Advances in Programming Languages

Lecture 7: Dependent Types

Ian Stark

School of Informatics
The University of Edinburgh

Monday 8 October 2018
Semester 1 Week 4

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
- Higher Types
- 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

Homework

1. Do This

Work through those "Exercises for the Reader" on encoding products and sums in System F.

2. Write This

The outline draft for your written coursework assignment. Aim to have by Monday's lecture *either* your "Hello, World!" screenshot in the Example section *or* a paragraph for each of three references in the Resources section.

Extensions

Pick a strongly-typed programming language then try writing (and typechecking) two sorters, two testers, and code to testManySorters.

Find out how { Java, Scala, C#, Haskell, ... } handles type { variance, bounds, quantification, kinds }.

Review

Polymorphic code can be used with values of different types. Method invocation in object-oriented languages can demonstrate *ad-hoc polymorphism*, where code does different things at different types; while *parametric polymorphism* (or *generics*) describes code that does the same thing at different types.

Parameterized types like Queue<String> express families of types with common structure, applying a type constructor to one or more type parameters.

Behavioural subtyping is based on Liskov's principle of substitutivity that S is a subtype of T if and only if any S can be used in place of a T.

Variance describes subtyping for parameterized types, where type parameters may be covariant, contravariant or invariant.

Review

Hindley-Milner is a way to type parametric polymorphism in the lambda-calculus, introducing type schemes to generalise types and let-binding syntax to use polymorphic functions.

Type Inference makes it possible to write strongly-typed polymorphic code that is expressive but uncluttered by type annotations; while "Algorithm W" automatically identifies a principle type that is the most general type possible for a term.

Rank-2 types and beyond describe things not reachable in Hindley-Milner: while type schemes can describe a whole collection of types, sometimes we want to go further and have functions demand arguments that are themselves polymorphic.

Review

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 $\mathsf{Prod}\,X\,Y = \forall Z.((X \to Y \to Z) \to Z)$.

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

Outline

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 : $* \to *$	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:

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

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.

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

Example: Safe Indexing

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

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

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

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

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

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

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 $\quad (\lambda x{:}A.M)\ N \longrightarrow M[N/x]$

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.

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

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

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

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

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

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

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

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.

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

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
$\forall x{:}A.B$
$\Pi x{:}A.B$
$(x{:}A) \to B$
$\{x{:}A\}B$

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

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.

```
data LamType  = I | O | Fun LamType LamType
data LamVar   = MkVar String

data LamTerm  = LamV LamVar | LamI Int | LamO Bool
              | LamLam LamVar LamTerm | LamApp LamTerm LamTerm
```

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

```
lamcheck : LamTerm -> Bool
```

This recursively traverses the term to make sure all values are used with their correct types.

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.

$$\begin{array}{rl} \mathsf{i}, \mathsf{o} : \mathsf{LamType} & \mathsf{lamv} : \mathsf{LamVar} \to \mathsf{LamTerm} \\ \mathsf{fun} : \mathsf{LamType} \to \mathsf{LamType} \to \mathsf{LamType} & \mathsf{lami} : \mathsf{Num} \to \mathsf{LamTerm} \\ & \mathsf{lamo} : \mathsf{Bool} \to \mathsf{LamTerm} \\ \mathsf{mkvar} : \mathsf{String} \to \mathsf{LamVar} & \mathsf{lamlam} : \mathsf{LamVar} \to \mathsf{LamTerm} \to \mathsf{LamTerm} \\ & \mathsf{lamapp} : \mathsf{LamTerm} \to \mathsf{LamTerm} \to \mathsf{LamTerm} \end{array}$$

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

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

This 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: with LamVar(t) and LamTerm(t) recording the type t of the variable or term being represented.

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

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

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

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

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

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

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

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

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

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:

$$\begin{array}{lll} \text{true} : \text{Prop} & \text{and} : \text{Prop} \to \text{Prop} \to \text{Prop} & \text{not} : \text{Prop} \to \text{Prop} \\ \text{false} : \text{Prop} & \text{or} : \text{Prop} \to \text{Prop} \to \text{Prop} & \text{imp} : \text{Prop} \to \text{Prop} \to \text{Prop} \end{array}$$

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

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

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

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.

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

Composing these in appropriate combinations reveals an equivalence of types:

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

Informally, having a proof of $(\mathrm{P} \wedge \mathrm{Q})$ 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:

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

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, including the *Edinburgh Logical Framework*, *Twelf*, *Alf*, and *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*, extensive work on dependently-typed *Haskell*, and F*.

Introduction

F* (pronounced F star) is a general-purpose functional programming language with effects aimed at program verification. It puts together the automation of an SMT-backed deductive verification tool with the expressive power of a proof assistant based on dependent types. After verification, F* programs can be extracted to efficient OCaml, F#, or C code. This enables verifying the functional correctness and security of realistic applications, such as a verified HTTPS stack.

Verified programming in F*

A tutorial

The F* Team
MSR, MSR-Inria, Inria

Contents


```
1  module Welcome
2
3  (*
4  This interface allows you to edi
5
6  For convenience, you can resize
7  Click on the border between fram
8
9  Any change you make on the edito
10 browser. Your local state gets r
11 until your browser data is clear
12 verify the contents of the edito
13
14 This editor lets you work on mul
15 on the green  button in the bar
16 Files can be deleted with the re
```

Welcome

```
Ready to verify some F* code.
```

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 $F^\star$.

Homework

1. Read This

The instructions for the APL written coursework assignment. https://is.gd/apl18coursework

2. Write This

The outline draft for your assignment, following in detail the instructions above.

Ada Lovelace Day 2018

Murchison House, King's Buildings

Tuesday 9 October 2018

https://is.gd/lovelace2018

Sydney Padua: 2D Goggles

11.00–2.30	Talks, activities, technology, building things
2.30–5.30	Women in STEM Wikipedia edit-a-thon
5.45–7.30	Discussion and networking event

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

Elm:Taming the Wild West Web using Functional Programming

Rupert Smith

7pm Tuesday 9 October 2018

The Outhouse, 12A Broughton Street Lane

http://www.edlambda.co.uk

This is the monthly meetup of EdLambda, an Edinburgh group for people interested in functional programming. APL students are very welcome to come along.