Quest

Luca Cardelli

Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

Outline

1. *Introduction*
2. *The language*
3. *Polymorphism*
4. *Abstract Types*
5. *Subtypes*
6. *Modules*
7. *Conclusions*

part 1

Introduction

Procedural language paradigms

Untyped λ-calculus *(Church)*

$\lambda(x)b \qquad f(a)$

First-order typed λ-calculus *(Church)*

$(\lambda(x^{A}{:}A)b^{B})^{A\to B} \qquad (f^{A\to B}(a^{A}))^{B}$

Polymorphic typed λ-calculus *(Hindley, Milner)*

$(\lambda(x^{A\{\alpha\}})b^{B\{\alpha\}})^{A\{\alpha\}\to B\{\alpha\}} \qquad (f^{A\{\alpha\}\to B\{\alpha\}}(a^{A\{C\}}))^{B\{C\}}$

Second-order typed λ-calculus *(Girard, Reynolds)*

$(\Lambda(t)\ \lambda(x^{A\{t\}}{:}A)b^{B\{t\}})^{\forall t\ A\{t\}\to B\{t\}}$

$(f^{\forall t\ A\{t\}\to B\{t\}}[C](a^{A\{C\}}))^{B\{C\}}$

Intuitionistic type theory *(Martin-Löf, Coquand&Huet)*

$(\lambda(t^{T}{:}T)\ \lambda(x^{A\{t\}}{:}A)b^{B\{t\}\{x\}})^{\forall(t{:}T)\forall(x{:}A\{t\})B\{t\}\{x\}}$

$(f^{\forall(t{:}T)\forall(x{:}A\{t\})B\{t\}\{x\}}(C)(a^{A\{C\}}))^{B\{C\}\{a\}}$

Programming language semantics

*Goal: reduce programming concepts
to mathematical concepts.*

Approach in these lectures:

*Programming Language:
Quest (used in examples)*

*Core Language:
Quest Kernel (used in explaining type rules)*

*Technique:
operational semantics
(variations of Martin-Löf's type theory)*

*Commitment:
use a single framework for explaining
polymorphism, abstract types, inheritance
and modularization.*

For presentation reasons:
(A) We will not provide typing rules for the full language (rely on translation to core language).
(B) We will not provide explicit translation of full language to core language (rely on similarities and intuition).

part 2

The Language

A language in the tradition of...

Simula	*(subtyping)*
CLU	*(abstract types)*
ML	*(polymorphism)*
Mesa, Modula-2	*(modules)*
Russell, Pebble	*(dependent types)*

We now have type systems able to interpret all these features in a single framework. Based on variations over Martin-Loef 's intuitionistic type theory.

The bad news: typechecking is undecidable.

The good news: typechecking is practical.

Quest Kernel Types

Type	*The type of all types*
Power(A)	*The type of all subtypes of A* *subtyping*
All(x:A) B	*Dependent function types* *functions, polymorphism, param. modules*
Some(x:A) B	*Dependent pair types* *tuple, bindings, abstract types*
Record t1:A1,…,tn:An end	*Record types* *records, objects, multiple inheritance*
Variant t1:A1,…,tn:An end	*Variant types* *enumerations, disjoint unions*
Rec(x:A) B	*Recursion* *recursive types and type operators*
Ref(A)	*Reference types* *state, side effects*

Kernel Syntax

x (identifiers) *t (tags)*
e (expressions) *d (definitions)*

e ::=

x |

Type |

Power(e) |

All(x: e) e |
fun(x: e) e |
e(e) |

Some(x: e) e |
pair(x = e: e) e: e |
lft(e) | *rht(e)* |

Record t: e, ... , t: e end |
record t = e, ... , t = e end |
e . t |

Variant t: e, ... , t: e end |
variant t.=.e : e.end |
case e | t(x: e) e ... | t(x: e) e end |

Ref(e) | *ref(e)* | *deref(e)* | *e := e* |

rec(x: e) e |

d ::= let x: e = e ;

Simple values

3+1;	*the value 4*
it;	*the last value, i.e. 4*

Simple declarations

```
let a = 3;                declare a to be a constant
let a:Int = 3;            the same, with type information
a;                        evaluate a

let var b = 3;            declare b to be a variable
b := 5;                   change b
b+3;                      = 8

let a = 3                 simultaneous declarations
and b = 5;

begin                     local declarations
   let a = 2*n;
   a+1;
end;                      result is 2*n+1
```

Function declarations

The successor function

```
let succ(x:Int):Int = x+1;
succ(0);
    = 1
```

A function of no arguments

```
let one():Int = 1;
one();
    = 1
```

A function of two arguments

```
let average(x,y:Int):Int =
    (x+y)/2;
average(3;5);
    = 4
```

A curried function

```
let twice(f:Int->Int)(y:Int):Int =
    f(f(y));
twice(succ)(3);
    = 5
```

Partial application

```
twice(succ);
it(3);
    = 5
```

Recursive declarations

Recursive functions

```
rec fact(n:Int):Int =
    if n is 0 then 1
    else n*fact(n-1) end;
```

Mutually recursive functions

```
rec f(a:Int):Int =
    if a is 0 then 0 else g(n-1) end
and g(b:Int):Int =
    if b is 0 then 0 else f(n-1) end;
```

Recursive values

```
rec self =
    tuple
        let b = 3;
        let f(n:Int):Int = n + self.b
    end;
```

Quest Syntax

Terminal symbols are in bold, non-terminals and meta-syntactic notation in roman.

