Advances in Programming Languages

Lecture 6: Dependent Types

Ian Stark

School of Informatics
The University of Edinburgh

Friday 7 October 2016
Semester 1 Week 3

THE UNIVERSITY of EDINBURGH

Topic: Some Types in Programming Languages

The current block of lectures look at some uses of *types*.

- Terms and Types
- Parameterized Types and Polymorphism
- Higher Polymorphism
- Dependent Types

The study of *Type Theory* is part of logic and the foundations of mathematics. However, many aspects of it apply directly to programming languages, and research in type systems has for many decades been an active route for the exchange of new ideas between computer science and mathematics.

Outline

Outline

Academic Integrity !

University of Edinburgh Undergraduate Assessment Regulations

Regulation 29

All work submitted for assessment by students is accepted on the understanding that it is the student's own effort without falsification of any kind.

http://www.inf.ed.ac.uk/teaching/tar

See also:

- University guidance
  http://www.ed.ac.uk/academic-services/students/conduct/academic-misconduct
- Informatics statement
  http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

Working practices

- Start with a blank document; all the words must be yours.
- Do not cut and paste from other documents.
  - Except for direct quotations, which must have source declared.
- Do not let others read your text; nor read theirs (except as directed).

You do not need an online tool to check your own work

Aims of this assignment

- To learn about the chosen topic
- To improve researching and learning skills
- To demonstrate said knowledge and skills

The tangible result is a document, written by you, demonstrating what you have learnt.

"iThenticate allows me to check my own work as well as the work from contributing co-authors and students. When I submit a paper to a journal I am confident that it is not duplicated anywhere. **iThenticate gives me peace-of-mind about the authenticity of my work**."

[name withheld], Professor of Finance at [...] University

iThenticate *Customer Success Stories*

Common Marking Scheme !

Honours Class	Mark (%)	Grade	Summary Description
I	90-100	A1	Excellent (Outstanding)
I	80-89	A2	Excellent (High)
I	70-79	A3	Excellent, MSc Distinction
II.1	60-69	B	Very Good, MSc Merit
II.2	50-59	C	Good, MSc Pass
III	40-49	D	Undergraduate Pass
Fail	30-39	E	Marginal Fail
Fail	20-29	F	Clear Fail
Fail	10-19	G	Bad Fail
Fail	0-9	H	Very Bad Fail

For undergraduate students, a mark of 40 or over is required to pass the course. For postgraduate students, a mark of 40 or over is sufficient for the diploma or certificate, 50 or over is required for a masters degree, and 70 or over is awarded distinction.

Grade Descriptors !

The assignment instructions include *descriptors* of the general standard expected at each of the grade levels. These are chiefly for essay-based assessment, but remain relevant for other coursework and exams. They also capture some distinctive features of the assessment model.

- Assessment is *criterion-referenced* rather than *norm-referenced*.
- Marks and grades are a measure of achievement, not a reward for effort or participation.
- Assessment is of the work, not the person. (Anonymous where possible)
- Mark schemes are not only about allocation of marks to different sections or aspects, but also the quality of performance in that aspect.

Note that the highest grades are reserved for work that is not just correct but also shows notable depth, originality, or other exceptional performance.

Outline

Review

Subtype polymorphism in Java, and type parameters that are covariant, contravariant or invariant.

Limitations of Hindley-Milner where type schemes can describe a whole collection of types, but functions cannot require arguments that are themselves polymorphic.

System F with type parameterization for polymorphic terms, so that $\forall X.A$ is a type, $\Lambda X.M : \forall X.A$ and if $F : \forall X.B$ then $F\,A : B\{A/X\}$.

Encoding datatypes in System F, like Prod $X\,Y = \forall Z.((X \to Y \to Z) \to Z)$.

Beyond System F with bounded and F-bounded quantification, $F_{<:}$, F_2 and F_ω.

Dependencies So Far

So far in constructing terms and types of the lambda-calculus and its various extension we have seen the following different interactions between types and terms.

First-class functions	$\lambda x{:}A.M : A \to B$	Terms that depend on terms
Parameterized types	Set<String>, Tree : * −> *	Types that depend on types
Polymorphic terms	reverse : $\forall A(\text{list } A \to \text{list } A)$	Terms that depend on types

Dependencies So Far

So far in constructing terms and types of the lambda-calculus and its various extension we have seen the following different interactions between types and terms.

