IFIP State of the Art Seminar on
Formal Description of Programming Concepts

Typeful Programming

Luca Cardelli

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

Outline

1. Introduction: typeful programming

   Typed λ-calculi and universe levels

   The Quest language; fundamentals

2. The Quest language; constructions

   Formal systems. Ex.: first-order typed λ-calculus

3. Quest core formal system

   Quest typing

   Quest subtyping

   Other Quest design topics

4. Modules and interfaces

   Module combination

   System programming

   Conclusions

Talk: Operations on records
Talk: Introduction to constructive type theory

Introduction

Typing trends

Languages for describing large and well-structured software systems have been evolving towards stronger and stronger notions of typing.

(Non-chronological)

Imperative: Fortran -> Algol60 ->Pascal ->Modula2

System: Assm -> C -> Mesa, C++

Object-or.: Smalltalk -> Simula67 -> Trellis/Owl, Modula3

Functional: Lisp -> Scheme -> CLU,ML,Id,Miranda

Concurrent: (semaphores/monitors) -> NIL, FX

Logical: Prolog -> ?

There have been also serious attempts to typecheck Smalltalk, Lisp/Scheme, and Prolog.

Typeful programming

Diverse language paradigms, imperative, functional , concurrent, object-oriented, etc. have been converging towards a common programming style; *typeful programming*.

This style is characterized by the widespread use of *type information* intended as a *partial specification* of a program. But:

Typing constraints should be (largely) *decidably verifiable;* the purpose of type constraints is not simply to state programmer intentions, but to actively trap programmer errors. (Arbitrary formal specifications do not have this property).

Typing constraints should be *transparent*: a programmer should be able to easily predict when a program will typecheck, and if it fails to typecheck the reason should be apparent. (Automatic theorem proving does not have this property, at least not yet).

Typing constraints should be *enforceable*: statically checked as much as possible, otherwise dynamically checked. (Program comments and conventions do not have this properties).

Hence we require *strong typing*, intended as any combination of static and dynamic typing that prevents *unchecked run time type errors* (the notion of *type error* has to be defined for each particular language).

It would be nice to require *static typing*, but this is impossible in practice, e.g. because of persistent data, bootstrapping, and embedded eval primitives. Dynamic typechecking can be confined so that it does not cause too many problems.

An old tenet claims that many diverse forms of computation can be understood in terms of the untyped
λ-calculus (or similarly minimal systems, e.g. CCS).

The new tenet is that many diverse forms of program typing can be understood in terms of suitable typed λ-calculi.

The purpose of these lectures is to show how existing and novel features of typeful languages can be expressed in a single type system. This will provide insights in how to extend or simplify these features.

This single system is in a sense rather powerful and sophisticated, but in another sense is also fundamentally simple: it is obtained by variations over a small and well-studied kernel.

Relevant concepts

Higher-order functions	(ISWIM)
Abstract types	(CLU)
Polymorphism	(ML)
Subtyping	(Simula67)
Modules and Interfaces	(Mesa)

Theory and practice

The conceptual framework for typeful programming is derived from various theories of typed λ-calculi, collectively called *type theory*.

In particular, we base our discussion on Girard's system $F\omega$ (also called the *higher-order λ-calculus*), extended with a notion of subtyping.

There is a strong correspondence between constructs in type theory and constructs in programming (this is because type theory is a form of constructive logic).
Constructs studied in type theory have provided new insights on constructs already developed in programming (e.g. for abstract types and polymorphism). Vice versa, some programming constructs pose interesting type theory questions.

This kind of theoretical understanding is very useful to understand, simplify and generalize programming constructs.
However we should not oversimplify: a programming language is not just a formal system. It has very different requirements and constraints in terms of compiler engineering, user "aesthetics", and programming "sociology".

Here is a list of situations where nice theory tends to ignore or conflict with programming reality.

Notation
- Conciseness vs. redundancy.
- Languages should be designed for readability

Scale
- Small cute example vs huge real programs.
- Languages should be designed for large programs.

Typechecking
- Should be decidable, efficient, and easily understood.

Translation
- Should be "linear" and efficient.

Efficiency
- Translated code should executed efficiently and not require complex optimizations.

Generality
- A language should be usable for building many different kinds of systems.
- (1) Theoretically complete (Turing-complete).
- (2) Practically complete:
  should be able to express, in order of ambition:
  - its own interpreter
  - its own translator
  - its own run-time system
  - its own operating system

Why types? (a methodological view)

To aid in the *evolution* of software systems.

Large software systems are not *created,* they evolve.

Evolving software systems are (unfortunately):

not correct
(people keep finding and fixing bugs)

or else, *not good enough*
(people keep improving on space and time requirements)

or else, *not clean enough*
(people keep restructuring for future evolution)

or else, *not functional enough*
(people keep adding and integrating new features)

Some form of "software hygiene" is necessary.

Reliability:

Naively, software either works or it does not, but evolving software is always sort of in-between.

> Working hardware is *reliable* if it does not break too often, in spite of wear.
>
> Evolving software is *reliable* if it does not break too often, in spite of change.

Type systems provide a way of controlling change, inspire some degree of confidence after each evolutionary step, and help in producing and maintaining *reliable* software systems.

<u>Why subtypes? (a methodological view)</u>

To aid in the *<u>extension</u>* of software systems.

A software system provides some service. To extend the service one can modify the system, but this is inconvenient and *unreliable*.

To increase reliability, there should be ways of extending a system from the *outside,* by adding to it without modifying it directly.

Subtyping is one such mechanism. The types handled by the basic system can be extended by the derived system. The extended types are still recognized by the basic system.

The extended system will be more reliable (with respect to changes to the basic system) if some *abstraction* of the basic system has been used in building the extended system.

Why polymorphism?

Type systems constrain the underlying untyped language.

Good type systems, while providing static checking, do not impose excessive constraints.

A simple untyped language

Terms: (x variables; k constants; l labels; a,b,c terms)

construct	*introduction*	*elimination (may fail)*
variable	x	
constant	k	
function	**fun**(x) b	b(a)
tuple	**tuple** l_1=a_1, ... , l_n=a_n **end**	b.l

Flexible, but computation may *fail*.

Failure points reduce software reliability.

To prevent *failure*, organize terms into a *type system*.

Try to preserve the flexibility of the untyped calculus as much as possible through *polymorphism*:

(**fun**(x) **tuple** fst=x snd=x **end**) (3)
(**fun**(x) **tuple** fst=x snd=x **end**) (true)

preserved by *parametric polymorphism.*

(**fun**(x) x.t) (**tuple** t=3 **end**)
(**fun**(x) x.t) (**tuple** t=3, u=true **end**)

preserved by *subtype polymorphism.*

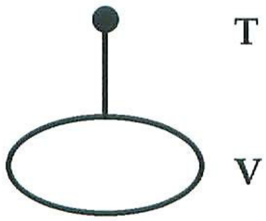
Typed λ-calculi and universe levels

Notation(s)

	Programming	*Logic*	*Semantics*
Functions	fun(x:A) x f(a)	$\lambda x{:}A.\ x$ (f a)	$\lambda x{\in}A.\ x$ f(a)
Polymorphic Func.	fun(A) fun(x:A) x fun(A::TYPE) b f[A] f(:A)	$\Lambda A.\ \lambda x{:}A.\ x$ $\Lambda A{:}Type.\ b$ f[A]	$\lambda A{\in}Type.\ b$ f(A)
Tuples	3,'x' tuple a=3, b='x' end let (x,y) = t t.a	$\langle 3,'x' \rangle$ fst(t) snd(t)	$\langle 3,'x' \rangle$ let $\langle x,y \rangle = t$ $\pi_i(t)$
Function Spaces	A->B All(x:A)B	$A \rightarrow B$ $A \supset B$	$A \rightarrow B$ $A \rightharpoonup B$
Cartesian Prod.	A#B Tuple f:A, s:B end	$A \times B$ $A \wedge B$	$A \times B$
Disjoint Union	A+B Variant a:A, b:B end	A+B $A \vee B$	A+B
Universal Quant.	All[A] A->A All (A) A->A	$\forall A.\ A \rightarrow A$	$\Pi A.\ A \rightarrow A$
Existential Quant.	Some[A] A#(A->B) Tuple A, f:A->B end	$\exists A.\ A \times (A \rightarrow B)$	$\Sigma A.\ A \times (A \rightarrow B)$

Levels
(values, types and operators, kinds)