	sequencing (strongest binding power)
… **|** …	*alternative (weakest binding power)*
[…]	*optionally*
{ … }	*zero or more times*
(…)	*grouping*
a(b)	*same as [a {b a}] (zero or more times a separated by b)*

top ::= *top level phrase*
  Component *ide* ***;***
    [***import*** *ide(****,****)* ***;*** *] [* ***parts*** *spec(****,****)* ***;*** *]*
    { spec ***;*** *}* ***end*** **|**
  component *ide* ***;*** ***implements*** *ide* ***;***
    [***import*** *spec(****,****)* ***;*** *] [* ***parts*** *(ide* **=** *ide)(****,****)* ***;*** *]*
    { phrase ***;*** *}* ***end*** **|**
  phrase ***;***

phrase ::= *phrase*
  def **|**
  exp

def ::= *definition*
Let *decl* | **Rec** *decl* |
let *decl* | **rec** *decl*

decl ::= *declaration*
bind **=** *exp* |
decl **and** *decl*

bind ::= *binding*
[**var**] *spec'* |
(([**var**] *spec'*) (**;**) **)** |
ide { **(** ([**var**] *spec*)(**;**) **)** } [*qual*]

spec ::= *ident. specification*
ide qual

spec' ::= *optional ident. specification*
ide [*qual*]

qual ::= *qualification (has type, subtype of)*
(**:** | **<:**) *exp*

block ::= *block*
{ *phrase* **;** } *exp* [**;**]

exp ::= *expression*

identifiers

ide |

ide **:=** *exp* |

types

Type |

subtypes

Power (*exp* ***)*** |

booleans

Bool | ***true*** | ***false*** |

/ | **\\/** | ***not*** | **==** | **#** |

if *exp* ***then*** *block* { ***elsif*** *exp* ***then*** *block* }

[***else*** *block*] ***end*** |

loop (*phrase* | ***while*** *exp*)(**;**) [**;**] ***end*** |

characters

Char | **'** *char* **'** |

ascii | **==** | **#** |

integers

Int | *exp* **..** *exp* | *integer* |

real | ***char*** | ***minus*** | **+** | **-** | ***** | **/** | **%** |

< | **>** | **<=** | **>=** | **==** | **#** |

reals

Real | *real* |

floor | ***minus*** | **+** | **-** | ***** | **/** |

< | **>** | **<=** | **>=** | **==** | **#** |

strings

String | *string* |

string | ***length*** | ***getChar*** | ***putChar*** | ***sub*** |

setSub | ***move*** | ***search*** | **<>** | **==** | **#** |

arrays

Array | ***array*** | ***size*** | ***index*** | ***update*** |

for *ide* ***from*** *exp* (***to*** | ***downto***) *exp* **do** *exp* ***end*** |

records

Record ([***var***] *spec*)(**;**) ***end*** | ***And*** | ***Ignoring*** |

record (*bind* **=** *exp*)(**;**) ***end*** | ***and*** | ***ignoring*** |

exp **.** *ide* | *exp* **.** *ide* **:=** *exp* |

variants

Variant ([***var***] *spec*)(**;**) ***end*** | ***Or*** | ***Forgetting*** |

variant *bind* **=** *exp* ***end*** |

case [*spec'* **=**] *exp* [***gives*** *exp*]

{ **|** *ide* **(** *spec'* (**;**) **)** *block* } [***else*** *block*] ***end*** |

exp ***as*** *ide* := *exp* |

functions

All **(** *spec*(**;**) **)** *exp* |

exp **(** ([*bind* **=**] *exp*)(**;**) **)** |

tuples

Unit | ***Tuple*** ([***var***] *spec*)(**;**) ***end*** |

unit | ***tuple*** ([*bind* **=**] *exp*)(**;**) ***end*** |

exp **.** *ide* | *exp* **.** *ide* **:=** *exp* |

grouping

(*exp* **)** | ***begin*** *block* ***end*** |

specs

exp qual |

dynamics

Dynamic | ***dynamic*** *exp* | ***coerce*** *exp* **to** *exp* |

persistency

extern | ***intern*** |

exceptions

  raise *signal [qual]* |

  try *exp { **on** signal block } [**finally** block]* ***end*** |

qualified

  exp **.** *ide* |

  exp **.** *ide* **:=** *exp* |

  exp **^** *ide* |

  exp **^** *ide* **:=** *exp* |

part 3

Polymorphism

Type declarations

The unit type

```
Let Unit:Type = Tuple end;
let unit:Unit = tuple end;
```

An integer pair type

```
Let IntPair:Type =
  Tuple fst:Int; snd:Int end;
```

A pair of that type

```
let p:IntPair =
  tuple fst=3; snd=4 end;
```

The integer function type

```
Let IntFun:Type = Int->Int;
```

..a function of that type

```
let f:IntFun = succ;
```

Type operators

Cartesian product

```
Let # (A:Type)(B:Type):Type =
  Tuple fst:A; snd:B end;
```

Function space

```
Let -> (A:Type)(B:Type):Type =
  All(x:A) B;
```

Homogeneous lists

```
Rec List(A:Type):Type =
  Variant
    nil: Unit;
    cons: Tuple hd:A; tl:List(A) end;
end;
```

Polymorphic functions

The type of the integer identity

```
Let IntId:Type =
  All(x:Int) Int;
```

The integer identity

```
let intId:IntId =
  fun(x:Int) x;
```

Usage of integer identity

```
intId(3);
```

The type of the polymorphic identity

```
  Let Id:Type =
    All(A:Type) All(a:A) A;
```

