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)

\[ \lambda(x)b \quad f(a) \]

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

\[ (\lambda(x^A:A)b^B) ^{A \rightarrow B} \quad (f^A \rightarrow B(a^A))^B \]

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

\[ (\lambda(x^A[x])b^B[x]) ^{A[x] \rightarrow B[x]} \quad (f^A[x] \rightarrow B[x](a^A[C]))^B[C] \]

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

\[ (\Lambda(t) \lambda(x^{A[t]}:A)b^{B[t]}) ^{\forall t A[t] \rightarrow B[t]} \]

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

**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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>The type of all types</td>
</tr>
<tr>
<td>Power(A)</td>
<td>The type of all subtypes of A</td>
</tr>
<tr>
<td>All(x:A) B</td>
<td>Dependent function types functions, polymorphism, param. modules</td>
</tr>
<tr>
<td>Some(x:A) B</td>
<td>Dependent pair types tuple, bindings, abstract types</td>
</tr>
<tr>
<td>Record t1:A1,..,tn:An end</td>
<td>Record types records, objects, multiple inheritance</td>
</tr>
<tr>
<td>Variant t1:A1,..,tn:An end</td>
<td>Variant types enumerations, disjoint unions</td>
</tr>
<tr>
<td>Rec(x:A) B</td>
<td>Recursion recursive types and type operators</td>
</tr>
<tr>
<td>Ref(A)</td>
<td>Reference types state, side effects</td>
</tr>
</tbody>
</table>
Kernel Syntax

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

\[ e ::= \]
\[ x \ | \]
\[ \text{Type} \ | \]
\[ \text{Power}(e) \ | \]
\[ \text{All}(x: e) \ e \ | \]
\[ \text{fun}(x: e) \ e \ | \]
\[ e(e) \ | \]
\[ \text{Some}(x: e) \ e \ | \]
\[ \text{pair}(x = e: e) \ e: e \ | \]
\[ \text{lft}(e) \ | \ \text{rht}(e) \ | \]
\[ \text{Record} \ t: e, \ldots, t: e \ \text{end} \ | \]
\[ \text{record} \ t = e, \ldots, t = e \ \text{end} \ | \]
\[ e . t \ | \]

\[ \text{Variant} \ t: e, \ldots, t: e \ \text{end} \ | \]
\[ \text{variant} \ t = .e : e.\text{end} \ | \]
\[ \text{case} \ e \ | \ t(x: e) \ e \ldots \ | \ t(x: e) \ e \ \text{end} \ | \]
\[ \text{Ref}(e) \ | \ \text{ref}(e) \ | \ \text{deref}(e) \ | \ e := e \ | \]
\[ \text{rec}(x: e) \ e \ | \]

\[ d ::= \text{let} \ x: e = e ; \]
**Simple values**

3+1;          the value 4

it;           the last value, i.e. 4

**Simple declarations**

\[
\begin{align*}
    \text{let } a &= 3; \\
    \text{let } a: \text{Int} &= 3; \\
    a; \\
    \text{let var } b &= 3; \\
    b := 5; \\
    b+3; \\
    \text{let } a &= 3 \\
    \text{and } b &= 5;
\end{align*}
\]

\[
\begin{align*}
    \text{declare } a \text{ to be a constant} \\
    \text{the same, with type information} \\
    \text{evaluate } a \\
    \text{declare } b \text{ to be a variable} \\
    \text{change } b \\
    = 8 \\
    \text{simultaneous declarations} \\
    \text{local declarations} \\
    \text{result is } 2n+1
\end{align*}
\]
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

\[
\text{rec } \text{fact}(n: \text{Int}): \text{Int} = \\
\text{if } n \text{ is } 0 \text{ then } 1 \\
\text{else } n \times \text{fact}(n-1) \text{ end;}
\]

Mutually recursive functions

\[
\text{rec } f(a: \text{Int}): \text{Int} = \\
\text{if } a \text{ is } 0 \text{ then } 0 \text{ else } g(n-1) \text{ end} \\
\text{and } g(b: \text{Int}): \text{Int} = \\
\text{if } b \text{ is } 0 \text{ then } 0 \text{ else } f(n-1) \text{ end;}
\]