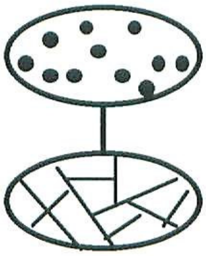
One type
(untyped λ-calculus)



let id = **fun**(x) x

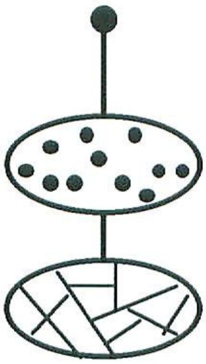
id(3)

Many types
(first-order typed λ-calculus)




```
let id : Int->Int =
   fun(x:Int) x

id(3)
```

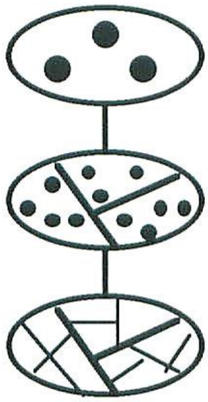
One kind
(second-order typed λ-calculus)
(second-order polymorphic λ-calculus)



let id : **All**(A) A->A =
  fun(A) **fun**(x:A) x

id(:Int)(3)

Many kinds
(higher-order typed λ-calculus)

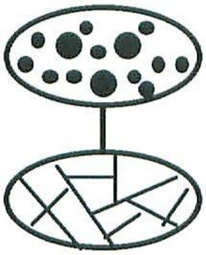


Let Endo:: TYPE=>TYPE =
  All(A::TYPE) A->A

let id : **All**(A::TYPE) Endo(A) =
  fun(A::TYPE) **fun**(x:A) x

id(:Int)(3)

Kinds are types
(higher-order typed λ-calculus with Type:Type)



let Endo: Type->Type =
  fun(A:Type) A→A

let id : **All**(A:Type) Endo(A) =
  fun(A:Type) **fun**(x:A) x

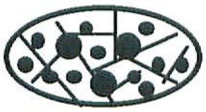
id(:Int)(3)

Problem: *static levelchecking* becomes impossible.

fun(A:Type) **fun**(x:A) x   is A a type or a kind?
is x a type or a value?

id (:Int) (3)   id (:Type) (:Int)
comp ↑ *run*   *comp* ↑ *run*

Types are values
(untyped λ-calculus with Type:Value)



K,T,V

let Endo = **fun**(A) A->A

let id = **fun**(A) **fun**(x) x

id(Int)(3)

Problem: *static typechecking* becomes impossible.

id (Int) (true)   id (3) (4)

Universes

The predicative hierarchy

Values	3	$\in$ **U0**
Types	Int, Int->Int, (Int->Int)->Int	$\in$ **U1**
Polytypes	All(A::U1)A->A	
Operators	U1=>U1	$\in$ **U2**
Higher operators	U2=>U2	$\in$ **U3**
		...

Problem: polymorphic functions and elements of abstract types are not values.

The impredicative hierarchy

Values	3	$\in$ **Value**
Types	Int, Int->Int, (Int->Int)->Int	
Polytypes	All(A::TYPE)A->A	$\in$ **Type**
Operators	Type=>Type	
Higher operators	(Type=>Type)=>Type	$\in$ **Kind**

Problem: model construction is delicate (but possible).

We will work in the impredicative hierarchy.

The Quest Language Fundamentals

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 on 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).

Why a new language?

There is now a general understanding of:

- Explicit polymorphism as:
  (predicative or impredicative) general products

- Abstract types as:
  (impredicative) general sums

- Interfaces as:
  (predicative or impredicative) general sums

- Inheritance (partially) as:
  subtyping

These language features where developed independently of these explanations. Very few attempts have been made to feed back the explanations into language features (main exceptions: SML's modules, Pebble's dependent types, Amber's subtyping).

Language overview

An exploration of type QUantifiers & SubTypes.

Based on Fω [Girard] plus subtyping;
impredicative value/type/kind structure.

Strongly typed. Explicit quantification for polymorphism and abstract types.

Modules and interfaces; modules are first-class values.

Structural typing and subtyping (type matching is determined by type structure, not by the way types are declared or named).
This includes structural matching of abstract types.

User-definable type operators and computations at the type level. Typechecking involves λ-reduction.

Generalized correspondence principle (uniformity of declarations/formal-parameters/interfaces and definitions/actual-parameters/modules).

Expression-based, functional style but with imperative features. Call-by-value evaluation.

Interactive but compiled

Addressing pragmatic questions such as:

Notation
(To make software easily readable and writable.)

Scale
(Care about the organization of large programs.
Promote reusability and extensibility.)

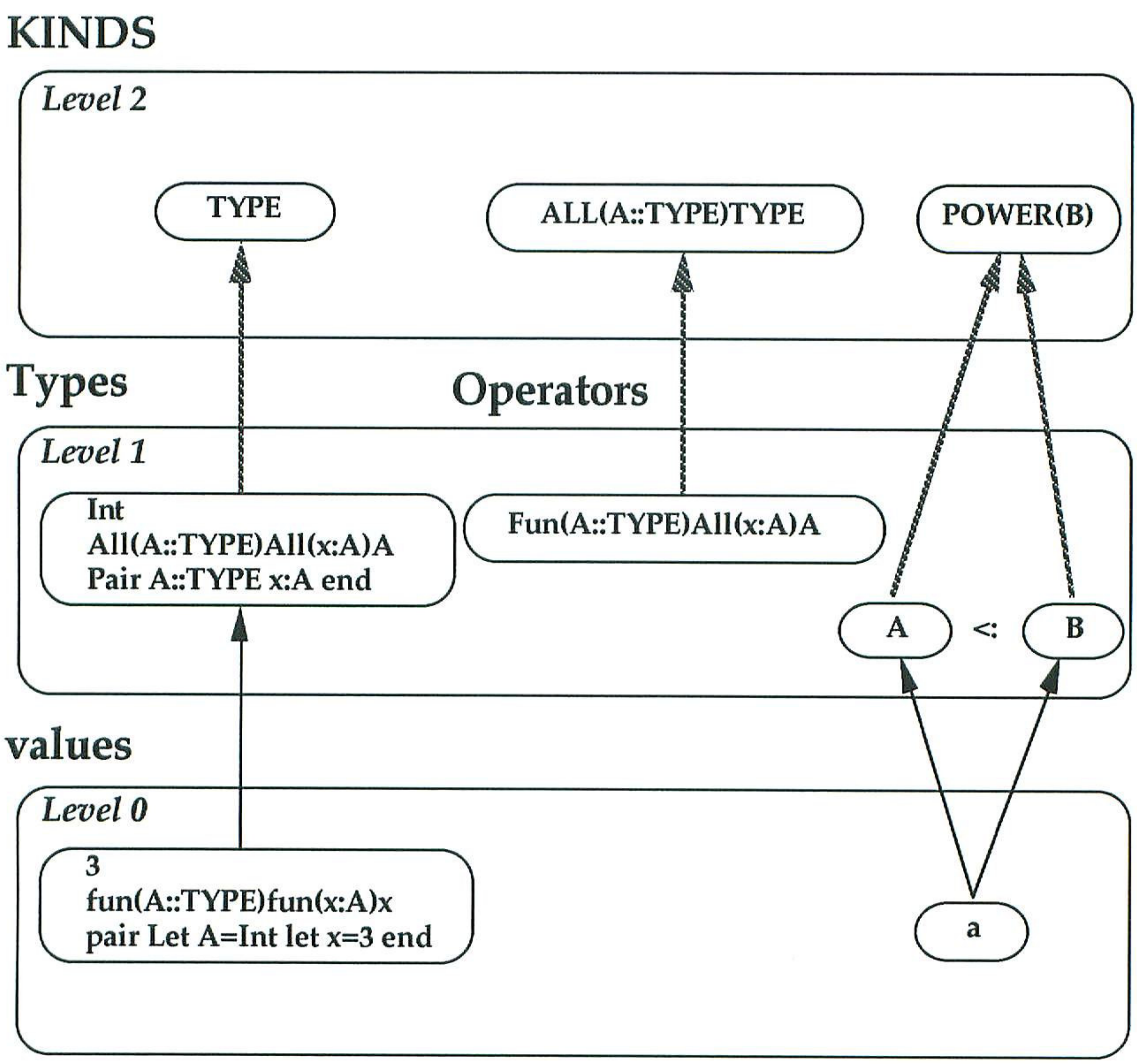
Typechecking
(Decidability, heuristics, efficiency.)

Translation
(Make it possible.)

Efficiency
(Of generated code.)

Generality
(A "real" language, as opposed to a special-purpose or application-oriented language. (a) Turing complete, and (b) able to express its own compiler conveniently.)