The polymorphic identity

```
  let id(A:Type)(a:A):A = a;
```

..application of a polymorphic function

```
  id(Int)(3);
```

..abbreviated application

`id(3);` *where the missing* `Int` *parameter can be inferred*

Specialized identities

```
  let intId: Int->Int = id(Int);
  let boolId: Bool->Bool = id(Bool);
```

Passing polymorphic functions

```
let f(g:Id): Int#Bool =
  tuple g(3); g(true) end;
```

i.e. `g(Int)(3)`, *etc.*

The polymorphic swap function

```
let swap(A:Type; B:Type)(p:A#B)
      : B#A =
  tuple p.snd; p.fst end;
```

..usage

```
swap(3; true);
```

i.e. `swap(Int;Bool)(3;true)`

Polymorphic lists

```
exception hd, tl;

Rec List(A:Type):Type =
  Variant
    nil: Unit;
    cons: Tuple hd:A; tl:List(A) end;
   end;

let nil(A:Type):List(A) =
  variant nil=unit end;

let cons(A:Type)(hd:A; tl:List(A))
    :List(A) =
  variant
    cons = tuple hd; tl end
  end;

let null(A:Type)(a:List(A)):Bool =
  case a
  | nil() true
  | cons(hd; tl) false
  end;
```

```
let head(A: Type)(a: List(A)):A
    raises hd =
  case a
  | nil() raise hd
  | cons(hd; tl) hd
  end;

let tail(A:Type)(a:List(A)):List(A)
    raises tl =
  case a
  | nil() raise tl
  | cons(hd; tl) tl
  end;

rec length(A:Type)(a:List(A)):Int =
  case a
  | nil() 0
  | cons(hd; tl) 1+length(tl)
  end;

rec map(A,B:Type)
     (f:A->B)(a:List(A)):List(B) =
  case a
  | nil() nil(B)
  | cons(hd:A; tl:List(A))
     cons(f(hd); map(f)(tl))
  end;
```

part 3A

Type Rules

Judgements

A well-formed signature S *provides a set of typed constants.*

$\vdash$ S sig

A well-formed environmene E*, over a signature* S*, provides a set of typed variables.*

$\vdash_S$ E env

In the signature S *and environment* E*, we can deduce that* a *has type* A.

E $\vdash_S$ a: A

Inferences

$$\frac{J_1 \quad \ldots \quad J_n}{J} \qquad (P)$$

Signatures

Signature Construction

$$\frac{}{\vdash \Diamond \text{ sig}}$$

$$\frac{\vdash S \text{ sig} \qquad \vdash_S A{:}\text{Type} \qquad c \notin \text{Dom}(S)}{\vdash S, c{:}A \text{ sig}}$$

Environments

Environment Construction

$$\frac{\vdash S \text{ sig}}{\vdash_S \Diamond \text{ env}}$$

$$\frac{\vdash_S E \text{ env} \qquad \vdash_S A{:}\text{Type} \qquad x \notin \text{Dom}(E)}{\vdash_S E, x{:}A \text{ env}}$$

Constants

Given

$$\frac{\vdash_S E\ \mathrm{env} \qquad c{:}A \in S}{E \vdash_S c{:}A}$$

Variables

Assumption

$$\frac{\vdash_S E\ \mathrm{env} \qquad x{:}A \in E}{E \vdash_S x{:}A}$$

Type

Type Formation

$$\frac{\vdash_S E\ \mathrm{env}}{E \vdash_S \mathrm{Type} : \mathrm{Type}}$$

Functions

$\rightarrow$ Formation

$$\frac{E \vdash_S A{:}\mathrm{Type} \qquad E \vdash_S B{:}\mathrm{Type}}{E \vdash_S A \rightarrow B : \mathrm{Type}}$$

$\rightarrow$ Introduction

$$\frac{E \vdash_S A{:}\mathrm{Type} \qquad E, x{:}A \vdash_S b{:}B \qquad x \notin B}{E \vdash_S \mathrm{fun}(x{:}A)\ b : A \rightarrow B}$$

$\rightarrow$ Elimination

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S b{:}\ A \rightarrow B}{E \vdash_S b(a){:}\ B}$$

Dependent functions

All Formation

$$\frac{E \vdash_S A{:}Type \qquad E, x{:}A \vdash_S B{:}Type}{E \vdash_S All(x{:}A)\ B : Type}$$

All Introduction

$$\frac{E \vdash_S A{:}Type \qquad E, x{:}A \vdash_S b{:}B}{E \vdash_S fun(x{:}A)\ b : All(x{:}A)\ B}$$

All Elimination

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S b{:}\ All(x{:}A)\ B}{E \vdash_S b(a){:}\ B\{x \leftarrow a\}}$$

Recursion

Rec Formation

$$\frac{E \vdash_S A{:}\mathrm{Type} \quad E, x{:}A \vdash_S a{:}A}{E \vdash_S \mathrm{rec}(x{:}A)\ a : A}$$

Type computations

Conversion

$$\frac{E \vdash_S a{:}A \quad E \vdash_S B{:}\mathrm{Type} \quad A =_{\beta\eta\mu} B}{E \vdash_S a{:}B}$$