Recursive values

\[
\text{rec } \text{self} = \\
\text{tuple} \\
\text{let } b = 3; \\
\text{let } f(n: \text{Int}): \text{Int} = n + \text{self}.b \\
\text{end;}
\]
Quest Syntax

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

- sequencing (strongest binding power)
  \[ ... \mid ... \]
- 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 \mid
component ide ; implements ide ;
  [ import spec(,) ; ] [ parts (ide = ide)(,) ; ]
  \{ phrase ; \} end \mid
phrase ;
```

**phrase ::=** phrase

```
def \mid
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
   \textbf{raise} \texttt{signal} \ [ \texttt{qual} \ ] \ |
   \textbf{try} \ \texttt{exp} \ \{ \texttt{on} \ \texttt{signal} \ \texttt{block} \} \ [ \texttt{finally} \ \texttt{block} \ ] \ \texttt{end} \ |

qualified
   \texttt{exp} \ . \ \texttt{ide} \ |
   \texttt{exp} \ . \ \texttt{ide} := \ \texttt{exp} \ |
   \texttt{exp} \ ^ \ \texttt{ide} \ |
   \texttt{exp} \ ^ \ \texttt{ide} := \ \texttt{exp} \ |
part 3

Polymorphism
Type declarations

The unit type
\[
\textbf{Let} \quad \text{Unit}:\text{Type} = \text{Tuple} \quad \textbf{end}; \\
\textbf{let} \quad \text{unit}:\text{Unit} = \text{tuple} \quad \textbf{end};
\]

An integer pair type
\[
\textbf{Let} \quad \text{IntPair}:\text{Type} = \\
\quad \text{Tuple} \quad \text{fst}:\text{Int}; \quad \text{snd}:\text{Int} \quad \textbf{end};
\]

A pair of that type
\[
\textbf{let} \quad p:\text{IntPair} = \\
\quad \text{tuple} \quad \text{fst}=3; \quad \text{snd}=4 \quad \textbf{end};
\]

The integer function type
\[
\textbf{Let} \quad \text{IntFun}:\text{Type} = \text{Int} \rightarrow \text{Int}; \\
\textbf{..a} \text{ function of that type} \\
\textbf{let} \quad f:\text{IntFun} = \text{succ};
\]
**Type operators**

**Cartesian product**

\[
\text{Let } \# \ (A:\text{Type})(B:\text{Type}):\text{Type} = \\
\text{Tuple \ fst:A; \ snd:B \ end;}
\]

**Function space**

\[
\text{Let } \rightarrow \ (A:\text{Type})(B:\text{Type}):\text{Type} = \\
\text{All}(x:A) \ B;
\]

**Homogeneous lists**

\[
\text{Rec } \text{List}(A:\text{Type}):\text{Type} = \\
\text{Variant} \\
\text{nil: Unit;} \\
\text{cons: Tuple \ hd:A; \ tl:List(A) \ end;} \\
\text{end;}
\]
**Polymorphic functions**

The type of the integer identity
\[
\text{Let } \text{IntId:Type} = \\
\quad \text{All}(x:\text{Int}) \text{ Int}; \\
\]

The integer identity
\[
\text{let intId:IntId} = \\
\quad \text{fun}(x:\text{Int}) \ x; \\
\]

Usage of integer identity
\[
\text{intId}(3); \\
\]

The type of the polymorphic identity
\[
\text{Let } \text{Id:Type} = \\
\quad \text{All}(A:\text{Type}) \text{ All}(a:A) \ A; \\
\]

The polymorphic identity
\[
\text{let id(A:Type)(a:A):A} = a; \\
\]

..application of a polymorphic function
\[
\text{id(Int)(3)}; \\
\]

..abbreviated application
\[
\text{id(3)}; \quad \text{where the missing Int parameter can be inferred} \\
\]

Specialized identities
\[
\text{let intId: Int->Int} = \text{id(Int)}; \\
\text{let boolId: Bool->Bool} = \text{id(Bool)}; \\
\]
Passing polymorphic functions