Language levels



- **All**(x:A)B (e.g: **fun**(x:Int)x)

  This is the type of functions from values in A to values in B, where A and B are types. The variable x can appear in B only in special circumstances, so this is normally equivalent to the function space A->B. Sample element:

- **All**(X::K)B (e.g: **fun**(A::TYPE)**fun**(x:A)x)

  This is the type of functions from types in K to values in B, where K is a kind, B is a type, and X may occur in B. Sample element:

- **ALL**(X::K)L (e.g: **Fun**(A::TYPE)A)

  This is the kind of functions from types in K to types in L, where K and L are kinds, and X may occur in L.

- **Pair** x:A y:B **end** (e.g: **pair let** x=true **let** y=3 **end**)

  This is the type of pairs of values in A and values in B, where A and B are types. The variable x can appear in B only in special circumstances, so this is normally equivalent to the cartesian product A#B.

- **Pair** X::K y:B **end** (e.g: **pair Let** X=Int **let** y:X=3 **end**)

  This is the type of pairs of types in K and values in B, where K is a kind, B is a type, and X may occur in B.

Binary vs. n-ary quantifiers

g: **All**(A::TYPE) **All**(a:A) **All**(f:A->Int) Int

= **fun**(A::TYPE) **fun**(a:A) **fun**(f: A->Int) f(a)

g(:Int)(3)(succ)

vs.

g: **All**(A::TYPE a:A f:A->Int) Int

= **fun**(A::TYPE a:A f:A->Int) f(a)

g(:Int 3 succ)

t: **Pair**(A::TYPE) **Pair**(a:A) A->Int

= **pair**(A::TYPE=Int) **pair**(a:A=3) succ **end** **end**

i.e.: t = **pair** :Int **pair** 3 succ **end** **end**

snd(**snd**(t))(**fst**(**snd**(t)) : Int

fst(**snd**(t)) : **Fst**(t)

vs.

t: **Tuple** A::TYPE a:A f:A->Int **end**

= **tuple**
Let A::TYPE =Int **let** a:A = 3 **let** f:A->Int = succ
end

i.e.: t = **tuple** :Int 3 succ **end**

t.f(t.a) : Int

t.a : t.A

Introduce the notion of a *signature* S,
e.g.:

A::TYPE a:A f:A->Int

n-ary universals: **All**(S) A

n-ary existentials: **Tuple** S **end**

Correspondingly, introduce the notion of a *binding* D $\therefore$ S
e.g.:

Let A::TYPE =Int **let** a:A = 3 **let** f:A→Int = succ

n-ary application f(D)

n-ary products **tuple** D **end**

Generalized correspondence principle

(Burstall-Lampson)

let $f(S_1)..(S_n){:}B = b \quad\cong\quad$ **let** $f =$ **fun**(S_1) .. **fun**$(S_n){:}B\ b$

$f(S_1)..(S_n){:}B \quad\cong\quad f{:}$ **All**(S_1) .. **All**$(S_n)B$

Signatures in:

Declarations

Let **A::TYPE** = Int let **a:A** = 3 let **f(x:A):Int** = x+1

Formal parameters

let f(**A::TYPE** **a:A** **f(x:A):Int**):A = ...

Types

All(**A::TYPE** **a:A** **f(x:A):Int**) A
Tuple **A::TYPE** **a:A** **f(x:A):Int** end

Interfaces

interface I import ...
export
A::TYPE
a:A
f(x:A):Int
end

Bindings in:

Definitions (e.g. at the top-level)

<u>**Let A::TYPE = Int**</u>   <u>**let a:A = 3**</u>   <u>**let f(x:A):Int = x+1**</u>
<u>**:Int**</u>   <u>**3**</u>   <u>**fun(x:Int)x+1**</u>

Actual parameters

f(<u>**Let A::TYPE = Int**</u>   <u>**let a:A = 3**</u>   <u>**let f(x:A):Int = x+1**</u>)
f(<u>**:Int**</u>   <u>**3**</u>   <u>**fun(x:Int)x+1**</u>)

Tuples

tuple
    <u>**Let A::TYPE = Int**</u>   <u>**let a:A = 3**</u>   <u>**let f(x:A):Int = x+1**</u>
end
tuple   <u>**:Int**</u>   <u>**3**</u>   <u>**fun(x:Int)x+1**</u>   end

Modules

module m:I import ...
export
    <u>**Let A::TYPE = Int**</u>
    <u>**let a:A = 3**</u>
    <u>**let f(x:A):Int = x+1**</u>
end

The Quest language Constructions

Simple values

`"abc";`	*the string "abc"*
`3+1;`	*the value 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
and b = 5;
```
simultaneous declarations

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

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

Recursive declarations

Recursive functions

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

Mutually recursive functions

```
let 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

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

Tuples

A triple and its type

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

A labeled pair and its type

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

A dependent pair and its type

```
tuple Let A:Type=Int let b:A=3 end;
   : Tuple A:Type b:A end
```

A labeled pair with type info

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

Selecting a field

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

Tuples and functions

A function expecting a pair

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

A legal application

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

A tuple with function components

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

Selection and application

```
q.succ(3);
```

Type declarations

The Ok type

```
Let Ok::TYPE = Tuple end;
let ok:Ok = tuple end;
```

An integer pair type

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

A pair of that type

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

The integer function type

```
Let IntFun::TYPE = All(:Int)Int;
```

..a function of that type

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

Type operators

Cartesian product

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

Function space

```
Let ->(A,B::TYPE)::TYPE =
  All(:A) B;
```

Ex.: `(Int # Int) -> Int :: TYPE`

Homogeneous lists

```
Let List(A::TYPE)::TYPE =
  Rec (B::TYPE)
    Option
      nil
      cons with hd:A tl:B end
    end;
```

Ex.: `List(Int) :: TYPE`

Polymorphic functions

The type of the integer identity

```
Let IntId::TYPE =
  Int -> Int;
```

The integer identity

```
let intId(a:Int):Int =
  a
```

Usage of integer identity

```
intId(3);
```

The type of the polymorphic identity

```
  Let Id::TYPE =
    All(A::TYPE) 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,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

```
let hd = exception "hd" end
and tl = exception "tl" end;

Let List(A::TYPE)::TYPE =
  Rec (B::TYPE)
    Option
      nil
      cons with hd:A tl:B end
    end;

let nil(A::TYPE):List(A) =
  option nil of List(A) end;

let cons(A::TYPE hd:A tl:List(A))
    :List(A) =
  option
    cons of List(A) with hd tl
  end;

let null(A::TYPE a:List(A)):Bool =
  case a
    when nil then true
    when cons then false
  end;

cons(:Int 3 nil(:Int))

cons(3 nil())
```

```
let head(A::TYPE)(a:List(A)):A =
  case a
    when nil then raise hd end
    when cons with p then p.hd
  end;

let tail(A::TYPE)(a:List(A)):List(A) =
  case a
    when nil then raise tl end
    when cons with p then p.tl
  end;

let rec length(A::TYPE)(a:List(A)):Int =
  case a
    when nil then 0
    when cons with p then 1+length(p.tl)
  end;

let rec map(A,B:TYPE)
      (f:A->B)(a:List(A)):List(B) =
  case a
    when nil then nil(:B)
    when cons with p then
      cons(:B)(f(p.hd) map(A B)(f)(p.tl))
  end;
```

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 f(alg:Alg):Int =
  alg.op(alg.obj);

f(alg1);
f(alg2);
```

Information hiding

Representation types are not revealed "outside":

```
:alg1.T;
  :alg1.T :: TYPE


alg1.obj;
  = <value> : alg1.T

alg1.obj + 1;
  error
```

Different implementations cannot be mixed:

```
alg2.op(alg1.obj);
  error
```

But values of the "same" implementation can be mixed:

```
alg1.op(alg1.obj);
  = 1 : Int
```

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
    T::TYPE obj:T op:T->Int
  end;
```

`Alg'` matches `Alg` 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.

This problem can be fixed by *branding,* i.e. associating a unique identifier with a type. Branding is said to be *generative,* because two elaborations of the same type expression generate two different types.

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.

Parametric abstract types