Examples

$$\dfrac{\dfrac{E, A{:}Type, x{:}A \vdash_S x{:}A}{E, A{:}Type \vdash_S fun(x{:}A)\ x : All(x{:}A)\ A}}{E \vdash_S fun(A{:}Type)\ fun(x{:}A)\ x : All(A{:}Type)\ All(x{:}A)\ A}$$

let id *be* fun(A:Type) fun(x:A) x

$$\dfrac{E \vdash_S 3{:}Int \qquad \dfrac{E \vdash_S Int{:}Type \qquad E \vdash_S id{:}\ All(A{:}Type)\ All(x{:}A)\ A}{E \vdash_S id(Int){:}\ All(x{:}Int)\ Int}}{E \vdash_S id(Int)(3){:}\ Int}$$

part 4

Abstract Types

Tuples

A triple and one of its types

```
tuple 3; true; 'c' end ;
   : Tuple Int; Bool; Char end
```

A labeled pair and two of its labeled types

```
tuple a=3; b=true end;
   : Tuple a:Int; b:Bool end
   : Tuple x:Int; y:Bool end
```

A dependent pair and two of its types

```
tuple A:Type=Int; b:A=3 end;
   : Tuple A:Type; b:A end
   : Tuple B:Type; c:B end
```

A labeled pair with type info

```
tuple a:Int=3; b:Bool=true end;
```

Tuple declarations

Binding a pair

```
let p = tuple a=3; b=true end;
  p = tuple a=3; b=true end
```

Splitting a pair

```
let (x:Int; y:Bool) = p;
  x = 3
  y = true
```

Matching a pair

```
let (x; y) = tuple a=3; b=true end;
  x = 3
  y = true
```

Tuple selection

Selecting a field

```
let p = tuple a=3; b=true end;
p.a;
  = 3
```

... abbreviation of

```
begin let (x; y) = p; x end;
```

Tuples and functions

A function expecting a pair

```
let f(x: Tuple a,b:Int end):Int =
  x.a+x.b;
```

Two legal applications

```
let p = tuple 3; 5 end;
f(p);
f(3;5);
```

A tuple with function components

```
tuple
  let succ(n:Int):Int = n+1;
  let plus(n,m:Int):Int = n+m;
end;
```

Selection and application

```
it.succ(3);
```

Abstract types

A signature (interface)

```
Let Alg: Type =
  Tuple
    T: Type;               (abstract type)
    obj: T;                (constants)
    op: T -> Int;          (operations)
  end;
```

An algebra (implementation)

```
let alg1 =
  tuple
    Let T:Type = Int;  (hidden representation)
    let obj:T = 0;
    let op:T->Int = succ;
  end;
```

Another implementation

```
let alg2 =
  tuple
    Let T:Type = List(Int);
    let obj:T = nil(Int);
    let op:T->Int = length(Int);
  end;
```

A function operating on any implementation of the interface

```
let fun(alg:Alg):Int =
  alg.op(alg.obj);
```

Information hiding (and lack of)

Note that `alg1` *can be defined totally independently of* `Alg`*. As a consequence, a package can implement more than one abstract type as long as its type matches the abstract type (this will come handy for modules).*

```
Let Alg': Type =
  Tuple
    D:Type; a:D; f:D->Int;
  end;
```

`Alg'`*matches* `Alg` *(by α-conversion) Hence* `alg1:Alg'`*, although* `Alg'` *was defined* ***after*** `alg1` *was created.* `Alg'` *can* ***impersonate*** `Alg`*. Sometimes non-impersonation is a required characteristics of abstract types. This is not the case here.*

Moreover, if `alg1` *and* `alg1'` *are both implementations of* `Alg` *based on* `Int`*, then* `alg1.C` *matches* `alg1'.C`*, and they both match* `Int`*. There is no **information hiding** when the packages are **manifest**.*

`alg1.obj + 1;`	*Legal!* `alg.C = Int`
`alg1.op(alg1'.obj);`	*Legal!* `alg1.C = alg1'.C`

***Non-generative** abstract types (matched by structure) have some advantages over **generative** ones (matched by name): e.g.: (a) automatic support of multiple implementations, (b) possibility of storing abstract objects beyond the life-span of a single program run, (c) in interactive systems, reloading an abstract type definition does not invalidate old objects.*

With the above disclaimers, the language does support information hiding, through ordinary λ-abstraction. Privacy of representation is automatically enforced when writing a function wishing to use ***any*** *implementation of a given abstract type.*

```
let fun(alg:Alg):Int =
  alg.obj + 1;              Illegal!
                          alg.C ≠ Int

let fun(alg1,alg2:Alg):Int =
  alg1.op(alg2.obj);    Illegal!
                          alg1.C ≠ alg2.C

let fun(alg:Alg; a,b:alg.C):Int =
  alg.op(a)+alg.op(b);   Legal!
                          alg.C = alg.C
```

Absolute privacy of implementation must be guaranteed by actually ***hiding*** *the implementation and making only its name and type available. This requires mechanisms outside the ones presented so far.*

The important point is that a package using another package should refer to it is at parameter, not as a global variable. Normally, this is automatically enforced by modularization and separate-compilation mechanisms.