First-class functions	$\lambda x{:}A.M : A \to B$	Terms that depend on terms
Parameterized types	Set<String>, Tree : $* \text{ —> } *$	Types that depend on types
Polymorphic terms	reverse : $\forall A(\text{list } A \to \text{list } A)$	Terms that depend on types

From these we can conjecture a natural fourth kind of dependency, which is the topic of this lecture.

Dependencies So Far

So far in constructing terms and types of the lambda-calculus and its various extension we have seen the following different interactions between types and terms.

First-class functions	$\lambda x{:}A.M : A \to B$	Terms that depend on terms
Parameterized types	Set<String>, Tree : $* \mathrel{-}> *$	Types that depend on types
Polymorphic terms	reverse : $\forall A(\text{list } A \to \text{list } A)$	Terms that depend on types

From these we can conjecture a natural fourth kind of dependency, which is the topic of this lecture.

Dependent types	$[[1,2,3],[4,5,6]] : \text{Matrix } 2\ 3$	Types that depend on terms.

Example: Matrices and Vectors

Dependent types allow us to give precise types to terms such as these:

$$\begin{aligned}
[[1,2,3],[4,5,6]] &: \mathsf{Matrix}\,2\,3 &\qquad \mathsf{Matrix} : \mathsf{Num} \to \mathsf{Num} \to * \\
\mathsf{identity\text{-}matrix}\ n &: \mathsf{Matrix}\,n\,n \\
\mathsf{matrix\text{-}invert} &: \forall n{:}\mathsf{Num}\,.\,(\mathsf{Matrix}\,n\,n \to \mathsf{Matrix}\,n\,n) \\
\mathsf{matrix\text{-}multiply} &: \forall n,m,p : \mathsf{Num}\,.\,(\mathsf{Matrix}\,n\,m \to \mathsf{Matrix}\,m\,p \to \mathsf{Matrix}\,n\,p)
\end{aligned}$$

Type-checking then ensures that the operations of matrix inversion and multiplication are applied only to matrices with an appropriate number of rows and columns.

Example: Matrices and Vectors

Dependent types allow us to give precise types to terms such as these:

$$
\begin{aligned}
[[1,2,3],[4,5,6]] &: \mathsf{Matrix}\ 2\ 3 \qquad\qquad \mathsf{Matrix} : \mathsf{Num} \to \mathsf{Num} \to * \\
\text{identity-matrix}\ n &: \mathsf{Matrix}\ n\ n \\
\text{matrix-invert} &: \forall n{:}\mathsf{Num}\,.\,(\mathsf{Matrix}\ n\ n \to \mathsf{Matrix}\ n\ n) \\
\text{matrix-multiply} &: \forall n,m,p : \mathsf{Num}\,.\,(\mathsf{Matrix}\ n\ m \to \mathsf{Matrix}\ m\ p \to \mathsf{Matrix}\ n\ p)
\end{aligned}
$$

Type-checking then ensures that the operations of matrix inversion and multiplication are applied only to matrices with an appropriate number of rows and columns.

These next examples, this time for fixed-length vectors of values, combine dependent types with polymorphism and also arithmetic within the types.

$$
\begin{aligned}
['c','a','t'] &: \mathsf{Vec}\ \mathsf{Char}\ 3 \qquad\qquad \mathsf{Vec} : * \to \mathsf{Num} \to * \\
\text{vector-map} &: \forall X, Y\,.\,(X \to Y) \to \forall n{:}\mathsf{Num}\,.\,(\mathsf{Vec}\ X\ n \to \mathsf{Vec}\ Y\ n) \\
\text{vector-append} &: \forall A\,.\,\forall n, m : \mathsf{Num}\,.\,\mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ (n+m)
\end{aligned}
$$

Example: Safe Indexing

Another kind of dependent type is exemplified by the constructor $\mathsf{EQ} : \mathsf{Num} \to \mathsf{Num} \to *$ where $\mathsf{EQ}\ n\ m$ contains exactly one element if $n = m$ and is empty if $n \neq m$. This is known as the *identity type* or *equality type*.

Similar type constructors $\mathsf{LT}\ n\ m$ and $\mathsf{LE}\ n\ m$ are equivalent to the one-element *unit type* 1 or empty *zero type* 0 according to whether or not $n < m$ or $n \leqslant m$.

