarrow \text{int} \rightarrow \text{int}
\end{align*}
\]
\[
E, x \Rightarrow 17 \vdash x \Rightarrow 17
\]

\[
E \vdash e_1 \Rightarrow 17 \quad E \vdash e_2 \Rightarrow 4
\]

\[
E \vdash e_1 + e_2 \Rightarrow 21
\]

\[
E \vdash e_1 \Rightarrow \lambda x.e \quad E \vdash e_2 \Rightarrow v_2 \quad E, x \Rightarrow v_2 \vdash e \Rightarrow v
\]

\[
E \vdash e_1(e_2) \Rightarrow v
\]
A language is *well-defined* (aka *type safe*) if the statics and the dynamics are *coherent*.

The statics “predicts” the form of value.

The dynamics “realizes” the prediction.

For example, a number should not be given a function type, nor a function a numeric type.
Expressing coherence is trickier than it seems!

What cannot happen, not just what does happen.

Robin’s answer was to introduce answers:

\[
\text{environment} \vdash \text{expression} \Rightarrow \text{answer}
\]

An answer is either a value or wrong (a technical device to express impossibility).
WELL-TYPED PROGRAMS DO NOT GO WRONG

Instrument dynamics with run-time checks:

$$E \vdash e_1 \Rightarrow "abc"$$

$$\frac{E \vdash e_1 + e_2 \Rightarrow \text{wrong}}{}$$

**Safety Theorem**: If $exp \Rightarrow typ$ and $exp \Rightarrow ans$, then $ans$ is not $\text{wrong}$.

Show that $answer$ admits $type$.

Show that $\text{wrong}$ does not admit a type.
Principal Type Theorem  In any given context a well-typed expression has a most general, or principal, type of which all others are substitution instances.

Computed using unification (constraint solving).

Corollary Either context ⊢ exp ⇒ typ or not.

Compute principal type (if it has one).

Check that typ is an instance of it.
Consider the function

\[ \lambda f. \text{map } f \ [1, 2, 3] \]

Constraints:

\[ \alpha = \beta \rightarrow \gamma \]
\[ \beta = \delta_1 \rightarrow \delta_2 \]
\[ \delta_1 \ \text{list} = \text{int list} \]

Solution:

\[ (\text{int} \rightarrow \delta) \rightarrow \delta \]
This methodology works well for functional programs, but can it scale up?

Computational effects, such as mutable storage and exceptions.

Modularity and abstraction mechanisms.

Modules posed the most interesting challenges.

(But effects caused trouble too!)
The most ambitious aspect of Standard ML was the module system (designed by Dave MacQueen).

**Signatures** are the types of modules.

**Structures** are hierarchical modules.

**Functors** are functions over modules.

The crux is the concept of *type sharing*, which controls visibility of types across interfaces.
signature QUEUE = sig
  type α queue
  val empty : α queue
  val insert : α × α queue → α queue
  val remove : α queue → α × α queue
end
STRUCTURES

structure Queue : QUEUE = struct
  type α queue = α list × α list
  val empty = (nil, nil)
  val insert = λ(x,q)....
  val remove = λq....
end
The **abstract** signature QUEUE *instantiates* to the **concrete** signature QUEUE’ given by

signature QUEUE’ = sig
  type \( \alpha \) queue = \( \alpha \) list \( \times \) \( \alpha \) list
  val empty : \( \alpha \) queue
  val insert : \( \alpha \times \alpha \) queue \( \rightarrow \) \( \alpha \) queue
  val remove : \( \alpha \) queue \( \rightarrow \) \( \alpha \times \alpha \) queue
end
Remarkably, the definition method scales to modules:

Statics: \( context \vdash module \Rightarrow interface \)

Dynamics: \( environment \vdash module \Rightarrow structure \)

Type sharing relationships are “guessed” non-deterministically.

Generalizes polymorphic inference described above with \( type \) definitions.
Principal Signature Theorem Every well-formed module has a *most general* interface of which all interfaces are *realizations* obtained by substitution.

*Signature matching* is mediated by realization.

(And enrichment, or “width” subtyping.)

Decidability of signature checking follows directly.
The Definition of Standard ML realizes Robin’s vision:

A language is defined by an inductive definition of its statics and dynamics.

Safety is formulated and proved using wrong answers.

Principality supports inference and checking.

At least seven compatible compilers exist for SML!
Nevertheless, *The Definition* has some shortcomings:

- Interaction between polymorphism and effects is problematic (loss of safety and principality).
- Dynamics “cheats” to manage exceptions.
- Use of *wrong* seems needlessly indirect.
- Fudge for the dynamic effect of enrichment order.
- Spurred lots of further research in how to do better.
The type-theoretic foundations for modularity.

