The design of a programming language for provably correct programs: success and failure

Don Sannella

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh
http://homepages.inf.ed.ac.uk/dts
The Standard ML functional programming language

- Origins
- Design and features
- Semantics

The Extended ML framework for specification and development of modular Standard ML software systems

An application of program proof: security certification
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

- For programming proof search strategies
  \[ s : \text{goal} \rightarrow (\text{goal list} \times (\text{thm list} \rightarrow \text{thm})) \]
- Higher order functions for strategy-building combinators
- Exception mechanism for backtracking
- Thm as an abstract data type, with the inference rules as its only constructors
- Polymorphism
- Interactive

Hope (Burstall, 1980)
Algebraic specification (Burstall/Goguen)
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

- For programming proof search strategies
- Higher order functions for strategy-building combinators
  \[ s : \text{goal} \to (\text{goal list} \times (\text{thm list} \to \text{thm})) \]
- Exception mechanism for backtracking
- Thm as an abstract data type, with the inference rules as its only constructors
- Polymorphism
- Interactive

Hope (Burstall, 1980)

Algebraic specification (Burstall/Goguen)
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

- For programming proof search strategies
- Higher order functions for strategy-building combinators
- Exception mechanism for backtracking
  ```
  REPEAT (x ORELSE y) THEN z
  ```
- Thm as an abstract data type, with the inference rules as its only constructors
- Polymorphism
- Interactive

Hope (Burstall, 1980)

Algebraic specification (Burstall/Goguen)
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

- For programming proof search strategies
- Higher order functions for strategy-building combinators
- Exception mechanism for backtracking
- Thm as an abstract data type, with the inference rules as its only constructors
  \[ 
  \text{MP} : \text{thm} \times \text{thm} \rightarrow \text{thm} 
  \]
- Polymorphism
- Interactive

Hope (Burstall, 1980)
Algebraic specification (Burstall/Goguen)
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

- For programming proof search strategies
- Higher order functions for strategy-building combinators
- Exception mechanism for backtracking
- Thm as an abstract data type, with the inference rules as its only constructors
- Polymorphism
  
  \texttt{reverse : \alpha \text{ list} \rightarrow \alpha \text{ list}}

- Interactive

Hope (Burstall, 1980)

Algebraic specification (Burstall/Goguen)
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

Hope (Burstall, 1980)

```
datatype α tree = empty
            | node of α tree * α * α tree

fun flatten empty = []
| flatten(node(t1,x,t2)) =
  (flatten t1) @ (x :: (flatten t2))
```

Algebraic specification (Burstall/Goguen)
Standard ML: Origins

Meta-language of LCF theorem prover (Milner, 1978)

Hope (Burstall, 1980)

Algebraic specification (Burstall/Goguen)

- Parameterised modules
- Interfaces and module bodies are separate
- Pushout-style application
- Stratification between code level and module level
Standard ML: Design and features

Design by committee with strong leadership (1983-1987)

- Mainly Edinburgh, plus Dave MacQueen
- Led by Robin Milner

Core language

Module language
Standard ML: Design and features

Design by committee with strong leadership (1983-1987)

Core language

- ML’s features
- Hope’s algebraic data types
- Cardelli’s labelled records
- Generalised exceptions
- Generalised references
- Call-by-value

Module language
Design by committee with strong leadership (1983-1987)

Core language

Module language (Dave MacQueen)
- Explicit interfaces (“signatures”)
- Software components (“structures”)
- Generic components (“functors”)
- Shared sub-components with explicit sharing declarations
Standard ML: Language definition (1990)

Syntax – 21 pages
- full “bare” syntax in 2.5 pages

Static semantics (type rules) – 30 pages

Dynamic semantics (evaluation rules) – 17 pages

Commentary (1991)
Standard ML: Language definition (1990)

Syntax – 21 pages

Static semantics (type rules) – 30 pages

\[
\frac{C \vdash \text{exp} : \tau' \rightarrow \tau \quad C \vdash \text{exp}' : \tau'}{C \vdash \text{exp \ exp'} : \tau}
\]

