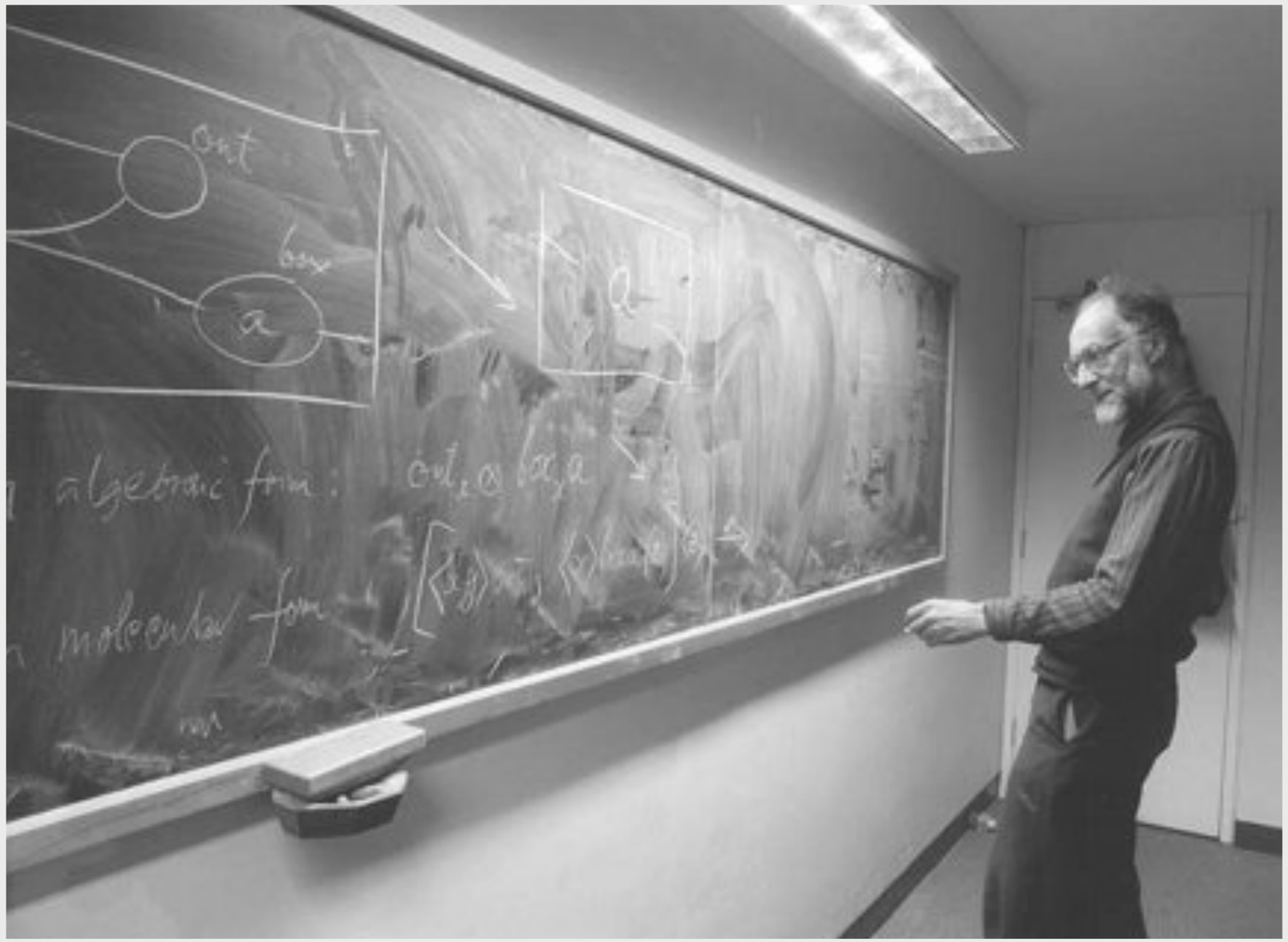
DEFINING A LANGUAGE

Robert Harper



ACKNOWLEDGEMENTS

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.