With these we can write a desired type for safe indexing into a fixed-length vector.

$$\text{vector-get} \ : \ \forall X \,.\, \forall i{:}\mathsf{Num} \,.\, \forall n{:}\mathsf{Num} \,.\, (\mathsf{LE}\ 0\ i) \to (\mathsf{LT}\ i\ n) \to (\mathsf{Vec}\ X\ n) \to X$$

In principle types like this allow compile-time checking of array bounds, and languages like *Dependent ML* have been implemented using this approach.

The challenge, though, is to make this manageable for the programmer by inferring types, values and proofs wherever possible, and automatically inserting them as *implicit arguments*.

Rules for Dependent Function Types

Types

Dependent Function Type

$$\frac{\Gamma, x : A \vdash \text{Type}\, B}{\Gamma \vdash \text{Type}\, \forall x{:}A.B}$$

Note that the dependent type B may mention x or any other variable from the context Γ.

Terms

Variable

$$\frac{}{\Gamma \vdash x : A} \quad x : A \in \Gamma$$

Abstraction

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.M) : \forall x{:}A.B}$$

Application

$$\frac{\Gamma \vdash F : \forall x{:}A.B \qquad \Gamma \vdash M : A}{\Gamma \vdash F\, M : B\{M/x\}}$$

Where type B does not in fact depend on parameter x, then we can write the type $(\forall x{:}A.B)$ as $(A \to B)$ and it behaves just like the simple function space.

The beta-reduction rule for dependent functions is exactly as for simple functions.

Beta-Reduction $$(\lambda x{:}A.M)\, N \;\longrightarrow\; M[N/x]$$

Dependent Pairs

In a *dependent pair* the type of the second component may depend on the value of the first. For example, consider a function that transforms a list of arbitrary length into a fixed-length vector.

$$\text{vector-create} \;:\; \mathsf{List}\, A \to (\exists n{:}\mathsf{Num}\,.\, \mathsf{Vec}\, A\, n)$$

Dependent pair type $(\exists n{:}\mathsf{Num}\,.\, \mathsf{Vec}\, A\, n)$ packages up a number n with a vector of length n.

$$(\,3,\ [\text{'}c\text{'}, \text{'}a\text{'}, \text{'}t\text{'}]\,) \;:\; (\exists n{:}\mathsf{Num}\,.\, \mathsf{Vec}\ \mathsf{Char}\, n)$$

These dependent pairs can also be useful as input types. For example:

$$\text{vector-filter} \;:\; \forall X\,.\,(X \to \mathsf{Bool}) \to (\exists n{:}\mathsf{Num}\,.\, \mathsf{Vec}\, X\, n) \to (\exists m{:}\mathsf{Num}\,.\, \mathsf{Vec}\, X\, m)$$

Rules for Dependent Pair Types

Types

Dependent Pair Type

$$\frac{\Gamma, x : A \vdash \mathsf{Type}\ B}{\Gamma \vdash \mathsf{Type}\ \exists x{:}A.B}$$

Note that the dependent type B may mention x or any other variable from the context Γ.

Terms

Pairing

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash (M, N) : \exists x{:}A.B}$$

Left Projection

$$\frac{\Gamma \vdash P : \exists x{:}A.B}{\Gamma \vdash \mathsf{fst}(P) : A}$$

Right Projection

$$\frac{\Gamma \vdash P : \exists x{:}A.B}{\Gamma \vdash \mathsf{snd}(P) : B\{\mathsf{fst}(P)/x\}}$$

Where type B does not in fact depend on parameter x, then we can write the type $(\exists x{:}A.B)$ as $(A \times B)$ and it behaves just like the simple pairing.

The reduction rules for dependent pairs are again just as in the simply-typed version.

$$\mathsf{fst}(M, N) \longrightarrow M \qquad\qquad \mathsf{snd}(M, N) \longrightarrow N$$

Many Variations on a Theme

Dependent types appear in many, many different type systems, often in combination with several of the features from System F and its extensions.

All are captured under the broad heading of "Type Theory", and have applications across computer science, logic, and even into linguistic theories of natural language.