```
Let StackPackage(Item::Type)::Type =
  Tuple
    Stack(A::TYPE)::TYPE
    empty:Stack(Item)
    isEmpty(s:Stack(Item)):Bool
    push(s:Stack(Item) i:Item):Stack(Item)
    top(s:Stack(Item)):Item
    pop(s:Stack(Item)):Stack(Item)
  end;

let stackFromList(Item::TYPE)
      :StackPackage(Item) =
  tuple
    Let Stack = List
    let empty = nil(:Item)
    let isEmpty = null(:Item)
    let push = cons(:Item)
    let top = head(:Item)
    let pop = 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);
```

Other Quest design topics

Quantifier closures

(A,B: types; K,L: kinds)

Π(X::K)L{X} is a kind (functions from types to types)

Kinds are closed under quantification over kinds.

Π(x:A)L{x} is a kind (functions from values to types)

Kinds are closed under quantification over types.

Π(X::K)B{X} is a type (functions from types to values)

Types are closed under quantification over kinds.

Π(x:A)B{x} is a type (functions from values to values)

Types are closed under quantification over types.

(Similarly for Σ.)

Quantifiers: pick your own

(A,B: types; K,L: kinds)

Quest	**ELF**	**Fω**	**Constructions**	
ALL(X::K) L	-	$K \Rightarrow L$	Π(X::K) L	a kind
-	Π(x:A) L	-	Π(x:A) L	a kind
All(X::K) B	-	$\forall$(X::K) B	Π(X::K) B	a type
$A \rightarrow B$	Π(x:A) B	$A \rightarrow B$	Π(x:A) B	a type

derivable:

- (could add)	-	-	Σ(X::K) L	a kind
-	-	-	Σ(x:A) L	a kind
Pair(X::K) B	-	-	Σ(X::K) B	a type
$A \times B$	-	-	Σ(x:A) B	a type

Phase distinctions

(a,b: values; A,B: types; K,L: kinds)

Compilers are organized in two *phases*:

1st phase (*compile-time*)	$E \vdash a{:}A$	typechecking
2nd phase (*run-time*)	$E \vdash a = b : A$	evaluation

The phases are distinct: value-equality judgements should not be used in the derivation of typing judgements:

1st $\quad E \vdash a{:}A \quad$ (a,A are symbolic expressions)

$\Downarrow$ translation

2nd $\quad E \vdash a = b : A \quad$ (a,b are bunches of bits)

Since strings of bits cannot be symbolically analyzed (or it is very hard to do so), we loose the ability to substitute equals for equals at run-time.

(Also, in practical language value computations may be *impure*.)

Strings of bits should not appear in symbolic expressions. I.e., run-time expressions should not appear within compile-time expressions.

Phase distinctions can be related to level distinctions:

Kinds and types are elaborated in the *compile-time* phase.

Values are elaborated in the *run-time* phase.

Levels separate phases in, e.g., second-order lambda calculus:

$(^{\underline{\text{fun(A::TYPE)}}}\ \text{fun}(\text{a}{:}^{\underline{\text{A}}})\ \text{a})\ ^{\underline{\text{(Int)}}}\ (3)$ $\qquad\qquad$ $(^{\underline{\text{1st}}}\ \text{2nd})$

But levels do not separate phases in full dependent types (there should be no subscripts nested inside superscripts):

$\text{last} = \text{fun}(\text{n}{:}^{\underline{\text{Int}}})\ \text{fun}(\text{a}{:}^{\underline{\text{Array}(_{\text{n}})}})\ \text{a}[\text{n-1}]$

Run-time should be distinct from compile-time:

(1) Weaker constraint: value-equality judgements should not be used in the derivation of type or kind judgements.

(2) Stronger constraint: forbid all substitutions of the form $B\{x \leftarrow a\}$ or $L\{x \leftarrow a\}$

Adopt (2); remove all the quantifiers that are made useless:

$\Pi(x{:}A)B$	has $B\{x \leftarrow a\}$ in the elim rule (keep $A \rightarrow B$).
$\Pi(x{:}A)L$	has $L\{x \leftarrow a\}$ in the elim rule.
	($A \rightarrow L$ has $B\{x \leftarrow a\}$ in reduction rule.)
$\Sigma(x{:}A)B$	has $B\{x \leftarrow lft(a)\}$ in the elim rule (keep $A \times B$).
$\Sigma(x{:}A)L$	has $L\{x \leftarrow lft(a)\}$ in the elim rule.
	(could keep $A \times L$, but it is isom. to $L \times A$.)

Hence, we can define the syntax so that value-expressions do not occur in type or kind expressions.

Note: constant expressions are evaluated at compile-time in standard compilers (e.g. in Array(n-1)). But these evaluations never involve real run-time values.

Subtyping

A <: B is the natural relation of subtyping, e.g. Nat<:Int (reflexive and transitive)

all objects of A are objects of B
or, objects of A have all the properties of objects of B (inheritance).

Instead of admitting arbitrary (semantic) inclusions, this relation is defined structurally (syntactically) on type operators, in order to maintain feasible typechecking.
E.g. if A' <: A and B <: B' then A→B <: A'→B'.

Then, if b:B, B<:A, and f:A→C, then f(b) is legal.
This is very important for software extensibility and reusability.

However, if b:B, B<:A, and f:A→A is the identity, then f(b):A.
This is not good because we loose type information.

One way to fix the above problem, is to introduce bounded quantifiers:

id : All(A<:B) All(a:A) A
= fun(A<:B) fun(a:A) a

t : Pair(A<:Int) A×(Int→A)×(A→Int)
= pair(A<:Int=Nat) <0, abs, pred>

One way of generalizing bounded quantifiers is to introduce POWER kinds: POWER(B) is the kind of all subtypes of B.

A<:B   is now an abbreviation for A::POWER(B)

All(A<:B) C   abbreviates   All(A::POWER(B)) C
  (i.e. we no longer need special bounded quantifiers)

From "bind $(x_1..x_n)$=a in .. x_i .." to "a.x_i"

The notation "bind $(x_1..x_n)$ = a in b" is too sensitive to changes in the type of "a" (when software is evolving).

Also, simple selectors like "bind $(x_1..x_n)$ = a in x_i" are very cumbersome to write.

Programming languages universally use the *dot* notation "a.x_i" (named projections) as the elimination rule for tuples.

But note that the type rule for "bind" has restrictions on the type variables which can occur in the result type. To preserve these restrictions, one must transform programs to embed every occurrence of "a.x_i" in an adequate (i.e. large enough) context of the form:

let y = c in ... y.x_i ... y.x_j ... end

which can then be interpreted as equivalent to:

bind $x_1..x_n$ = c in ... x_i ... x_j ... end

i.e., the type rules for "a.x_i" are contextual (in the dependent case) and not easy to formulate concisely, although it is clear what should be done according to the above correspondence.

The loss of α-conversion

Again, one must embed an occurrence of "$y.x_i$" in a large enough context of the form:

$$\text{let } y = c \text{ in } \ldots\ y.x_i \ldots\ y.x_j \ldots \text{ end}$$

which can then be transformed into:

$$\text{bind } x_1..x_n = c \text{ in } \ldots\ x_i \ldots\ x_j \ldots \text{ end}$$

Unfortunately, the "large enough" context required by the above transformation may be hard to establish. The "c" quantity above may be out of reach, e.g. if "c" is defined in a different module: we cannot write a "let" which embraces multiple modules.

Although α-conversion is still possible on a large-scale, by α-converting all the modules involved, this is no longer a local operation of an individual construct.

I.e. α-conversion it does not *scale up* to large software systems.

So, the ordinary interpretation of "a.x" is that "x" is a fixed name,which is not subject to α-conversion. I.e. while it is clear that (with the usual free-variable restrictions):

$$\text{Some}(x{:}A)\ B\{x\} \quad \leftrightarrow_\alpha \quad \text{Some}(y{:}A)\ B\{y\}$$

$$\text{bind } (x_1..x_n) = c \text{ in } \ldots x_i \ldots x_j \ldots \text{ end}$$
$$\leftrightarrow_\alpha \quad \text{bind } (y_1..y_n) = c \text{ in } \ldots y_i \ldots y_j \ldots \text{ end}$$

the average programmer would be extremely surprised to see the following program typecheck:

```
let f = fun(t: Tuple  x:Int  y:Int  end) t.x + t.y
let a = tuple   let y=3   let x=4   end;
f(a)
```

This feeling is even stronger for abstract types: abstract types with different operator names are instinctively regarded as different abstract types, although the corresponding existential types may α-convert. Any such matching up to α-conversion would be considered an "accidental" match which should be trapped by the typechecker.

The consequence for our language is that signatures and bindings are *not* equivalent up to α-conversion.

This implies that even functions are not α-convertible. Functions are normally α-convertible in programming languages, but note that languages with a notion of "keyword parameters" (which, again, is advocated for large software systems) also loose α-convertibility to various degrees.

We however adopt a weaker form of α-conversion. In many situations, identifiers may be omitted; such *omitted* identifiers match any other identifier.

Compilation techniques

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

Compiler
Interactive, bootstrapped.
Recursive descent, in-core.
Full closures.
Producing bytecode (initially).

Linker
Interactive.
Version control.

Run-Time
Pickling (support for separate compilation and linking).
GC (Compacting).

Quest Syntax