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

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

The polymorphic swap function

```plaintext
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.

$\vdash S \text{ sig}$

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

$\vdash_s E \text{ env}$

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

$E \vdash_s a : A$

Inferences

\[
\begin{array}{c}
J_1 \hspace{1cm} \ldots \hspace{1cm} J_n \hspace{1cm} (P) \\
\hline
J
\end{array}
\]
Signatures

Signature Construction

\[ \vdash \diamond \text{sig} \]

\[ \vdash S \text{ sig} \quad \vdash_{S} A: \text{Type} \quad c \notin \text{Dom}(S) \]

\[ \vdash S, c : A \text{ sig} \]

Environments

Environment Construction

\[ \vdash S \text{ sig} \]

\[ \vdash_{S} \diamond \text{env} \]

\[ \vdash_{S} E \text{ env} \quad \vdash_{S} A: \text{Type} \quad x \notin \text{Dom}(E) \]

\[ \vdash_{S} E, x : A \text{ env} \]
Constants

Given

\[
\begin{align*}
\Gamma_S E_{\text{env}} & \quad c : A \in S \\
E & \quad c : A
\end{align*}
\]

Variables

Assumption

\[
\begin{align*}
\Gamma_S E_{\text{env}} & \quad x : A \in E \\
E & \quad x : A
\end{align*}
\]

Type

Type Formation

\[
\begin{align*}
\Gamma_S E_{\text{env}} \\
E & \quad \text{Type} : \text{Type}
\end{align*}
\]
Functions

→ Formation

\[
\frac{E \vdash A : \text{Type} \quad E \vdash B : \text{Type}}{E \vdash A \rightarrow B : \text{Type}}
\]

→ Introduction

\[
\frac{E \vdash A : \text{Type} \quad E, x : A \vdash b : B \quad x \notin B}{E \vdash \text{fun}(x : A) \ b : A \rightarrow B}
\]

→ Elimination

\[
\frac{E \vdash a : A \quad E \vdash b : A \rightarrow B}{E \vdash b(a) : B}
\]
**Dependent functions**

**All Formation**

\[
E \vdash \ A \colon \text{Type} \quad E, \ x \colon A \vdash B \colon \text{Type} \\
\hline
E \vdash \ \text{All}(x \colon A) \ B : \text{Type}
\]

**All Introduction**

\[
E \vdash \ A \colon \text{Type} \quad E, \ x \colon A \vdash b \colon B \\
\hline
E \vdash \ \text{fun}(x \colon A) \ b : \text{All}(x \colon A) \ B
\]

**All Elimination**

\[
E \vdash a : A \quad E \vdash b : \text{All}(x \colon A) \ B \\
\hline
E \vdash b(a) : B\{x \leftarrow a\}
\]
Recursion

Rec Formation

\[ E \vdash S \text{ Type} \quad E, x : A \vdash_S a : A \]
\[ \frac{}{E \vdash_S \text{ rec}(x : A) a : A} \]

Type computations

Conversion

\[ E \vdash_S a : A \quad E \vdash_S B : \text{ Type} \quad A = _{\beta\eta\mu} B \]
\[ \frac{}{E \vdash_S a : B} \]
Examples

\[
\begin{align*}
E, A: \text{Type}, & \ x:A \vdash_S x:A \\
E, A: \text{Type} & \vdash_S \text{fun}(x:A) \ x : \text{All}(x:A) \ A \\
E & \vdash_S \text{fun}(A: \text{Type}) \ \text{fun}(x:A) \ x : \text{All}(A: \text{Type}) \ \text{All}(x:A) \ A
\end{align*}
\]

\[
\text{let id be } \text{fun}(A: \text{Type}) \ \text{fun}(x:A) \ x
\]

\[
\begin{align*}
E & \vdash_S \text{Int: Type} \quad E & \vdash_S \text{id: All}(A: \text{Type}) \ \text{All}(x:A) \ A \\
E & \vdash_S 3: \text{Int} \quad E & \vdash_S \text{id}(\text{Int}) : \text{All}(x: \text{Int}) \ \text{Int} \\
E & \vdash_S \text{id}(\text{Int})(3) : \text{Int}
\end{align*}
\]
part 4

Abstract Types
Tuples

A triple and one of its types

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

A labeled pair and two of its labeled types

```plaintext
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

```plaintext
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

```plaintext
tuple a: Int=3; b: Bool=true end;
```
**Tuple declarations**

**Binding a pair**

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

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

**Splitting a pair**

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

x = 3
y = true
```

**Matching a pair**

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

x = 3
y = true
```

**Tuple selection**

**Selecting a field**

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

p.a;

= 3
```

... abbreviation of

```plaintext
begin let (x; y) = p; x end;
```
Tuples and functions