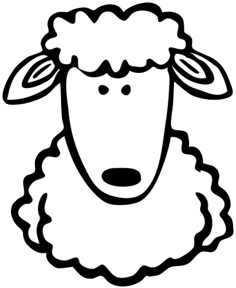
Reflecting this variety, the basic type constructions appear in many syntactic forms. For example:

Function Type	Pair Type
$\forall x{:}A.B$	$\exists x{:}A.B$
$\Pi x{:}A.B$	$\Sigma x{:}A.B$
$(x{:}A) \to B$	$(x{:}A) \times B$
$\{x{:}A\}B$	$\langle x{:}A\rangle B$

Outline

A Small Language

Lam



Consider the following very small typed functional language:

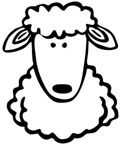
Types	$\tau ::= \iota \mid o \mid \tau \to \tau$
Terms	$M ::= c^\tau \mid x^\tau \mid \lambda x^\tau.M \mid M\,M$

Here ι is a type of integers, o a type of booleans, c^τ represents a constant of type τ and x^τ a variable of type τ.

Suppose that we wish to work with *Lam* types and terms as an *object language* within some host programming language, the *meta language*.

We shall do this using a *deep embedding*, where we directly manipulate object language syntax.

Lam: Simple Types



With a simply-typed programming language, we can define datatypes "LamType", "LamVar" and "LamTerm" for *Lam* types, variables and terms, with the following useful operations:

$$\begin{aligned} \mathsf{i}, \mathsf{o} &: \mathsf{LamType} \\ \mathsf{fun} &: \mathsf{LamType} \to \mathsf{LamType} \to \mathsf{LamType} \\ \\ \mathsf{mkvar} &: \mathsf{String} \to \mathsf{LamVar} \end{aligned}$$

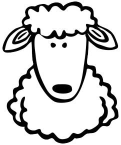
$$\begin{aligned} \mathsf{lamv} &: \mathsf{LamVar} \to \mathsf{LamTerm} \\ \mathsf{lami} &: \mathsf{Num} \to \mathsf{LamTerm} \\ \mathsf{lamo} &: \mathsf{Bool} \to \mathsf{LamTerm} \\ \mathsf{lamlam} &: \mathsf{LamVar} \to \mathsf{LamTerm} \to \mathsf{LamTerm} \\ \mathsf{lamapp} &: \mathsf{LamTerm} \to \mathsf{LamTerm} \to \mathsf{LamTerm} \end{aligned}$$

With these we also need a function to check that the *Lam* terms we build are well-typed.

$$\mathsf{lamcheck} : \mathsf{LamTerm} \to \mathsf{Bool}$$

The "lamcheck" function recursively traverses the term to make sure all values are used with their correct types.

Lam: Dependent Types



With a dependently-typed host language, we can define datatypes that correctly record the *Lam* type structure:

$$\mathrm{i}, \mathrm{o} : \mathrm{LamType}$$

$$\mathrm{fun} : \mathrm{LamType} \to \mathrm{LamType} \to \mathrm{LamType}$$

$$\mathrm{mkvar} : \forall \mathrm{t}{:}\mathrm{LamType}.(\mathrm{String} \to \mathrm{LamVar}(\mathrm{t}))$$

$$\mathrm{lamv} : \forall \mathrm{t}{:}\mathrm{LamType}.\mathrm{LamVar}(\mathrm{t}) \to \mathrm{LamTerm}(\mathrm{t})$$

$$\mathrm{lami} : \mathrm{Num} \to \mathrm{LamTerm}(\mathrm{i})$$

$$\mathrm{lamo} : \mathrm{Bool} \to \mathrm{LamTerm}(\mathrm{o})$$

$$\mathrm{lamlam} : \forall (s, \mathrm{t} : \mathrm{LamType}) \,.\, (\mathrm{LamVar}(s) \to \mathrm{LamTerm}(\mathrm{t}) \to \mathrm{LamTerm}(\mathrm{fun}\, s\, \mathrm{t}))$$

$$\mathrm{lamapp} : \forall (s, \mathrm{t} : \mathrm{LamType}) \,.\, (\mathrm{LamTerm}(\mathrm{fun}\, s\, \mathrm{t}) \to \mathrm{LamTerm}(s) \to \mathrm{LamTerm}(\mathrm{t}))$$

We now have that whenever $M : \mathrm{LamTerm}(\mathrm{t})$, the value M is a correctly-typed *Lam* term, of the type given by $\mathrm{t} : \mathrm{LamType}$.