```
Program ::=
   {[Interface | Module | Linkage | Binding] ";"}

Interface ::=
   ["unsound"] "interface" ide ["import" Import] "export" Signature "end"

Module ::=
   ["unsound"] "module" ide ":" ide ["import" Import] "export" Binding "end"

Linkage ::=
   "import" Import

Import ::=
   {[ideList] ":" ide}

Kind ::=
   ide |
   "TYPE" |
   "POWER" "(" Type ")" |
   "ALL" "(" TypeSignature ")" Kind |
   ide "_" ide |
   "{" Kind "}"

Type ::=
   ide {"." ide} |
   "Ok" | "Bool" | "Char" | "String" | "Int" | "Real" |
   "Array" "(" Type ")" | "Exception" |
   "All" "(" Signature ")" Type |
   "Tuple" Signature "end" |
   "Option" OptionSignature "end" |
   "Auto" [ide] HasKind "with" Signature "end" |
   "Record" ValueSignature "end" |
   "Variant" ValueSignature "end" |
   "Fun" "(" TypeSignature ")" [HasKind] Type |
   "Rec" "(" ide HasKind ")" Type |
   Type "(" TypeBinding ")" |
   Type infix Type |
   Type "_" ide |
   "{" Type "}"

Value ::=
   ide |
   "ok" | "true" | "false" | char | string | integer | real |
   "if" Test ["then" Binding] {"elsif" Test ["then" Binding]} ["else" Binding]"end"|
   "begin" Binding "end" |
   "loop" Binding "end" | "exit" |
   "while" Test "do" Binding "end" |
   "for" ide "=" Binding ( "upto" | "downto" ) Binding "do" Binding "end" |
   "fun" "(" Signature ")" [":" Type] Value |
   Value "(" Binding ")" |
   Value (infix | "is" | "isnot" | ":=") Value |
   "tuple" Binding "end" |
   "auto" (["let" ide [HasKind] "=" ] ":") Type "with" Binding "end" |
```

```
    "option" (ide | "ordinal" "(" Value ")") "of" Type ["with" Binding] "end" |
    "record" ValueBinding "end" |
    "variant" ["var"] ide "of" Type ["with" Value] "end" |
    Value ("." | "?" | "!") ide |
    "case" Binding CaseBranches "end" |
    "array" Binding "end" |
    Value "[" Binding "]" [ ":=" Value ] |
    "inspect" Binding InspectBranches "end" |
    "exception" ide [":" Type] "end" |
    "raise" Value ["with" Value] ["as" Type] "end"|
    "try" Binding TryBranches "end" |
    "{" Value "}"

Signature ::=
    { "DEF" KindDecl |
      "Def" ["Rec"] TypeDecl |
      [IdeList] HasKind |
      ["var"] IdeList (HasType|ValueFormals) | HasMutType }

TypeSignature ::=
    { "DEF" KindDecl |
      "Def" ["Rec"] TypeDecl |
      [IdeList] HasKind }

ValueSignature ::=
    {["var"] IdeList HasType}

OptionSignature ::=
    {IdeList ["with" Signature "end"]}

Binding ::=
    { "DEF" KindDecl |
      "Def" ["Rec"] TypeDecl |
      "Let" ["Rec"] TypeDecl |
      "let" ["rec"] ValueDecl |
      "::" Kind |
      ":" Type |
      "var" "(" Value ")" |
      "@" Value |
       Value }

TypeBinding ::=
    { Type }

ValueBinding ::=
    {["var"] ide "=" Value}

KindDecl ::=
    ide "=" Kind |
    KindDecl "and" KindDecl

TypeDecl ::=
    ide [HasKind | TypeFormals] "=" Type |
    TypeDecl "and" TypeDecl

ValueDecl ::=
    ["var"] ide [HasType | ValueFormals] "=" Value |
```

```
   ValueDecl "and" ValueDecl

TypeFormals ::=
   {"(" TypeSignature ")"} HasKind

ValueFormals ::=
   {"(" Signature ")"} ":" Type

Test :=
   Binding {"andif" Test | "orif" Test}

CaseBranches ::=
   {"when" IdeList ["with" ide [":" Type]] "then" Binding}
   ["else" Binding]

InspectBranches ::=
   {"when" Type ["with" {IdeList [":" Type]}] "then" Binding}
   ["else" Binding]

TryBranches ::=
   {"when" Binding ["with" ide [":" Type]] "then" Binding}
   ["else" Binding]

HasType ::=
   ":" Type

HasMutType ::=
   ":" Type | ":" "Var" "(" Type ")"

HasKind ::=
   "<:" Type | "::" Kind

IdeList ::=
   ide | ide "," IdeList

Operators:
   prefix monadic:      not minus size
   infix:            /\  \/  +  -  *  /  %  <  >  <=  >=  <>

Keywords:
DEF ALL Array Auto Def All Let Fun Option Rec Record Tuple Var Variant and andif
array auto begin case downto else elsif end exception export for fun if import
inspect interface is isnot let loop module of orif option ordinal raise rec record
then try tuple unsound upto var variant when while with ? ! : :: <: := = _ @