A function expecting a pair

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

Two legal applications

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

A tuple with function components

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

Selection and application

```plaintext
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 \( \alpha \)-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 \texttt{alg1} and \texttt{alg1'} are both implementations of \texttt{Alg} based on \texttt{Int}, then \texttt{alg1.C} matches \texttt{alg1'.C}, and they both match \texttt{Int}. There is no \textit{information hiding} when the packages are \textit{manifest}.

\begin{verbatim}
alg1.obj + 1; \textbf{Legal!}  
alg.C = Int

alg1.op(alg1'.obj); \textbf{Legal!}  
alg1.C = alg1'.C
\end{verbatim}

\textbf{Non-generative} abstract types (matched by structure) have some advantages over \textbf{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 \( \lambda \)-abstraction. Privacy of representation is automatically enforced when writing a function wishing to use any implementation of a given abstract type.

```plaintext
let fun (alg:Alg):Int =
    alg.obj + 1;            \text{Illegal!}
    alg.C \neq \text{Int}

let fun (alg1,alg2:Alg):Int =
    alg1.op(alg2.obj);     \text{Illegal!}
    alg1.C \neq alg2.C

let fun (alg:Alg; a,b:alg.C):Int =
    alg.op(a)+alg.op(b);   \text{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

× Formation

\[
E \vdash_s A : \text{Type} \quad E \vdash_s B : \text{Type} \\
E \vdash_s A \times B : \text{Type}
\]

× Introduction

\[
E \vdash_s a : A \quad E \vdash_s b : B \\
E \vdash_s <a, b> : A \times B
\]

× Elimination

\[
E \vdash_s c : A \times B \\
E \vdash_s \text{lft}(c) : A \\
E \vdash_s \text{rht}(c) : B
\]

\[
E \vdash_s c : A \times B \\
E \vdash_s \text{lft}(c) : A \\
E \vdash_s \text{rht}(c) : B
\]
**Dependent pairs**

**Some Formation**

\[
\frac{E \vdash_S A: \text{Type} \quad E, x:A \vdash_S B: \text{Type}}{E \vdash_S \text{Some}(x:A) B : \text{Type}}
\]

**Some Introduction**

\[
\frac{E \vdash_S a:A \quad E \vdash_S b[x\leftarrow a]: B[x\leftarrow a]}{E \vdash_S \text{pair}(x:A=a) b:B : \text{Some}(x:A) B}
\]

**Some Elimination**

\[
\frac{E \vdash_S \text{c: Some}(x:A) B \quad E \vdash_S \text{c: Some}(x:A) B}{E \vdash_S \text{lft}(c): A \quad E \vdash_S \text{rht}(c): B[x\leftarrow \text{lft}(c)]}
\]
Examples

$$E \vdash_S 0 : \text{Int} \quad E \vdash_S \text{succ} : \text{Int} \rightarrow \text{Int}$$

$$E \vdash_S \text{Int} : \text{Type} \quad E \vdash_S <0, \text{succ}> : \text{Int} \times (\text{Int} \rightarrow \text{Int})$$

$$E \vdash_S \text{pair}(A : \text{Type} = \text{Int}) <0, \text{succ}> : A \times (A \rightarrow \text{Int})$$

$$: \text{Some}(A : \text{Type}) \ A \times (A \rightarrow \text{Int})$$

$$let \ pack \ be \ \text{pair}(A : \text{Type} = \text{Int}) <0, \text{succ}> : A \times (A \rightarrow \text{Int})$$

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

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

$$let \ Abs \ be \ \text{lft}(\text{pack}) \ and \ op \ be \ \text{rht}(\text{pack})$$

$$E \vdash_S \text{op} : \text{Abs} \times (\text{Abs} \rightarrow \text{Int}) \quad E \vdash_S \text{op} : \text{Abs} \times (\text{Abs} \rightarrow \text{Int})$$