Dynamic semantics (evaluation rules) – 17 pages

Commentary (1991)
Standard ML: Language definition (1990)

Syntax – 21 pages

Static semantics (type rules) – 30 pages

Dynamic semantics (evaluation rules) – 17 pages

\[
E \vdash \text{dec} \triangleright E' \quad E + E' \vdash \text{exp} \triangleright v \\
E \vdash \text{let} \ \text{dec} \ \text{in} \ \text{exp} \ \text{end} \triangleright v
\]

Commentary (1991)
Standard ML: Language definition (1990)

Syntax – 21 pages

Static semantics (type rules) – 30 pages

Dynamic semantics (evaluation rules) – 17 pages

Commentary (1991)

- Explanation of the semantics
- Theorems about the language, e.g. deterministic evaluation, type soundness, existence of principal types
Outline

The Standard ML functional programming language

The Extended ML framework for specification and development of modular Standard ML software systems

- Motivation
- Design
- Theory
- Semantics
- Proof
- Tools
- Failure
- Post mortem

An application of program proof: security certification
Extended ML: Motivation (1985)

Pure functional programming allows straightforward proofs of properties because of referential transparency

- Equational reasoning
- Structural induction
- Standard ML is not pure, but almost
Extended ML: Motivation (1985)

Pure functional programming allows straightforward proofs of properties because of referential transparency

Algebraic specification theory (Sannella/Tarlecki et al)
- algebraic models
- axiomatic specifications
- specification structure
- proof of consequences
- stepwise refinement
- information hiding
- parameterisation
- behavioural equivalence
- independence from logical system
Pure functional programming allows straightforward proofs of properties because of referential transparency

Algebraic specification theory (Sannella/Tarlecki et al)

Standard ML language definition provides a basis for establishing soundness
Minimal extension of Standard ML

- Axioms in first-order logic with equality
- Placeholder for expressions and types that haven’t been written yet
Extended ML: Design (1985)

Minimal extension of Standard ML

A “wide spectrum” language

- Covering specifications, programs, and intermediate stages of development
Extended ML: Design (1985)

Minimal extension of Standard ML
A “wide spectrum” language
Simple and intuitive for ML programmers
Leave out references: too hard
Otherwise stick with full Standard ML
Axioms are just boolean expressions containing extra constants

- \( \forall x : t \Rightarrow \text{expr} \)
- \( \exists x : t \Rightarrow \text{expr} \)
- \( \text{expr} \equiv \text{expr}' \)
- \( \text{expr} \text{ terminates} \)
- \( \text{expr} \text{ raises exn} \)
Axioms are just boolean expressions containing extra constants

Hard problem: interactions between features

- polymorphism
- quantification
- equality
- abstraction boundaries
- exceptions and non-termination
Axioms are just boolean expressions containing extra constants
Hard problem: interactions between features
Looked for solution that is natural for ML programmers
Example: quantification over a polymorphic type

- \( \forall (x, xs) \Rightarrow [x]@xs == xs@[x] \)
- ... looks like it should be false
- ... but it is polymorphic – with types we have
  \( \forall (x: \alpha, xs: \alpha \text{ list}) \Rightarrow [x]@xs == xs@[x] \)
- true if \( \alpha \) is \text{unit}, false otherwise!
- ... so it is taken to have no meaning
- \( \forall xs \Rightarrow \exists ys \Rightarrow xs@ys == ys@xs \)
  is true, because \( y=[] \) satisfies it
Axioms are just boolean expressions containing extra constants

Hard problem: interactions between features

Looked for solution that is natural for ML programmers

**Example: quantification over a polymorphic type**

- Easy solution: require explicit quantification over type variables
- But ML has implicit polymorphism!