Notes:
The @ keywords can only appear in actual-parameter bindings;.
Bindings evaluated for a single result must end with a value component (modulo
manifest declarations).
Bindings in the array construct may begin with a type (the type of array elements)
and must then contain only values.
Recursive value bindings can only contain constructors, i.e. functions, tuples, etc.
```

Formal systems

General principles

Typed (and untyped) λ-calculi can be described as *formal systems* based on *judgements*:

$$E \vdash \Phi$$

Where E is a list of *assumptions* for variables free in Φ, and Φ is the *conclusion*. Normally E is called the *environment*.

Formal systems for type inference are called *type inference systems*.

Common judgements in type inference systems are "a given term has a given type in a given environment" and "two given terms are equal members of a given type, in a given environment".

Judgements like $E \vdash \Phi$ are used to define a *relation* $\vdash$ (*entailment*) between assumptions and conclusions, which determines the *valid* conclusions.

This relation is normally defined inductively by *axioms* and *inference rules*:

$$E_0 \vdash \Phi_0$$

$$\frac{E_1 \vdash \Phi_1 \quad \ldots \quad E_n \vdash \Phi_n}{E_{n+1} \vdash \Phi_{n+1}}$$

(and nothing else is in the $\vdash$ relation)

First-order typed λ-calculus (with subtyping)

Types (X variables; K constants; l labels; A,B,C types)

K_i	basic constants and operators
$A \rightarrow B$	functions

extensions

$A \times B$	pairs
$\langle l_1{:}A_1, \ldots, l_n{:}A_n \rangle$	records
$[l_1{:}A_1, \ldots, l_n{:}A_n]$	variants
X	variables (just for recursion)
Rec(X) A	recursive types

Terms (x variables; k constants; t tags; a,b,c terms)

x	
k_{ij}	
fun(x:A) b	b(a)

extensions

<a,b>	lft(c) rht(c)
$\langle l_1{=}a_1, \ldots, l_n{=}a_n \rangle$	c.l
[l=a] as A	case c $\mid$ $[l_1{=}x_1]\ b_1$ $\mid \ldots \mid$ $[l_n{=}x_n]\ b_n$
rec(x:A) a	

Free variables (omitted)

Substitution (omitted)

Environments (E)

$\varnothing$
E, x:A

extensions

E, X type	for recursion
E, x:A	for subtyping

Judgements

$\vdash$ E env	E is a well-formed environment
E $\vdash$ A type	A is a legal type (trivial)
E $\vdash$ a:A	a has type A
E $\vdash$ A$\leftrightarrow$B	A and B are equivalent types (trivial)
E $\vdash$ a$\leftrightarrow$b:A	a and b equivalent terms of type A

extensions

E $\vdash$ A<:B	A is a subtype of B
E $\vdash$ C$\downarrow$X	C is contractive in X

General rules

Pure calculus

Environments

$$\vdash \varnothing \text{ env}$$

$$\frac{E \vdash A \text{ type} \qquad x \notin Dom(E)}{\vdash E, x{:}A \quad \text{env}}$$

Type congruence
(↔ is a substitutive equivalence relation over well-formed types)

$$\frac{E \vdash A \leftrightarrow B}{E \vdash B \leftrightarrow A} \qquad \frac{E \vdash A \leftrightarrow B \quad E \vdash B \leftrightarrow C}{E \vdash A \leftrightarrow C}$$

$$\frac{E \vdash K_i \text{ type}}{E \vdash K_i \leftrightarrow K_i} \qquad \frac{E \vdash A \leftrightarrow A' \quad E \vdash B \leftrightarrow B'}{E \vdash A \rightarrow B \leftrightarrow A' \rightarrow B'}$$

Congruence
(↔ is a substitutive equivalence relation over the syntax of terms)

(omitted)

Recursion extensions

Environments

$$\frac{\vdash E\ \mathrm{env} \qquad X \notin \mathrm{Dom}(E)}{\vdash E, X\ \mathrm{type}\quad \mathrm{env}}$$

Contractiveness (eliminates types such as Rec(X)X*)*

$$\frac{E, X\ \mathrm{type} \vdash K\ \mathrm{type}}{E \vdash K{\downarrow}X} \qquad \frac{E, X\ \mathrm{type} \vdash A\ \mathrm{type} \quad E, X\ \mathrm{type} \vdash B\ \mathrm{type}}{E \vdash (A \to B){\downarrow}X}$$

$$\frac{E, X\ \mathrm{type} \vdash Y\ \mathrm{type} \qquad Y \neq X}{E \vdash Y{\downarrow}X} \qquad \frac{E, Y\ \mathrm{type} \vdash A{\downarrow}X \quad E, X\ \mathrm{type} \vdash A{\downarrow}Y \qquad Y \neq X}{E \vdash (\mathrm{Rec}(Y)A){\downarrow}X}$$

Type Equivalence

$$\frac{E \vdash \mathrm{Rec}(X)\ A\ \ \mathrm{type} \qquad Y \notin \mathrm{Dom}(E)}{E \vdash \mathrm{Rec}(X)A \leftrightarrow \mathrm{Rec}(Y)\ A\{X \leftarrow Y\}}$$

$$\frac{E \vdash \mathrm{Rec}(X)\ A\ \ \mathrm{type}}{E \vdash \mathrm{Rec}(X)\ A \leftrightarrow A\{X \leftarrow \mathrm{Rec}(X)\ A\}}$$

$$\frac{E \vdash A \leftrightarrow C\{X \leftarrow A\} \quad E \vdash B \leftrightarrow C\{X \leftarrow B\} \quad E \vdash C{\downarrow}X}{E \vdash A \leftrightarrow B}$$

Retyping

$$\frac{E \vdash a{:}A \quad E \vdash A \leftrightarrow B}{E \vdash a{:}B}$$

Subtyping extensions

Environments

$$\frac{E \vdash A \text{ type} \qquad X \notin Dom(E)}{\vdash E, X{<:}A \;\; env}$$

Reflexivity

$$\frac{E \vdash A \leftrightarrow B}{E \vdash A{<:}B}$$

Transitivity

$$\frac{E \vdash A{<:}B \qquad E \vdash B{<:}C}{E \vdash A{<:}C}$$

Subsumption

$$\frac{E \vdash a{:}A \qquad E \vdash A{<:}B}{E \vdash a{:}B}$$

(Retyping is now derivable)

Specific typing rules

Formation	*Introduction*	*Elimination*
$\vdash E\ \mathrm{env}$ $E, X\ \mathrm{type} \vdash X\ \mathrm{type}$	$E \vdash A\ \mathrm{type}$ $E, x{:}A \vdash x{:}A$	
$\vdash E\ \mathrm{env}$ $E \vdash K_i\ \mathrm{type}$	$\vdash E\ \mathrm{env}$ $E \vdash k_{i,j}{:}K_i$	
$E \vdash A\ \mathrm{type} \quad E \vdash B\ \mathrm{type}$ $E \vdash A \rightarrow B\ \mathrm{type}$	$E, x{:}A \vdash b{:}B$ $E \vdash \mathrm{fun}(x{:}A)\ b : A \rightarrow B$	$E \vdash b{:}A \rightarrow B \quad E \vdash a{:}A$ $E \vdash b(a) : B$

Extensions

$E \vdash A\ \mathrm{type} \quad E \vdash B\ \mathrm{type}$ $E \vdash A \times B\ \mathrm{type}$	$E \vdash a{:}A \quad E \vdash b{:}B$ $E \vdash <a,b> : A \times B$	$E \vdash c{:}A \times B$ $E \vdash \mathrm{lft}(c){:}A \quad E \vdash \mathrm{rht}(c){:}B$
$E \vdash A_i\ \mathrm{type}$ $E \vdash \langle l_i{:}A_i \rangle\ \mathrm{type}$	$E \vdash a_i{:}A_i$ $E \vdash \langle l_i{=}a_i \rangle : \langle l_i{:}A_i \rangle$	$E \vdash c{:}\langle l_i{:}A_i \rangle$ $E \vdash c.l_j : A_j$
$E \vdash A_i\ \mathrm{type}$ $E \vdash [l_i{:}A_i]\ \mathrm{type}$	$E \vdash a{:}A_j$ $E \vdash [l_j{=}a]\ \mathrm{as}\ [l_i{:}A_i] : [l_i{:}A_i]$	$E \vdash c{:}[l_i{:}A_i] \quad E, x_i{:}A_i \vdash b_i{:}C$ $E \vdash \mathrm{case}\ c \mid [l_i{=}x_i]\ b_i : C$
$E \vdash A{\downarrow}X$ $E \vdash \mathrm{Rec}(X)\ A\ \mathrm{type}$	$E, x{:}A \vdash a{:}A$ $E \vdash \mathrm{rec}(x{:}A)\ a : A$	

Specific subtyping rules

Subtyping

$$\vdash E\ env \qquad \text{for some pairs } i,j$$
$$E \vdash K_i <: K_j$$

$$E \vdash A' <: A \quad E \vdash B <: B'$$
$$E \vdash A \rightarrow B <: A' \rightarrow B'$$

Extensions

$$E \vdash A <: A' \quad E \vdash B <: B'$$
$$E \vdash A \times B <: A' \times B'$$

$$E \vdash A_i <: B_i \quad E \vdash A_j\ type$$
$$E \vdash \langle l_i{:}A_i,\ l_j{:}A_j \rangle <: \langle l_i{:}B_i \rangle$$

$$E \vdash A_i <: B_i \quad E \vdash B_j\ type$$
$$E \vdash [l_i{:}A_i] <: [l_i{:}B_i,\ l_j{:}B_j]$$

$$E, Y\ type, X <: Y \vdash A <: B \qquad X \neq Y \quad X \notin FV(B) \quad Y \notin FV(A)$$
$$E \vdash Rec(X)\ A <: Rec(Y)\ B$$

Specific computation rules

Computation

(omitted)

$$\frac{E, x{:}A \vdash b{:}B \quad y \notin Dom(E)}{E \vdash fun(x{:}A)b \leftrightarrow fun(y{:}A)b\{x \leftarrow y\} : B}$$

$$\frac{E, x{:}A \vdash b{:}B \quad E \vdash a{:}A}{E \vdash (fun(x{:}A)b)(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

Extensions

(omitted)

(omitted)

(omitted)

$$\frac{E, x{:}A \vdash a{:}A \quad y \notin Dom(E)}{E \vdash rec(x{:}A)a \leftrightarrow rec(y{:}A)\, a\{x \leftarrow y\} : A}$$

$$\frac{E, x{:}A \vdash a{:}A}{E \vdash rec(x{:}A)\, a \leftrightarrow a\{x \leftarrow rec(x{:}A)\, a\} : A}$$

Quest core formal system

Syntax

Signatures (S):
$\emptyset$
S, X::K
S, x:A

Bindings (D):
$\emptyset$
D, Let X::K=A
D, let x:A=a

Kinds (K, L, M):
TYPE
ALL(X::K)L
POWER(A)

Types and operators
(A,B,C; *type idents* X,Y,Z; *labels* l):

X
All(S)A
Tuple S end
Fun(X::K)A
A(B)

Record l_1:A_1, ... , l_n:A_n end
Variant l_1:A_1, ... , l_n:A_n end
Set A_1, ... , A_n end
Rec(X::TYPE) A

Values (a,b,c; *value idents* x,y,z):

x
fun(S) a
a(D)
tuple D end
bind S = a in b end

record l_1=a_1, ... , l_n=a_n end
a.t
variant l=a end
case a ... when(l_i=x_i) b_i ... end
set a_1, ... , a_n end ...
rec(x:A)a

Let t = **tuple Let** A = Int **let** a = 3 **end**