STANDARD ML

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.

THE ENTERPRISE OF SEMANTICS

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.

AN ELEGANT IDEA

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.

NATURAL SEMANTICS

The statics consists of *typing judgements*

$$context \vdash expression \Rightarrow type$$

The dynamics consists *evaluation judgements*

$$environment \vdash expression \Rightarrow value$$

Both are given by *inductive definitions* in the form of *inference rules* like those used in formal logic.

STATIC SEMANTICS

$$\frac{}{\Gamma, x \Rightarrow \tau \vdash x \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow int \quad \Gamma \vdash e_2 \Rightarrow int}{\Gamma \vdash e_1 + e_2 \Rightarrow int}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow real \to int \quad \Gamma \vdash e_2 \Rightarrow real}{\Gamma \vdash e_1(e_2) \Rightarrow int}$$

STATIC SEMANTICS

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:

$$\frac{\Gamma, x \Rightarrow int \vdash x \Rightarrow int}{\Gamma \vdash \lambda x.x \Rightarrow int \rightarrow int}$$

DYNAMIC SEMANTICS

$$\frac{}{E, x \Rightarrow 17 \vdash x \Rightarrow 17}$$

$$\frac{E \vdash e_1 \Rightarrow 17 \quad E \vdash e_2 \Rightarrow 4}{E \vdash e_1 + e_2 \Rightarrow 21}$$

$$\frac{E \vdash e_1 \Rightarrow \lambda x.e \quad E \vdash e_2 \Rightarrow v_2 \quad E, x \Rightarrow v_2 \vdash e \Rightarrow v}{E \vdash e_1(e_2) \Rightarrow v}$$

TYPE SAFETY

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.

RIGHT AND WRONG

Expressing coherence is trickier than it seems!

What *cannot happen*, not just what *does happen*.

Robin's answer was to introduce *answers*:

$$\mathit{environment} \vdash \mathit{expression} \Rightarrow \mathit{answer}$$

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:

$$\frac{E \vdash e_1 \Rightarrow \text{“}abc\text{”}}{E \vdash e_1 + e_2 \Rightarrow \boldsymbol{wrong}}$$

Safety Theorem: If *exp* ⇒ *typ* and *exp* ⇒ *ans*, then *ans* is not ***wrong***.

Show that *answer* admits *type*.

Show that ***wrong*** does not admit a type.

PRINCIPAL TYPES

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 $context \vdash exp \Rightarrow typ$ or not.

Compute principal type (if it has one).

Check that *typ* is an instance of it.

PRINCIPAL TYPES

Consider the function

$$\lambda f.map\ f\ [1,2,3]$$

Constraints:

$$\alpha = \beta \to \gamma$$

$$\beta = \delta_1 \to \delta_2$$

$$\delta_1\ list = int\ list$$

Solution:

$$(int \to \delta) \to \delta$$

SCALING UP

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!)

MODULES

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.