Programs in the host language can be statically type-checked so that all code handling the object language respects *Lam* type-correctness.

A Small Logic

We can carry out a similar embedding of logic within a host language, say with a type "Prop" of propositions and the following term constructors:

$$
\begin{aligned}
\mathsf{true} &: \mathsf{Prop} & \mathsf{and} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{false} &: \mathsf{Prop} & \mathsf{or} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{not} &: \mathsf{Prop} \to \mathsf{Prop} & \mathsf{imp} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}
\end{aligned}
$$

A Small Logic

We can carry out a similar embedding of logic within a host language, say with a type "Prop" of propositions and the following term constructors:

$$\begin{aligned} \mathsf{true} &: \mathsf{Prop} & \mathsf{and} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\ \mathsf{false} &: \mathsf{Prop} & \mathsf{or} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\ \mathsf{not} &: \mathsf{Prop} \to \mathsf{Prop} & \mathsf{imp} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \end{aligned}$$

With a dependent type constructor "$\mathsf{ProofOf} : \mathsf{Prop} \to *$" we can build logical proofs:

$$\begin{aligned} \mathsf{conj} &: \forall (P, Q : \mathsf{Prop}) \,.\, (\mathsf{ProofOf}(P) \to \mathsf{ProofOf}(Q) \to \mathsf{ProofOf}(\mathsf{and}\ P\ Q)) \\ \mathsf{proj1} &: \forall (P, Q : \mathsf{Prop}) \,.\, (\mathsf{ProofOf}(\mathsf{and}\ P\ Q) \to \mathsf{ProofOf}(P)) \\ \mathsf{triv} &: \mathsf{ProofOf}(\mathsf{true}) \quad \ldots \textit{and other terms capturing rules of logical deduction} \end{aligned}$$

A Small Logic

We can carry out a similar embedding of logic within a host language, say with a type "Prop" of propositions and the following term constructors:

$$
\begin{aligned}
\mathsf{true} &: \mathsf{Prop} & \mathsf{and} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{false} &: \mathsf{Prop} & \mathsf{or} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{not} &: \mathsf{Prop} \to \mathsf{Prop} & \mathsf{imp} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}
\end{aligned}
$$

With a dependent type constructor "$\mathsf{ProofOf} : \mathsf{Prop} \to *$" we can build logical proofs:

$$
\begin{aligned}
\mathsf{conj} &: \forall (P, Q : \mathsf{Prop})\,.\,(\mathsf{ProofOf}(P) \to \mathsf{ProofOf}(Q) \to \mathsf{ProofOf}(\mathsf{and}\ P\ Q)) \\
\mathsf{proj1} &: \forall (P, Q : \mathsf{Prop})\,.\,(\mathsf{ProofOf}(\mathsf{and}\ P\ Q) \to \mathsf{ProofOf}(P)) \\
\mathsf{triv} &: \mathsf{ProofOf}(\mathsf{true}) \qquad \ldots \textit{and other terms capturing rules of logical deduction}
\end{aligned}
$$

If $P : \mathsf{Prop}$ then any value $M : \mathsf{ProofOf}(P)$ is a valid proof of the proposition P. The host language typechecker validates all steps of the logical proof.

A Small Logic

We can carry out a similar embedding of logic within a host language, say with a type "Prop" of propositions and the following term constructors:

$$
\begin{aligned}
\mathsf{true} &: \mathsf{Prop} & \mathsf{and} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{false} &: \mathsf{Prop} & \mathsf{or} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\mathsf{not} &: \mathsf{Prop} \to \mathsf{Prop} & \mathsf{imp} &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}
\end{aligned}
$$

With a dependent type constructor "$\mathsf{ProofOf} : \mathsf{Prop} \to *$" we can build logical proofs:

$$
\begin{aligned}
\mathsf{conj} &: \forall (P, Q : \mathsf{Prop}) \,.\, (\mathsf{ProofOf}(P) \to \mathsf{ProofOf}(Q) \to \mathsf{ProofOf}(\mathsf{and}\ P\ Q)) \\
\mathsf{proj1} &: \forall (P, Q : \mathsf{Prop}) \,.\, (\mathsf{ProofOf}(\mathsf{and}\ P\ Q) \to \mathsf{ProofOf}(P)) \\
\mathsf{triv} &: \mathsf{ProofOf}(\mathsf{true}) \qquad \ldots \textit{and other terms capturing rules of logical deduction}
\end{aligned}
$$

If $P : \mathsf{Prop}$ then any value $M : \mathsf{ProofOf}(P)$ is a valid proof of the proposition P. The host language typechecker validates all steps of the logical proof.