... :t.A ... t.a ...

$\approx$

bind A::TYPE a:Int = **tuple Let** A = Int **let** a = 3 **end**
in ... :A ... a ...

Judgements

Formation

$\vdash$ S sig	S is a signature
S $\vdash$ K kind	K is a kind
S $\vdash$ A type	A is a type (same as S $\vdash$ A::TYPE)

Equivalence

S $\vdash$ S'<.∴.>S"	equivalent signatures
S $\vdash$ K<::>L	equivalent kinds
S $\vdash$ A<:>B	equivalent types

Inclusion

S $\vdash$ S'<.∴.S"	S' is a subsignature of S"
S $\vdash$ K<::L	K is a subkind of L
S $\vdash$ A<:B	A is a subtype of B (same as S $\vdash$ A::POWER(B))

Membership

S $\vdash$ D.∴.S'	D has signature S'
S $\vdash$ A::K	A has kind K
S $\vdash$ a:A	a has type A

Notation

S S' is the concatenation (iterated extension) of S with S'.

Signatures and bindings are ordered, however we freely use the notation X∈Dom(S) (type X is defined in S), x∈Dom(S) (value x is defined in S), S(X) (the kind of X in S), and S(x) (the type of x in S). Similarly for bindings, where D(X) is the type associated with X in D, and D(x) is the value associated with x in D.

E{X←A} denotes the substitution of the type variable X by the type A within an expression E of any sort.

E{D} denotes the sequential substitution of all the variables X∈Dom(D) and x∈Dom(D) by D(X) and D(x), within an expression E of any sort.

Equivalence of signatures, kinds and types

(Omitted. It involves reflexivity, transitivity, congruence, and typed β- and η-conversion of type operators.)

Self Inclusion

$S \vdash S' <\therefore> S''$ $\quad$ $S \vdash K <::> L$ $\quad$ $S \vdash A <:> B$

$S \vdash S' <\therefore S''$ $\quad$ $S \vdash K <:: L$ $\quad$ $S \vdash A <: B$

Subsumption

$S \vdash D \therefore S' \quad S \vdash S' <\therefore S''$ $\quad$ $S \vdash A::K \quad S \vdash K <:: L$ $\quad$ $S \vdash a:A \quad S \vdash A <: B$

$S \vdash D \therefore S''$ $\quad$ $S \vdash A :: L$ $\quad$ $S \vdash a : B$

Conversion

$S, X::K \vdash B::L \quad S \vdash A::K$

$S \vdash (\mathrm{Fun}(X::K)B)(A) <:> B\{X \leftarrow A\}$

Signatures

$$\vdash \varnothing \text{ sig}$$

$$S \vdash K \text{ kind} \qquad X \notin Dom(S)$$
$$\vdash S, X{::}K \text{ sig}$$

$$S \vdash A \text{ type} \qquad x \notin Dom(S)$$
$$\vdash S, x{:}A \text{ sig}$$

Bindings

$$\vdash S \text{ sig}$$
$$S \vdash \varnothing \therefore \varnothing$$

$$S \vdash D \therefore S' \qquad S \vdash A\{D\}{::}K\{D\}$$
$$S \vdash D, \text{Let } X{::}K{=}A \therefore S', X{::}K$$

$$S \vdash D \therefore S' \qquad S \vdash a\{D\}{:}A\{D\}$$
$$S \vdash D, \text{let } x{:}A{=}a \therefore S', x{:}A$$

Ex.

$$S \vdash Int :: TYPE$$
$$S \vdash \varnothing \therefore \varnothing \qquad S \vdash Int\{\varnothing\} :: TYPE\{\varnothing\} \qquad\qquad S \vdash 3 : Int$$
$$S \vdash \varnothing, \text{Let } A{::}TYPE{=}Int \therefore \varnothing, A{::}TYPE \qquad S \vdash 3\{\varnothing, \text{Let } A{=}Int\} : A\{\varnothing, \text{Let } A{=}Int\}$$
$$S \vdash \varnothing, \text{Let } A{::}TYPE{=}Int, \text{let } a{:}A{=}3 \therefore \varnothing, A{::}TYPE, a{:}A$$

Kinds

$$\frac{\vdash S \text{ sig}}{S \vdash \text{TYPE kind}}$$

$$\frac{S \vdash K \text{ kind} \quad S, X::K \vdash L \text{ kind}}{S \vdash \text{ALL}(X::K)L \text{ kind}}$$

$$\frac{S \vdash A \text{ type}}{S \vdash \text{POWER}(A) \text{ kind}}$$

Types and Operators

$$\frac{\vdash S \text{ sig} \quad X \in \text{Dom}(S)}{S \vdash X :: S(X)}$$

$$\frac{S\ S' \vdash A \text{ type}}{S \vdash \text{All}(S')A \text{ type}}$$

$$\frac{\vdash S\ S' \text{ sig}}{S \vdash \text{Tuple } S' \text{ end type}}$$

$$\frac{S, X::K \vdash B::L}{S \vdash \text{Fun}(X::K)B :: \text{ALL}(X::K)L}$$

$$\frac{S \vdash B::\text{ALL}(X::K)L \quad S \vdash A::K}{S \vdash B(A) :: L\{X \leftarrow A\}}$$

$$\frac{S \vdash A_1 \text{ type} \quad S \vdash A_n \text{ type}}{S \vdash \text{Record } l_1:A_1, \ldots, l_n:A_n \text{ end type}}$$

$$\frac{S \vdash A_1 \text{ type} \quad S \vdash A_n \text{ type}}{S \vdash \text{Variant } l_1:A_1, \ldots, l_n:A_n \text{ end type}}$$

$$\frac{S \vdash A_1 \text{ type} \quad S \vdash A_n \text{ type}}{S \vdash \text{Set } A_1, \ldots, A_n \text{ end type}}$$

$$\frac{S, X\text{::TYPE} \vdash A \text{ type}}{S \vdash \text{Rec}(X\text{::TYPE})\ A \text{ type}}$$

Values

$$\frac{\vdash S \text{ sig} \qquad x \in \text{Dom}(S)}{S \vdash x : S(x)}$$

$$\frac{S\ S' \vdash a{:}A}{S \vdash \text{fun}(S')a : \text{All}(S')A}$$

$$\frac{S \vdash a{:}\text{All}(S')A \quad S \vdash D {\therefore} S'}{S \vdash a(D) : A\{D\}}$$

$$\frac{S \vdash D {\therefore} S'}{S \vdash \text{tuple } D \text{ end} : \text{Tuple } S' \text{ end}}$$

$$\frac{S \vdash a{:}\text{Tuple } S' \text{ end} \quad S \vdash B \text{ type} \quad S\ S' \vdash b{:}B}{S \vdash \text{bind } S' = a \text{ in } b \text{ end} : B \text{ end}}$$

$$\frac{S \vdash a_1{:}A_1 \ \ldots\ S \vdash a_n{:}A_n}{S \vdash \text{record } l_1{=}a_1, \ldots, l_n{=}a_n \text{ end} : \text{Record } l_1{:}A_1, \ldots, l_n{:}A_n \text{ end}}$$

$$\frac{S \vdash a : \text{Record } t_1{:}A_1, \ldots, t_n{:}A_n \text{ end} \qquad i \in 1..n}{S \vdash a.l_i : A_i}$$

$$\frac{S \vdash B_1 \text{ type} \ \ldots\ S \vdash B_m \text{ type} \qquad \forall i \in 1..n\ \exists j \in 1..m\ S \vdash a_i{:}Bj}{S \vdash \text{set } a_1, \ldots, a_n \text{ end} : \text{Set } B_1, \ldots, B_m \text{ end}}$$

$$S, x{:}A \vdash a{:}A$$

$$S \vdash rec(x{:}A)a : A$$

Ex.

$S \vdash \text{alg} : \text{Tuple } A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int end}$
$S \vdash \underline{\text{Int}} :: \text{TYPE}$
$S, A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int} \vdash \underline{f(a)} : \underline{\text{Int}}$

$S \vdash \text{bind } A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int} = \text{alg in } \underline{f(a)} \text{ end} : \underline{\text{Int}}$

$S \vdash \text{alg} : \text{Tuple } A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int end}$
$S \vdash \underline{\text{Int}} :: \text{TYPE}$
$S, A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int} \vdash \underline{a+1} : \underline{\text{Int}}$?

$S \vdash \text{bind } A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int} = \text{alg in } \underline{a+1} \text{ end} : \underline{\text{Int}}$?

$S \vdash \text{alg} : \text{Tuple } A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int end}$
$S \vdash \underline{A} :: \text{TYPE}$?
$S, A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int} \vdash \underline{a} : \underline{A}$

$S \vdash \text{bind } A\text{::TYPE}, a\text{:}A, f\text{:}A \rightarrow \text{Int} = \text{alg in } \underline{a} \text{ end} : \underline{A}$?

SubSignatures

$$\frac{S \vdash S' <\therefore S'' \quad S\,S' \vdash K <:: L}{S \vdash S', X::K \ <\therefore\ S'', X::L}$$

$$\frac{S \vdash S' <\therefore S'' \quad S\,S' \vdash A <: B}{S \vdash S', x:A \ <\therefore\ S'', x:B}$$

SubKinds

$$\frac{S \vdash K' <:: K \quad S, X::K' \vdash L <:: L'}{S \vdash ALL(X::K)L <:: ALL(X::K')L'}$$

$$\frac{S \vdash A <: B}{S \vdash POWER(A) <:: POWER(B)}$$

$$\frac{S \vdash A \text{ type}}{S \vdash POWER(A) <:: TYPE}$$

SubTypes

$$\frac{S \vdash S'' <\therefore S' \quad S, S'' \vdash A' <: A''}{S \vdash All(S')A' <: All(S'')A''}$$

$$\frac{\vdash S S' S'' \text{ sig} \quad S \vdash S' <\therefore S'''}{S \vdash \text{Tuple } S'S'' \text{ end} <: \text{Tuple } S''' \text{ end}}$$

$$\frac{S \vdash A_1 <: B_1 \ \ldots\ S \vdash A_n <: B_n \ \ldots\ S \vdash A_m \text{ type}}{S \vdash \text{Record } l_1:A_1, \ldots, l_n:A_n, \ldots, l_m:A_m \text{ end} <: \text{Record } l_1:B_1, \ldots, l_n:B_n \text{ end}}$$

$$\frac{S \vdash A_1 <: B_1 \ \ldots\ S \vdash A_n <: B_n \ \ldots\ S \vdash B_m \text{ type}}{S \vdash \text{Variant } l_1:A_1, \ldots, l_n:A_n \text{ end} <: \text{Variant } l_1:B_1, \ldots, l_n:B_n, \ldots, l_m:B_m \text{ end}}$$

$$\frac{\forall i \in 1..n\ \exists j \in 1..m\ S \vdash A_i <: B_j}{S \vdash \text{Set } A_1, \dots, A_n \text{ end} <: \text{Set } B_1, \dots, B_m \text{ end}}$$

Quest typing

Function types

An object of type `All(x:A)B` (or simply `A→B`) is a function `fun(x:A)b` of argument `x:A` and result `b:B`

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

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

intId(3);
```