SIGNATURES

signature QUEUE = sig
  type α queue
  val empty : α queue
  val insert : $\alpha \times \alpha$ queue $\rightarrow \alpha$ queue
  val remove : α queue $\rightarrow \alpha \times \alpha$ 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
```

REALIZATION

The **abstract** signature QUEUE *instantiates* to the **concrete** signature QUEUE′ given by

signature QUEUE′ = sig
  type α queue = **α list × α list**
  val empty : α queue
  val insert : $\alpha \times \alpha$ queue $\rightarrow \alpha$ queue
  val remove : α queue $\rightarrow \alpha \times \alpha$ queue
end

MODULES

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*.

PRINCIPALITY, REVISITED

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.

SUCCESSES AND FAILURES

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!

SUCCESSES AND FAILURES

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.

TYPE-THEORETIC FOUNDATIONS

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.

TYPE-THEORETIC FOUNDATIONS

Phase distinction: types are static, values dynamic.

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

Singleton kinds: τ has kind S(ϱ) iff τ is equivalent to ϱ.

Generativity: track effects, object identity / ownership

General Σ and Π signatures.

REDEFINING A LANGUAGE

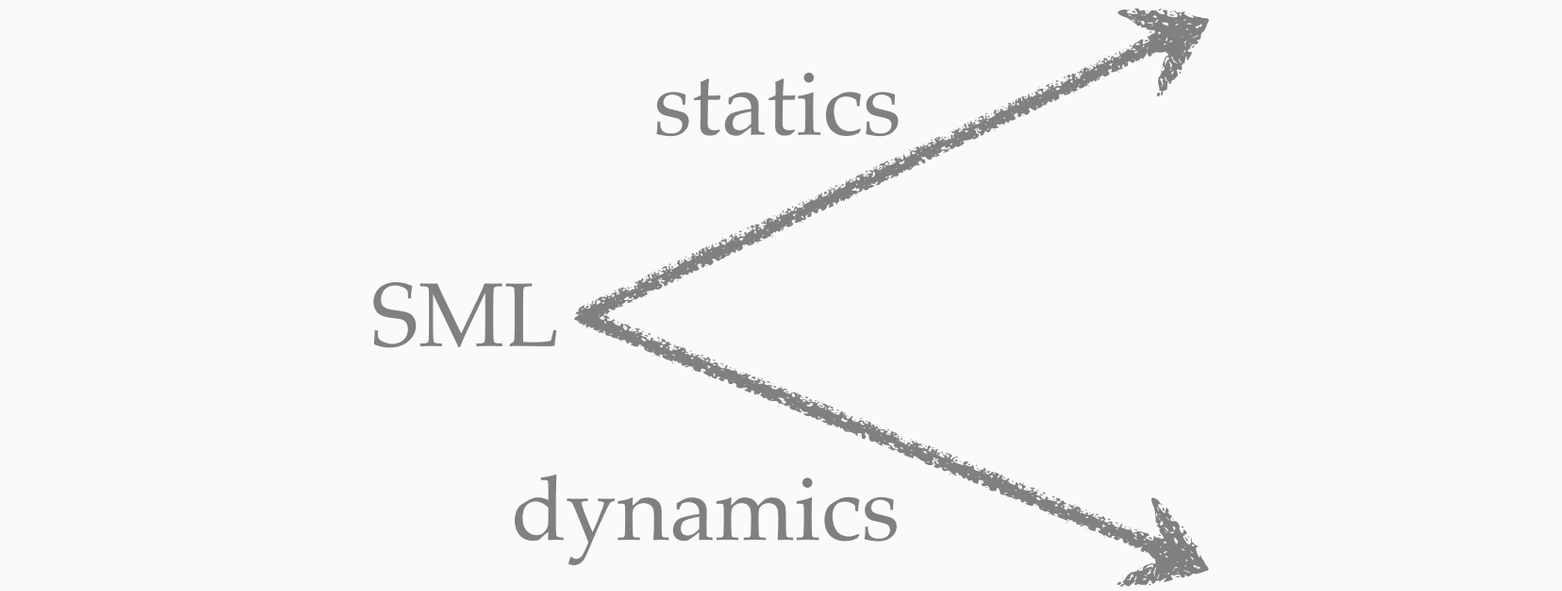
Statics is now *elaboration* from an "external language" to a type-theoretic "internal language".