Parametric abstract types

```
Let StackPackage(Item:Type):Type =
  Tuple
    Stack: Type->Type;
    empty: Stack(Item);
    isEmpty: Stack(Item)->Bool;
    push: (Stack(Item)#Item)->
            Stack(Item);
    top: Stack(Item)->Item;
    pop: Stack(Item)->Stack(Item)
  end;

let StackFromList(Item:Type)
      : StackPackage(Item) =
  tuple
    Let Stack: Type->Type =
      List;
    let empty: Stack(Item) =
      nil(Item);
    let isEmpty: Stack(Item)->Bool =
     null(Item);
    let push: (Stack(Item)#Item)->
            Stack(Item) =
      cons(Item);
    let top: Stack(Item)->Item =
      head(Item);
    let pop: Stack(Item)->Stack(Item) =
      tail(Item);
  end;
```

```
Let IntStack:Type =
  StackPackage(Int);

let IntStackFromList:IntStack =
  StackFromList(Int);

IntStackFromList.push(
  IntStackFromList.empty;
  0);



Let GenericStack:Type =
  All(Item:Type) StackPackage(Item);

let f(genericStack:GenericStack): Int =
  begin
    let p = genericStack(Int);
    p.top(p.push(3;p.empty))
  end;

f(StackFromList);
```

part 4A

Type Rules

Pairs

$\times$ Formation

$$\frac{E \vdash_S A{:}\mathrm{Type} \qquad E \vdash_S B{:}\mathrm{Type}}{E \vdash_S A{\times}B : \mathrm{Type}}$$

$\times$ Introduction

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S b{:}B}{E \vdash_S \langle a,b\rangle : A{\times}B}$$

$\times$ Elimination

$$\frac{E \vdash_S c{:}A{\times}B}{E \vdash_S \mathrm{lft}(c){:}\ A} \qquad \frac{E \vdash_S c{:}A{\times}B}{E \vdash_S \mathrm{rht}(c){:}\ B}$$

Dependent pairs

Some Formation

$$\frac{E \vdash_S A{:}\mathrm{Type} \qquad E,\ x{:}A \vdash_S B{:}\mathrm{Type}}{E \vdash_S \mathrm{Some}(x{:}A)\ B : \mathrm{Type}}$$

Some Introduction

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S b\{x{\leftarrow}a\}{:}B\{x{\leftarrow}a\}}{E \vdash_S \mathrm{pair}(x{:}A{=}a)\ b{:}B : \mathrm{Some}(x{:}A)\ B}$$

Some Elimination

$$\frac{E \vdash_S c{:}\ \mathrm{Some}(x{:}A)\ B}{E \vdash_S \mathrm{lft}(c){:}\ A} \qquad \frac{E \vdash_S c{:}\ \mathrm{Some}(x{:}A)\ B}{E \vdash_S \mathrm{rht}(c){:}\ B\{x{\leftarrow}\mathrm{lft}(c)\}}$$

Examples

$$\dfrac{E \vdash_S \text{Int:Type} \qquad \dfrac{E \vdash_S \ 0\text{:Int} \qquad E \vdash_S \ \text{succ:Int}\rightarrow\text{Int}}{E \vdash_S \ \text{<0,succ>} \ : \ \text{Int}\times(\text{Int}\rightarrow\text{Int})}}{\begin{array}{c} E \vdash_S \ \text{pair(A:Type=Int) <0,succ>:A}\times(\text{A}\rightarrow\text{Int}) \\ : \ \text{Some(A:Type) A}\times(\text{A}\rightarrow\text{Int}) \end{array}}$$

let pack *be* pair(A:Type=Int) <0,succ>:A×(A→Int)

$$\dfrac{E \vdash_S \text{pack: Some(A:Type) A}\times(\text{A}\rightarrow\text{Int})}{E \vdash_S \text{rht(pack): lft(pack)}\times(\text{lft(pack)}\rightarrow\text{Int})}$$

let Abs *be* lft(pack) *and* op *be* rht(pack)

$$\dfrac{\dfrac{E \vdash_S \text{op: Abs}\times(\text{Abs}\rightarrow\text{Int})}{E \vdash_S \text{lft(op):Abs}} \qquad \dfrac{E \vdash_S \text{op: Abs}\times(\text{Abs}\rightarrow\text{Int})}{E \vdash_S \text{rht(op): Abs}\rightarrow\text{Int}}}{E \vdash_S \text{rht(op)(lft(op)): Int}}$$

part 5

Subtypes

Multiple inheritance

A multiple inheritance hierarchy

```
Let Object:Type =
  Record var age:Int end;

Let Vehicle:Type =
  Object And
    Record speed:Int end;

Let Machine:Type =
  Object And
    Record fuel:String end;

Let Car:Type =
  Vehicle And Machine;
```

$$\begin{array}{ccccc} & \le & \textit{Vehicle} & \le & \\ \textit{Car} & & & & \textit{Object} \\ & \le & \textit{Machine} & \le & \end{array}$$

The same hierarchy

```
Let Object:Type =
  Record var age:Int end;

Let Car:Type =
  Object And
    Record speed:Int;fuel:String end;

Let Vehicle:Type =
  Car Ignoring speed;

Let Machine:Type =
  Car Ignoring fuel;
```

Objects

```
let myObj:Object =
  record var age = 3 end;

let myCar:Car =
  record
    var age = 4;
    fuel = "Gasoline";
    speed = 140;
  end;
```

Increasing the age

```
let older(obj:Object):Object =
  begin
    obj.age := obj.age+1;
    obj;
  end;
```

`older(myObj);` *ok*

`older(myCar);` *by subtyping,* `mycar:Obj`

Classes and methods

A class with an instance variable and two methods, one of which returns self:

```
Rec Counter:Type =
  Record
    var count: Int;
    fetch: Unit -> Int;
    incr: Int -> Counter;
  end;
```