We've Been Here Before

Orwell: The purpose of Newspeak was not only to provide a medium of expression for the world-view and mental habits proper to the devotees of Ingsoc, but to make all other modes of thought impossible. [*1984*]

Outline

An Observation on the Embedded Small Logic

Here are some functions in the host language where we have embedded our small logic.

$$
\begin{aligned}
\text{conj} &: \forall(P, Q : \text{Prop})\,.\,(\text{ProofOf}(P) \to \text{ProofOf}(Q) \to \text{ProofOf}(\text{and } P\ Q)) \\
\text{proj1} &: \forall(P, Q : \text{Prop})\,.\,(\text{ProofOf}(\text{and } P\ Q) \to \text{ProofOf}(P)) \\
\text{proj2} &: \forall(P, Q : \text{Prop})\,.\,(\text{ProofOf}(\text{and } P\ Q) \to \text{ProofOf}(Q))
\end{aligned}
$$

Composing these in appropriate combinations reveals an equivalence of types:

$$\text{ProofOf}(\text{and } P\ Q) \;\longleftrightarrow\; \text{ProofOf}(P) \times \text{ProofOf}(Q)$$

Informally, having a proof of $(P \wedge Q)$ is equivalent to having a proof of P and also a proof of Q.

This only works when we give an *intuitionistic*, or *constructive*, meaning to logic. Other interpretations are available.

Correspondence

This equivalence of types works for other logical connectives, too:

$$\begin{aligned}
\mathsf{ProofOf}(\mathsf{and}\ \mathrm{P}\ \mathrm{Q}) &\longleftrightarrow \mathsf{ProofOf}(\mathrm{P}) \times \mathsf{ProofOf}(\mathrm{Q}) \\
\mathsf{ProofOf}(\mathsf{or}\ \mathrm{P}\ \mathrm{Q}) &\longleftrightarrow \mathsf{ProofOf}(\mathrm{P}) + \mathsf{ProofOf}(\mathrm{Q}) \\
\mathsf{ProofOf}(\mathsf{imp}\ \mathrm{P}\ \mathrm{Q}) &\longleftrightarrow \mathsf{ProofOf}(\mathrm{P}) \to \mathsf{ProofOf}(\mathrm{Q})
\end{aligned}$$

This connection is a manifestation of the *Curry-Howard correspondence* or *propositions-as-types*.

Curry-Howard exposes a close link between the logic and the structure of proofs on one hand, and computer programs and their execution on the other. The correspondence works at many levels and has been an extremely rich source of new ideas in both programming languages and mathematical logic.

Examples of Curry-Howard

Propositions vs. Types

Propositional Logic	True	1	Simple Types
	False	0	
Logical connectives	$P \wedge Q$	$A \times B$	Type constructors
	$P \vee Q$	$A + B$	
	$P \Rightarrow Q$	$A \to B$	
Predicate Logic	$P(x)$	$A(x)$	Dependent Types
Quantification	$\forall x {\in} A.Q(x)$	$\forall x{:}A.B(x)$	
	$\exists x {\in} A.Q(x)$	$\exists x{:}A.B(x)$	

Types $\longleftrightarrow$ Propositions

Terms, Programs $\longleftrightarrow$ Proofs

Term reduction, program execution $\longleftrightarrow$ Proof rewriting, transformation

Outline

Programming Languages for Mathematical Proof

With the small logic earlier we had an embedding of propositions and proofs using types "Prop" and "ProofOf(P)". This approach can be extended to cover a wide range of proofs in logic and mathematics. For example, if A is a type and term $P : A \to \text{Prop}$ is a predicate on values of type A, then consider the dependent type:

$$\exists(x : A)\,.\,(\text{ProofOf}(P(x)))$$

A value of this type is a pair of an element of A with a proof that it satisfies property P. This captures the *set comprehension* construction:

$$\{\, x \in A \mid P(x) \,\} \quad \subseteq A$$

It is also an example of a *refinement type* in a programming language.