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

$$E \vdash_S \text{rht}(\text{op})(\text{lft}(\text{op})) : \text{Int}$$
part 5

Subtypes
Multiple inheritance

A multiple inheritance hierarchy

\begin{align*}
\text{Let } & \text{ Object:Type } = \\
& \text{ Record } \text{ var age:Int end; } \\
\text{Let } & \text{ Vehicle:Type } = \\
& \text{ Object And } \\
& \text{ Record speed:Int end; } \\
\text{Let } & \text{ Machine:Type } = \\
& \text{ Object And } \\
& \text{ Record fuel:String end; } \\
\text{Let } & \text{ Car:Type } = \\
& \text{ Vehicle And Machine; } \\
\end{align*}
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

```plaintext
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")

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

`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

\[ C : \text{Power}(A) \quad \text{means} \quad C \subseteq A \]

(written \[ C <: A \])

\[ \text{All}(x <: A) \ B \quad \text{(i.e. All}(x : \text{Power}(A)) \ B) \]

\[ \text{fun}(x <: A) \ B \]

\[ \text{Tuple} \ldots ; \ x <: A ; \ldots \ \text{end} \]

\[ \text{tuple} \ldots ; \ x <: A = a ; \ldots \ \text{end} \]
**Higher-order inheritance**

```plaintext
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: \textbf{All} (A <: Int) \textbf{All} (B : Type) Type

array: \textbf{All} (A <: Int) \textbf{All} (B : Type)
\hspace{1cm} \textbf{All} (b : B) \textbf{Array} (A) (B)

array(1..6) (Bool) (false):
\hspace{1cm} \textbf{Array}(1..6) (Bool)
part 5A

Type Rules
Record Formation

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

Record Introduction

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

Record Elimination

\[ E \vdash_S \text{r : Record } t_1 : A_1 \ldots t_n : A_n \text{ end} \]
\[ E \vdash_S \text{r.}_{i} : A_i \]
Power Formation

\[ E \vdash_S A : \text{Type} \]
\[ E \vdash_S \text{Power}(A) : \text{Type} \]

Power Introduction

\[ E \vdash_S A : \text{Type} \]
\[ E \vdash_S A \leq A \]

Power Elimination

\[ E \vdash_S a : A \quad E \vdash_S A \leq B \]
\[ E \vdash_S a : B \]
Power Record

\[ E \vdash_S A_1 \leq B_1 \quad \ldots \quad E \vdash_S A_n \leq B_n \quad \ldots \quad E \vdash_S A_m \text{: Type} \]

\[ E \vdash_S \text{Record } t_1:A_1 \ldots t_n:A_n \ldots t_m:A_m \text{ end} \leq \text{Record } t_1:B_1 \ldots t_n:B_n \text{ end} \]

Power Variant

\[ E \vdash_S A_1 \leq B_1 \quad \ldots \quad E \vdash_S A_n \leq B_n \quad \ldots \quad E \vdash_S B_m \text{: Type} \]

\[ E \vdash_S \text{Variant } t_1:A_1 \ldots t_n:A_n \text{ end} \leq \text{Variant } t_1:B_1 \ldots t_n:B_n \ldots t_m:B_m \text{ end} \]
Power →

\[ E \vdash_S A' \leq A \quad E \vdash_S B \leq B' \]
\[ \frac{}{E \vdash_S A \rightarrow B \leq A' \rightarrow B'} \]

Power All

\[ E \vdash_S A' \leq A \quad E, x:A' \vdash_S B \leq B' \]
\[ E \vdash_S \text{All}(x:A) B \leq \text{All}(x:A') B' \]

Power \times

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

Power Some

\[ E \vdash_S A \leq A' \quad E, x:A \vdash_S B \leq B' \]
\[ \frac{}{E \vdash_S \text{Some}(x:A) B \leq \text{Some}(x:A') B'} \]
Power Type

\[
E \vdash_S A : \text{Type} \\
\hline \\
E \vdash_S \text{Power}(A) \leq \text{Type}
\]

Power Power

\[
E \vdash_S A \leq B \\
\hline \\
E \vdash_S \text{Power}(A) \leq \text{Power}(B)
\]
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:

```python
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:

\[
\text{ExtPoint} <: \text{Point};
\]

and we define the parametric circle interface as:

\[
\text{Circle: All}(P <: \text{Point}) \text{ Type};
\]

Then, we can form the interface:

\[
\text{Circle(ExtPoint)}
\]

(although we do not have inclusions between \text{Circle(Point)} and \text{Circle(ExtPoint)} because the \text{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 constraints.
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 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 caching).
  Stack retention analysis.
  Producing bytecode (initially).

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

Run-Time
  RCMaps (heap and stack).
  Pickling.
  GC (Compacting).