Creating an instance of that class:

```
let newCounter(init:Int):Counter =
  rec self:Counter =
    record
      let var count = init;
      let incr(n:Int):Unit =
        begin
          self.count := self.count + n;
          self;
        end;
      let fetch():Int =
        self.count;
    end;
```

Cascading operations:

```
newCounter(0).incr(3).fetch();
```

We have accounted for:

Methods

Message passing

Object instantiation

Private instance variables

Self

Cascading operations

Parametric inheritance

The simple aging function

```
let older(obj:Object):Object =
  begin
    obj.age := obj.age+1;
    obj;
  end;

older(myCar);
```

the result type is Object
unwanted loss of type information

The parametric aging function ("<:" reads "subtype of")

```
let older(A<:Object)(obj:A):A =
  begin
    obj.age := obj.age+1;
    obj;
  end;

older(Car)(myCar);
older(myCar);
older(myCar).speed;
```

`older(Car)(myCar);` — *the result type is now* `Car`
`older(myCar);` — *the same, abbreviated*
`older(myCar).speed;` — *this works*

Power types

`Power(A)` *is the type of all subtypes of* `A`

`C: Power(A)` *means* `C` $\leq$ `A`
(written `C <: A`*)*

```
All(x<:A) B        (i.e. All(x:Power(A)) B)

fun(x<:A) B

Tuple ...; x<:A; ... end

tuple ...; x<:A = a; ... end
```

Higher-order inheritance

```
let speed: Vehicle->Int = ... ;

  Vehicle->Int   ≤   Car->Int
```
(speed takes cars)

```
let serialNo: Int->Car = ... ;

  Int->Car   ≤   Int->Vehicle
```
(serialNo returns vehicles)

```
let f: Vehicle->Vehicle = ...

  Vehicle->Vehicle   ≤   Car->Object

age(f(myCar));          (used as Car->Object)
```

```
let f': All(A<:Vehicle) A->A = ... ;

age'(Car)(f'(Car)(myCar));
age'(f'(myCar));
```
(abbreviated)

Static arrays

```
Array: All(A<:Int) All(B:Type) Type

array:  All(A<:Int) All(B:Type)
           All(b:B) Array(A)(B)

array(1..6)(Bool)(false):
 Array(1..6)(Bool)
```

part 5A

Type Rules

Record Formation

$$\frac{E \vdash_S A_1\text{: Type} \quad \ldots \quad E \vdash_S A_n\text{: Type}}{E \vdash_S \text{Record } t_1{:}A_1 \ldots t_n{:}A_n \text{ end : Type}}$$

Record Introduction

$$\frac{E \vdash_S a_1\text{: } A_1 \quad \ldots \quad E \vdash_S a_n\text{: } A_n}{E \vdash_S \text{record } t_1{=}a_1 \ldots t_n{=}a_n \text{ end : Record } t_1{:}A_1 \ldots t_n{:}A_n \text{ end}}$$

Record Elimination

$$\frac{E \vdash_S r \text{ : Record } t_1{:}A_1 \ldots t_n{:}A_n \text{ end}}{E \vdash_S r.t_i\text{: } A_i}$$

Power Formation

$$\frac{E \vdash_S A{:}\mathrm{Type}}{E \vdash_S \mathrm{Power}(A) : \mathrm{Type}}$$

Power Introduction

$$\frac{E \vdash_S A{:}\mathrm{Type}}{E \vdash_S A \leq A}$$

Power Elimination

$$\frac{E \vdash_S a{:}A \quad E \vdash_S A \leq B}{E \vdash_S a{:}\ B}$$

Power Record

$$\frac{E \vdash_S A_1 \leq B_1 \quad \ldots \quad E \vdash_S A_n \leq B_n \quad \ldots \quad E \vdash_S A_m\text{: Type}}{E \vdash_S \text{Record } t_1{:}A_1 \ldots t_n{:}A_n \ldots t_m{:}A_m \text{ end} \leq \text{Record } t_1{:}B_1 \ldots t_n{:}B_n \text{ end}}$$

Power Variant

$$\frac{E \vdash_S A_1 \leq B_1 \quad \ldots \quad E \vdash_S A_n \leq B_n \quad \ldots \quad E \vdash_S B_m\text{: Type}}{E \vdash_S \text{Variant } t_1{:}A_1 \ldots t_n{:}A_n \text{ end} \leq \text{Variant } t_1{:}B_1 \ldots t_n{:}B_n \ldots t_m{:}B_m \text{ end}}$$

Power $\to$

$$\frac{E \vdash_S A' \leq A \quad E \vdash_S B \leq B'}{E \vdash_S A \to B \leq A' \to B'}$$

Power All

$$\frac{E \vdash_S A' \leq A \quad E, x{:}A' \vdash_S B \leq B'}{E \vdash_S \mathrm{All}(x{:}A)\ B \leq \mathrm{All}(x{:}A')\ B'}$$

Power $\times$

$$\frac{E \vdash_S A \leq A' \quad E \vdash_S B \leq B'}{E \vdash_S A \times B \leq A' \times B'}$$

Power Some

$$\frac{E \vdash_S A \leq A' \quad E, x{:}A \vdash_S B \leq B'}{E \vdash_S \mathrm{Some}(x{:}A)\ B \leq \mathrm{Some}(x{:}A')\ B'}$$

Power Type

$$\frac{E \vdash_S A : \mathrm{Type}}{E \vdash_S \mathrm{Power}(A) \le \mathrm{Type}}$$