The same idea can be further used to model other constructions in set theory.

Programming Languages for Mathematical Proof

Building logic and proof using dependent types within a host programming language is an effective way to construct large mathematical proofs: the machine assists both in creating the proofs, and automatically ensuring their correctness through typechecking.

Turing on why writing programs will always be interesting

> . . . There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.

This approach has been promoted by several theorem-proving systems: the *Edinburgh Logical Framework*, *Twelf*, *Alf*, and most extensively *Coq*.

Coq was used in Gonthier's machine-checked proof of the four-colour theorem; and by Leroy to create the verified *CompCert* C compiler.

However, the *Flyspeck* proof of Kepler's conjecture and the seL4 verified kernel both use a slightly different approach based on Higher-Order Logic

Writing Verified Programs

With a programming language and a logic both embedded as object-languages within a dependently-typed metalanguage, it is possible to specify and verify programs entirely through static type-checking.

However, it's also possible to apply this to the host language itself: to build a dependently-typed language that is suitable for proving theorems, writing programs, and in particular proving that those programs are correct.

This the approach of the *Agda*, *Epigram* and *Idris* programming languages.

Some other functional languages use dependent types to increase the expressiveness and precision of the type system: not necessarily proving mathematical theorems or functional correctness (although they may do that too), but developing the idea of static typing as a way to improve program quality. Examples include *Dependent ML*, *Cayenne* and $F^\star$.

Introduction

F* (pronounced F star) is an ML-like functional programming language aimed at program verification. Its type system includes polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus. Together, these features allow expressing precise and compact specifications for programs, including functional correctness and security properties. The F* type-checker aims to prove that programs meet their specifications using a combination of SMT solving and manual proofs. Programs written in F* can be translated to OCaml or F# for execution.

F* - guide

Microsoft Research

Translate this page

Spanish

Microsoft® Translator

other tutorials close

Certifying program correctness with F*

F* is a verification-oriented programming language developed at Microsoft Research. It follows in the tradition of the ML family of languages in that it is a typed, strict, higher-order programming language. However, its type system is significantly richer than ML's, allowing functional correctness specifications to be stated and checked semi-automatically.

This tutorial provides a first taste of verified programming in F*. We will focus initially on writing several small, purely functional programs and write specifications for these programs that can be automatically verified by F*. Next, we will discuss verifying higher-order programs that also make use of state. Finally, we consider designing and implementing a small cryptographic

```
module Hello
(*
   Verification in F* is based on "refinement types".
   An example of a refinement type is (x:int{x=0}).
   This is the type of constant 0, a subset of the typ
*)
type zero = x:int{x=0}

(* This is an error, of course. *)
let fail = assert (0=1)
```

'▶' shortcut: Alt+B

Outline

Summary

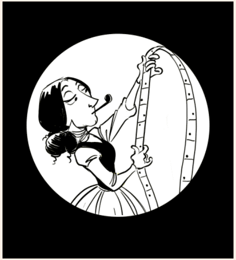
- Dependent function types $\forall x{:}A.B$ and pair types $\exists x{:}A.B$ complete the possible dependencies of {types, terms} on {types, terms}.
- Numerical examples of dependent types include fixed-length vectors and $m \times n$ matrices. Type-checking then ensures compatible lengths or dimensions.
- With deep embedding of an object language in a dependently-typed host language, types can be used to automatically check object-language properties.
- A key example is machine-assisted theorem proving: constructing and checking proofs of logical statements within a host programming language.
- This is linked to the *Curry-Howard correspondence* between propositions and types, proofs and terms.
- Arising from this are a spectrum of languages from proof-assistants like *Coq* to dependently-typed programming languages like $F^\star$.

Ada Lovelace Day 2016

University Main Library

Tuesday 11 October 2016

http://is.gd/lovelace2016



Sydney Padua: 2D Goggles

10–1	Talks, activities, technology, building things
1–5	Women in STEM Wikipedia edit-a-thon ⟵ **book online now**

Ada Lovelace Day is an international celebration day of the achievements of women in science, technology, engineering and maths (STEM). It aims to increase the profile of women in STEM and, in doing so, create new role models who will encourage more girls into STEM careers and support women already working in STEM.

http://findingada.com