Lots of very interesting problems to do with modules
Methodology for formal development of modular software systems by stepwise refinement and decomposition
Theory is independent of language used for axioms and language used for coding “in the small”

- Experiments with Prolog
- Experiments with knowledge representation language

Lots of very interesting problems to do with modules
Methodology for formal development of modular software systems by stepwise refinement and decomposition
Theory is independent of language used for axioms and language used for coding “in the small”
Drove development of theory of algebraic specification

- behavioural equivalence
- stable constructions
- parameterisation
- implementation of specifications
- institution-independent language definitions

Lots of very interesting problems to do with modules
Methodology for formal development of modular software systems by stepwise refinement and decomposition
Theory is independent of language used for axioms and language used for coding "in the small"
Drove development of theory of algebraic specification
Very productive synergy with algebraic specification work

Determined to build on top of Standard ML semantics

- Static semantics
- Dynamic semantics
- Verification semantics
- Dependencies more complex than before

Diagram:

- Modules static semantics
- Core static semantics

- Modules dynamic semantics
- Core dynamic semantics

Determined to build on top of Standard ML semantics

- Static semantics
- Dynamic semantics
- Verification semantics
- Dependencies more complex than before

Determined to build on top of Standard ML semantics

- Static semantics
- Dynamic semantics
- Verification semantics
- Dependencies more complex than before
- Result was 140 pages
- Found some errors in the Standard ML semantics

Determined to build on top of Standard ML semantics
More hard problems

Example: domain of quantification for function types
Determined to build on top of Standard ML semantics
More hard problems

Example: domain of quantification for function types

- Set-theoretic functions?
- Computable functions?
- Function space in a model of parametric polymorphism?
- Expressible functions?
- But what does “expressible” mean exactly?

Determined to build on top of Standard ML semantics
More hard problems
Example: domain of quantification for function types
Very complex rules

\[
\text{Comp}(FE, s) = VE
\]

\[
\begin{align*}
\gamma > \gamma_1 &= (C, \tau) \cdot \gamma_2 \\
\sigma^\#(C) + \text{Stat } VE \vdash_{\text{STAT}} atexp^* \Rightarrow \tau', \emptyset, \gamma_3 \\
\sigma^\#(\tau) &= \tau' \\
\text{Dyn}(s, FE + VE) \vdash_{\text{DYN}} atexp^* \Rightarrow v_{\text{DYN}}, (\top, \text{ens}) \\
\exists s'. s, (FE + VE, \gamma_1 \cdot \gamma_3) \vdash (\text{fnx } \Rightarrow \text{exp}^*) \text{atexp}^* \Rightarrow \text{true}, s' \\
\text{s.} (FE, \gamma) \vdash \text{forall } x = \Rightarrow \text{exp}^* \Rightarrow \text{true}, s
\end{align*}
\]

Determined to build on top of Standard ML semantics
More hard problems
Example: domain of quantification for function types
Very complex rules
New version of Standard ML language definition (1997)

➢ ... time to start again?
For first-order monomorphic functions that always terminate and never raise exceptions, everything is as expected

- Otherwise, it’s a nightmare
For first-order monomorphic functions that always terminate and never raise exceptions, everything is as expected.

Multi-valued logic because we used boolean expressions as axioms, and boolean expressions can raise exceptions.

➢ So logical connectives sometimes behave strangely.
For first-order monomorphic functions that always terminate and never raise exceptions, everything is as expected.

Multi-valued logic because we used boolean expressions as axioms, and boolean expressions can raise exceptions.

Reasoning about exceptions is intractable (Pitts/Stark 1993)

➢ So equality isn’t even reflexive (expr == expr is not always true)
For first-order monomorphic functions that always terminate and never raise exceptions, everything is as expected.

Multi-valued logic because we used boolean expressions as axioms, and boolean expressions can raise exceptions.

Reasoning about exceptions is intractable (Pitts/Stark 1993)

Specifying higher-order functions is messy: functional arguments typically need to be specified to always terminate and never raise exceptions.

- Likewise for higher-order functional arguments, provided their functional arguments do so.

For first-order monomorphic functions that always terminate and never raise exceptions, everything is as expected. Multi-valued logic because we used boolean expressions as axioms, and boolean expressions can raise exceptions. Reasoning about exceptions is intractable (Pitts/Stark 1993). Specifying higher-order functions is messy: functional arguments typically need to be specified to always terminate and never raise exceptions.