Power Power

$$\frac{E \vdash_S A \le B}{E \vdash_S \mathrm{Power}(A) \le \mathrm{Power}(B)}$$

part 6

Modules

Universally closed modules (Pebble)

```
Let Point:Type =
  Tuple
    T: Type;
    new: (Real # Real) -> T;
    x: T -> Real;
    y: T -> Real;
  end;

let CartesianPointMod:Point =
  tuple
    Let T:Type =
      Tuple x: Real; y: Real end;
    let new(a:Real; b:Real):T =
      tuple x=a; y=b end;
    let x(p:T):Real = p.x;
    let y(p:T):Real = p.y;
  end;

let PolarPointMod:Point =
  ...
```

```
Let Circle(APoint:Point):Type =
  Tuple
    T: Type;
    new: (APoint.T # Real) -> T;
    center: T -> APoint.T;
    radius: T -> Real;
  end;

let CircleMod(APoint:Point)
      :Circle(APoint) =
  tuple
    Let T:Type =
      Tuple
        center: APoint.T;
        radius: Real;
      end;
    let new(c:APoint.T; r:Real):T =
      tuple center=c; radius=r end;
    let center(c:T):APoint.T =
      c.center;
    let radius(c:T):Real =
      c.radius;
  end;
```

```
Let Square(APoint:Point):Type =
  Tuple
    T: Type;
    new: (APoint.T # Real) -> T;
    northWest: T -> APoint.T;
    southEast: T -> APoint.T;
  end;

let SquareMod(APoint:Point)
      : Square(APoint) =
  tuple
    Let T:Type =
      Tuple
        northWest: APoint.T;
        southEast: APoint.T;
      end;
  let new(middle:APoint.T;
          radius:APoint.T):T =
    tuple
      northWest= ...;
      southEast= ...;
    end;
  let northWest(r:T):APoint.T =
    r.northWest;
  let southEast(r:T):APoint.T =
    r.southEast;
end;
```

```
Let Geometry(
    APoint:Point;
    ACircle:Circle(APoint);
    ASquare: Square(APoint))
    :Type =
  Tuple
    boundingSquare:
      ACircle.T -> ASquare.T;
  end;

let GeometryMod(
    APoint:Point;
    ACircle:Circle(APoint);
    ASquare:Square(APoint))
    :Geometry(APoint;ACircle;ASquare) =
  tuple
    let boundingSquare(c:ACircle.T)
        : ASquare.T =
      ASquare.new(
        ACircle.center(c);
        ACircle.radius(c))
  end;
```

Note that `boundingSquare` *typechecks only because the circle and square modules are based on the* ***same*** *implementation of point. Suppose they were based on different implementations:*

```
ACircle: Circle(APoint1)
ACircle.center: ACircle.T -> APoint1.T

ASquare: Square(APoint2)
ASquare.new: APoint2.T # Real -> ASquare.T
```

then `APoint1.T` *would not match* `APoint2.T`.

Drawback of universally closed modules:
the interface of a module must mention the
whole import hierarchy of its imports!

```
let link(
    PointMod: Point;
    CircleMod: All(P:Point) Circle(P);
    SquareMod: All(P:Point) Square(P);
    GeometryMod:
      All(P:Point;
            C:Circle(P);R:Square(P))
        Geometry(P;C;R))
    : Geometry(PointMod;
        CircleMod(PointMod);
        SquareMod(PointMod)) =
  GeometryMod(
    PointMod;
    CircleMod(PointMod);
    SquareMod(PointMod));

let CartesianGeometry =
  link(CartesianPointMod,
    CircleMod; SquareMod; GeometryMod);

CartesianGeometry.boundingSquare(...);
```

Submodules

Suppose we define a subinterface of point:

```
ExtPoint <: Point;
```

and we define the parametric circle interface as:

```
Circle: All(P<:Point) Type;
```

Then, we can form the interface:

```
Circle(ExtPoint)
```

(although we do not have inclusions between `Circle(Point)` *and* `Circle(ExtPoint)` *because the P parameter appears on opposite sides of arrows in circle operations.)*

Existentially closed modules (Standard ML)

```
Let Point:Type =
  Tuple
    T: Type;
    new: (Real # Real) -> T;
    x: T -> Real;
    y: T -> Real;
  end;

let CartesianPointMod:Point =
  tuple
    Let T:Type =
      Tuple x: Real; y: Real end;
    let new(a:Real; b:Real):T =
      tuple x=a; y=b end;
    let x(p:T):Real = p.x;
    let y(p:T):Real = p.y;
  end;

let PolarPointMod:Point =
  ...
```

```
Let Circle:Type =
  Tuple
    APoint: Point;
    T: Type;
    new: (APoint.T # Real) -> T;
    center: T -> APoint.T;
    radius: T -> Real;
  end;

let CircleMod(P:Point):Circle =
  tuple
    Let APoint:Point = P;
    Let T:Type =
      Tuple
        center: APoint.T;
        radius: Real;
      end;
    let new(c:APoint.T; r:Real):T =
      tuple center=c; radius=r end;
    let center(c:T):APoint.T =
      c.center;
    let radius(c:T):Real =
      c.radius;
  end;
```

```
Let Square:Type =
  Tuple
    APoint: Point;
    T: Type;
    new: (APoint.T # Real) -> T;
    northWest: T -> APoint.T;
    southEast: T -> APoint.T;
  end;

let SquareMod(P:Point):Square =
  tuple
    Let APoint:Point = P;
    Let T:Type =
      Tuple
        northWest: APoint.T;
        southEast: APoint.T;
      end;
  let new(middle:APoint.T;
          radius:APoint.T):T =
    tuple
      northWest= ...;
      southEast= ...;
    end;
  let northWest(r:T):APoint.T =
    r.northWest;
  let southEast(r:T):APoint.T =
    r.southEast;
end;
```

```
Let Geometry:Type =
  Tuple
    ACircle: Circle;
    ASquare: Square;
    boundingSquare:
      ACircle.T -> ASquare.T;
  end;

let GeometryMod(
    C:Circle;
    R:Square)
    :Geometry =
  tuple
    Let ACircle:Circle = C;
    Let ASquare:Square = R;
    let boundingSquare(c:ACircle.T)
        : ASquare.T =
      ASquare.new(
        ACircle.center(c);
        ACircle.radius(c))
  end;
```