$$\mathit{context} \vdash \mathit{expression} \Rightarrow \mathit{term} : \mathit{type}$$

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

$$\mathit{term}\ [\mathit{memory}] \mapsto \mathit{term}'\ [\mathit{memory}']$$

REDEFINING A LANGUAGE



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).

CERTIFYING COMPILERS

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.

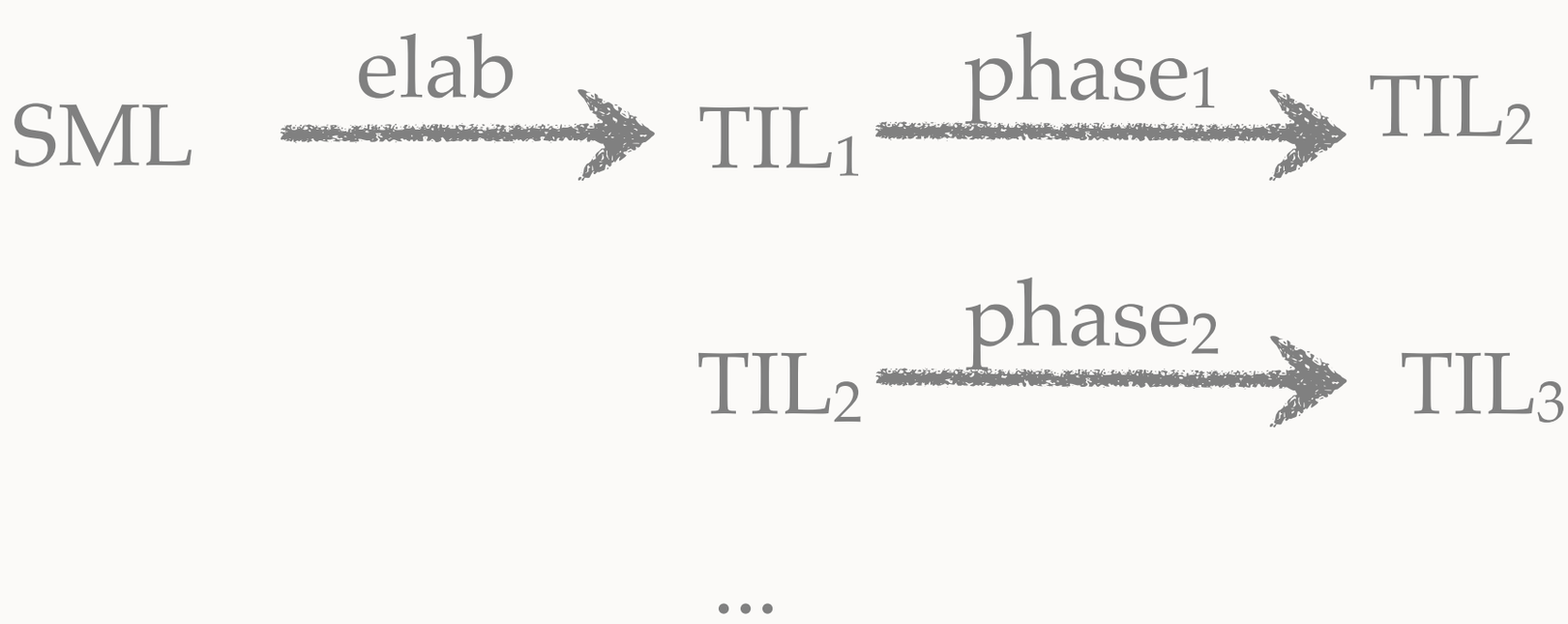
Compiler transformations are type-preserving.

eg, continuation conversion a la Griffin

Object code is Morrisett's *typed assembly language.*

type checking ensures safety

CERTIFYING COMPILERS



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:

app_s : of (app M N) B ← of M (arr A B) ← of N A.

app_d : steps (app M N) (app M′ N) ← steps M M′.

Relational meta-theory acts on derivations:

pres : steps M M′ → of M A → of M′ A → type.

prog : of M A → val-or-step M.

MECHANIZATION USING TWELF

State cases of a proof preservation and progress.

```
- : pres (app_d DM) (app_s SM SN) (app_s SM' SN)
← pres DM SM SM'.
```

Twelf checks coverage and termination.

```
∀ D : steps M M'  ∀ S : of M A  ∃ S' : of M' A  ⊤
```

A HUGE SUCCESS

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

eg, Haskell, O'Caml, F#, Scala

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

Compatibility among compilers.

Safety properties, code certification.