We gave up.

Parsers and typecheckers
Proof obligation generator (prototype)
Limited proof support by translation into PVS (prototype)
Extended ML: Failure

Very good for teaching formal methods to students who know Standard ML already

Otherwise, not enough user interest

➢ Specifications are too hard to write
➢ Formal development of modular programs from specifications is possible, but a lot of work
➢ Proving correctness of single-threaded functional programs is too much work for too little payoff

Proof is intractable
Extended ML: Post mortem

We were too ambitious
There are features of ML that are hard to handle in isolation

➢ But nobody really knew that at the time
Extended ML: Post mortem

We were too ambitious
There are features of ML that are hard to handle in isolation
... and they are much harder to handle in combination
Doing it formally for a “real” language was very hard

➢ But I still believe in that goal
Extended ML: Post mortem

We were too ambitious
There are features of ML that are hard to handle in isolation
... and they are much harder to handle in combination
Doing it formally for a “real” language was very hard
Doing design and semantics long before proof and tools
was a big mistake

Correctness of pure functional programs is not a problem in
practice
Extended ML: Alternatives

Start with a small subset, do semantics, proofs and tools for that
Add a feature and iterate
Stop when the next iteration is too hard
Attractive starting point: Moggi’s computational lambda calculus (1989)
Forget proofs, focus on specification-based testing
Testing as a useful approximation to proof
Sometimes it is even as good as proof
Axioms as an aid to programming productivity
Extended ML: Alternatives

Be much less ambitious about the kinds of properties to be proved
Focus on properties that people care about
... and situations where having a proof of that property is valuable

Security certification!
Outline

The Standard ML functional programming language

The Extended ML framework for specification and development of modular Standard ML software systems

An application of program proof: security certification

- Proof-carrying code
- Evidence-based certification
In Microsoft I trust

Do you want to install and run "Provides Files to Add Active Debugging to Hosts and Engines" signed on 7/27/2000 10:29 AM and distributed by:

Microsoft Corporation

Publisher authenticity verified by VeriSign Commercial Software Publishers CA

Caution: Microsoft Corporation asserts that this content is safe. You should only install/view this content if you trust Microsoft Corporation to make that assertion.

Always trust content from Microsoft Corporation

Yes  No  More Info
Who should read this bulletin: All customers using Microsoft® products.

Technical description: In mid-March 2001, VeriSign, Inc., advised Microsoft that on January 29 and 30, 2001, it issued two VeriSign Class 3 code-signing digital certificates to an individual who fraudulently claimed to be a Microsoft employee. …

Impact of vulnerability: Attacker could digitally sign code using the name “Microsoft Corporation”.
**Proof-carrying code (Necula, 1997)**

**PCC** certifies code with a condensed formal proof of a desired property.

- Checked by client before installation / execution
- Proofs may be hard to generate, but are easy to check
- Independent of trust networks: unforgeable, tamper-evident

A *certifying compiler* uses types and other high-level source information to create the necessary proof to accompany machine code.
PCC architecture

---

**Code producer**
- Source program
- Certifying compiler
- Safety proof
- Compiled code

---

**Network**

---

**Code consumer**
- Safety policy
- Proof checker
- Compiled code
- Safety proof

---

RUN IT
OK?
Types and annotations can be inferred using a separate linear constraint solver, and proofs can be generated from type derivations.
More generally: Evidence-based Security

PCC certifies code with a condensed formal proof of a desired property.

- Checked by client before installation / execution
- Proofs may be hard to generate, but are easy to check
- Independent of trust networks: unforgeable, tamper-evident

Evidence-based security is about certifying code with checkable evidence of a desired property.

Proof-carrying code is just one example.

Some forms of evidence provide weaker guarantees than proof.
Conclusion

Things are sometimes a lot harder than they appear.

Doing theory and practice hand-in-hand is important for both.

Times change and new applications can build on old work.

This is a fruitful area for research and experimentation.