Advantage of existentially closed modules: the interface of a module only mentions its direct imports.

Drawback: boundingSquare does not typecheck!

```
ACircle: Circle
ACircle.center: ACircle.T -> ACircle.APoint.T

ASquare: Square
ASquare.new:
  ASquare.APoint.T # Real -> ASquare.T
```

then `ACircle.APoint.T` *does not match* `ASquare.APoint.T`*.*

*Solution: augment the type system to include **sharing constraints** of the form* `ACircle.APoint.T = ASquare.APoint.T`*. These are verified at link (application) time.*

Unfortunately, sharing contraints are as painful to maintain as the import hierarchy was for universally closed modules. Hence sharing contraints should be generated automatically, and verified automatically at link time.

```
let link(
    PointMod:Point;
    CircleMod: Point -> Circle;
    SquareMod: Point -> Square;
    GeometryMod:
      Circle#Square -> Geometry)
    : Geometry =
  GeometryMod(
    CircleMod(PointMod);
    SquareMod(PointMod));

let CartesianGeometry =
  link(CartesianPointMod,
    CircleMod; SquareMod; GeometryMod);

CartesianGeometry.boundingSquare(...);
```

Modules

```
Interface Point;
  T: Type;
  new: (Real # Real) -> T;
  x: T -> Real;
  y: T -> Real;
end;

Module CartesianPointMod;
implements Point;
  Let T:Type =
    Tuple x: Real; y: Real end;
  let new(a:Real; b:Real):T =
    tuple x=a; y=b end;
  let x(p:T):Real = p.x;
  let y(p:T):Real = p.y;
end;

Module PolarPointMod;
implements Point;
  ...
end;
```

```
Interface Circle;
import APoint: Point;
  T: Type;
  new: (APoint.T # Real) -> T;
  center: T -> APoint.T;
  radius: T -> Real;
end;

Module CircleMod;
implements Circle;
import APoint: Point;
  Let T =
    Tuple
      center: APoint.T;
      radius: Real;
    end;
  let new(c:APoint.T; r:Real):T =
    tuple center=c; radius=r end;
  let center(c:T):APoint.T =
    c.center;
  let radius(c:T):Real =
    c.radius;
end;
```

```
Interface Square;
import APoint: Point;
  T: Type;
  new: (APoint.T # Real) -> T;
  northWest: T -> APoint.T;
  southEast: T -> APoint.T;
end;

Module SquareMod;
implements Square;
import APoint: Point;
  Let Square =
    Tuple
      northWest: APoint.T;
      southEast: APoint.T;
    end;
  let new(middle:APoint.T;
          radius:APoint.T):T =
    tuple
      northWest= ...;
      southEast= ...;
    end;
  let northWest(r: Square): APoint.T =
    r.northWest;
  let southEast(r: Square): APoint.T =
    r.southEast;
end;
```

```
Interface Geometry;
import
    ACircle: Circle, ASquare: Square;
  boundingSquare:
    ACircle.T -> ASquare.T;
end;

Module GeometryMod;
implements Geometry;
import ACircle:Circle, ASquare:Square;
  let boundingSquare(c:ACircle.T)
      : ASquare.T =
    ASquare.new(
      ACircle.center(c);
      ACircle.radius(c))
end;
```

Assumes automatic generation and verification of sharing constranints.

```
System CartesianGeometry;
implements Geometry;
link
  CartesianPointMod: Point,
  SquareMod: Point -> Square,
  CircleMod: Point -> Circle,
  GeometryMod:
    Square#Circle -> Geometry;
end;
```

part 7

Conclusions

Conclusions

Dependent types, subtyping and recursive types account for a wide range of language features.

Their integration unifies functional and object-oriented programming in a typed framework.

The resulting type system, although undecidable, is effectively typecheckable. Typechecking is be based on a normal-order reducer for an extended lambda-calculus.

Stratification helps in distinguishing compile-time and run-time phases, and in introducing updatable state.

A prototype typechecker has been built which deals with dependent functions, dependent pairs, recursion, subtyping and limited type inference

Techniques

Typechecker

Reduction to head normal form, for matching.
Loop detection, for recursive types.
Unification, for inference.

Interpreter

Standard, throw-away.

Compiler

Interactive, bootstrapped.
Recursive descent, one-pass, in-core.
Closures.
Subtyping (method cacheing).
Stack retention analysis.
Producing bytecode (initially).

Linker

Components.
Subtyping (?).
Module sharing constraints (?).

Run-Time

RCMaps (heap and stack).
Pickling.
GC (Compacting).