MacQueen: dependent types.

Leroy: manifest types, applicative functors.

H+Lillibridge, H+Stone: translucent sums, singleton kinds

Russo+Dreyer: higher-order polymorphism.

Crucial for code certification and mechanization.
Phase distinction: types are static, values dynamic.

Open-scope abstraction: Queue.queue is abstract in all contexts

Singleton kinds: $\tau$ has kind $S(\emptyset)$ iff $\tau$ is equivalent to $\emptyset$.

Generativity: track effects, object identity / ownership

General $\Sigma$ and $\Pi$ signatures.
REDEFINING A LANGUAGE

Statics is now *elaboration* from an “external language” to a type-theoretic “internal language”.

\[
\text{context} \vdash \text{expression} \Rightarrow \text{term} : \text{type}
\]

Dynamics is defined on internal language using Plotkin’s *structural operational semantics*.

\[
\text{term} \left[ \text{memory} \right] \Rightarrow \text{term'} \left[ \text{memory'} \right]
\]
REDEFINING A LANGUAGE

SML statics SML dynamics

SML statics TIL dynamics TIL
Redefining a Language

Safety may be expressed as *progress* and *preservation*.

*Progress*: every well-formed state is either final or makes a transition.

*Preservation*: every transition from a well-formed state is well-formed.

No need for artificial *wrong* transitions that cannot occur (and avoids problems with exceptions).
The type-theoretic framework is crucial to type-based code certification.

Transform a series of typed internal languages starting with elaboration through to assembly.

Transfer external language typing properties to object code.

Example: TILT/TAL compiler for Standard ML.
CERTIFYING COMPILERS

The statics is the front-end, elaborating SML into a clean type theory.

Compiler transformations are type-preserving.

eg, continuation conversion a la Griffin

Object code is Morrisett’s typed assembly language.

type checking ensures safety
CERTIFYING COMPILERS

SML \xrightarrow{\text{elab}} \text{TIL}_1 \xrightarrow{\text{phase}_1} \text{TIL}_2

\text{TIL}_2 \xrightarrow{\text{phase}_2} \text{TIL}_3

\ldots
MECHANIZED
METATHEORY

Doing meta-theory at scale is not humanly feasible.

Hundreds of twisty little cases, all alike.

(Except the one that isn’t.)

Mechanization is clearly desirable, but it proved difficult to use general provers to check safety of The Definition of Standard ML.

van Inwegen’s early effort to prove safety in HOL
MECHANIZED META-THEORY

The Redefinition of Standard ML is much more amenable to mechanization.

Type theory instead of “static semantic objects.”

Transitional, rather than relational, dynamics.

Twelf makes formalization and verification easy!

LF encoding of internal language

Relational meta-theory + coverage checking.
MECHANIZED META-THEORY

Safety of *The Redefinition of Standard ML* has been fully verified (Crary + D. Lee + H).

Statics and dynamics expressed in LF.

Relational meta-theory verified by Twelf coverage checking.

About 30,000 lines of Twelf developed using “extreme programming”.
MECHANIZATION USING TWELF

LF encoding of statics and dynamics:

\[
\text{app}_s : \text{of} \ (\text{app} \ M \ N) \ B \leftarrow \text{of} \ M \ (\text{arr} \ A \ B) \leftarrow \text{of} \ N \ A.
\]

\[
\text{app}_d : \text{steps} \ (\text{app} \ M \ N) \ (\text{app} \ M' \ N) \leftarrow \text{steps} \ M \ M'.
\]

Relational meta-theory acts on derivations:

\[
\text{pres} : \text{steps} \ M \ M' \rightarrow \text{of} \ M \ A \rightarrow \text{of} \ M' \ A \rightarrow \text{type}.
\]

\[
\text{prog} : \text{of} \ M \ A \rightarrow \text{val-or-step} \ M.
\]
State cases of a proof preservation and progress.

\[-: \text{pres (app\textunderscore d DM) (app\textunderscore s SM SN) (app\textunderscore s SM'} \text{ SN)}\]
\[\leftarrow \text{pres DM SM SM}'.\]

Twelf checks coverage and termination.

\[\forall D : \text{steps M M'} \quad \forall S : \text{of M A} \quad \exists S' : \text{of M'} A \quad \top\]
A HUGE SUCCESS

Robin’s methods inspired much future work in language design, and will continue to do so:

Haskell, O’Caml, F#, Scala

Precise language definition is not only possible, but practical and useful.

Compatibility among compilers.

Safety properties, code certification.