Parametric polymorphism

(Girard-Reynolds)

An object of type **`All`**`(X::K)B{X}` is a polymorphic function **`fun`**`(X::K)b` of argument `A::K` and result `b:B{X←A}`

```
Let Id::TYPE =
  All(A::TYPE) All(x:A) A;

let id:Id =
  fun(A::TYPE) fun(x:A) x;

id(:Int);
  =   fun(x:Int) x          = intId
  :   All(x:Int) Int        = IntId

id(Int)(3);
  = 3 : Int
```

A polymorphic function can be applied to *its own* type:

`id(:Id)(id)` uses impredicativity in an essential way

Type operators

An object of kind **`ALL`**`(X::K)L{X}` is an operator **`Fun`**`(X::K)b` of argument `A::K` and result `B::L{X←A}`

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

Int -> Int;             ->(Int)(Int)
  = All(x:Int) Int :: TYPE
```

Simple tuples

An object of type **Tuple** x:A, y:B **end** is a pair of left component a:A and right component b:B.

```
Let IntPackage::TYPE =
  Tuple x:Int f:Int→Int end;

let intPackage::IntPackage =
  tuple let x=0 let f=succ end;

intPackage.f(intPackage.x);
  = 1:Int
```

Abstract types

(Mitchell-Plotkin)

An object of type **`Tuple`** `X::K, y:B{X}` **`end`** is a package of representation `A::K` and implementation `y:B{X←A}`.

```
Let Package::TYPE =
  Tuple
    A::TYPE                         abstract type
    a:A f:A→Int                       interface
  end;

let package:Package =
    tuple
      Let A=Int                     hidden representation
      let a=0 let f=succ            implementation
    end;
```

An element of an abstract type can be implemented by *its own* type

```
let package1:Package =
    tuple
      Let A=Package
      let a=package
      let f(p:Package)=p.f(p.a)
    end;
```

uses impredicativity in an essential way

Polymorphism + abstract types

```
Let Stack :: ALL(A::TYPE)TYPE =
  Fun(A::TYPE)
    Tuple
      S::TYPE
      empty: S
      push: All(a:A s:S)S
      top: All(s:S)A
      pop: All(s:S)S
    end;

:Stack(Int)
```

Quest subtyping

Power Kinds

For any type `A`, `POWER(A)` is the kind of all the subtypes of `A`.

`A::POWER(B)` means that `A<:B` (`A` is a subtype of `B`)

```
A <: B          ≈ A::POWER(B)
fun(A<:B) c     ≈ fun(A::POWER(B)) c
All(A<:B) C     ≈ All(A::POWER(B)) C
```

Formation

$$\frac{S \vdash A \text{ type}}{S \vdash \mathrm{POWER}(A) \text{ kind}}$$

Introduction

$$\frac{S \vdash A{<:>}B}{S \vdash A <: B}$$

Elimination

$$\frac{S \vdash a{:}A \quad S \vdash A{<:}B}{S \vdash a : B}$$

Power-Power

$$\frac{S \vdash A{<:}B}{S \vdash \mathrm{POWER}(A) <:: \mathrm{POWER}(B)}$$

Power-Type

$$\frac{S \vdash A \text{ type}}{S \vdash \mathrm{POWER}(A) <:: \mathrm{TYPE}}$$

Records

```
Let Object =
  Record age:Int end;
Let Vehicle =
  Record age:Int speed:Int end;
Let Machine =
  Record age:Int fuel:String end;
Let Car =
  Record age:Int speed:Int fuel:String end;
```

Subtyping is *multiple*

```
          <:   Vehicle   <:
    Car                        Object
          <:   Machine   <:
```

Subtyping is *structural*

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

let myCar: Car =
  record age=3 speed=120 fuel="gas" end;
```

Subtyping works *in width* and *in depth*

$$\frac{S \vdash A_1 <: B_1 \quad \dots \quad S \vdash A_n <: B_n \quad \dots \quad S \vdash A_m \text{ type}}{S \vdash \text{Record } t_1{:}A_1, \dots, t_n{:}A_n, \dots, t_m{:}A_m \text{ end} <: \text{Record } t_1{:}B_1, \dots, t_n{:}B_n \text{ end}}$$

Higher-order Subtypes

$$\frac{S \vdash A' <: A \quad S \vdash B <: B'}{S \vdash A \to B <: A' \to B'}$$

$$\frac{S \vdash S'' <\therefore S' \quad S, S'' \vdash A' <: A''}{S \vdash \mathrm{All}(S')A' <: \mathrm{All}(S'')A''}$$

```
let speed: Vehicle → Int = ... ;
  Vehicle → Int    <:    Car → Int
```
(`speed` *takes cars*)

```
let speed': All(A<:Vehicle)A→Int = ... ;
  All(A<:Vehicle)A→Int   <:   All(A<:Car)A→Int

let serialNo: Int → Car = ... ;
  Int → Car    <:    Int → Vehicle
```
(`serialNo` *returns vehicles*)

```
let f: Vehicle → Vehicle = ...
  Vehicle → Vehicle    <:    Car → Object

age(f(myCar));

let f': All(A<:Vehicle) A → A = ... ;
      (f': Vehicle → Vehicle)

age(:Car)(f'(:Car)(myCar));
```
(*f'* *, used as* `Car → Object`)

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

The subsumption rule

$$\frac{S \vdash a{:}A \quad S \vdash A{<:}B}{S \vdash a : B}$$

Then `myCar:Car` implies `myCar:Object`.

```
let age': Object → Int =
  fun(x:Object) x.age;

age'(myCar);
  = 3:Int
```

The subsumption rule is useful but not sufficient by itself:

```
let objId': Object → Object =
  fun(a:Object) a;

objId'(myCar);
  = myCar:Object

objId'(myCar).speed;
```

Wrong!

Subtyping + polymorphism

```
let age: All(A<:Object) A → Int =
  fun(A<:Object) fun(x:A) x.age;
```

Here we must check `A::TYPE`, but `A::POWER(Object)`.
Hence we use embedding (`POWER(X)<::TYPE`).
Then we must check that `x` has a record type in `x.age`,
but `x:A`. However `A<:Object` , hence by subsumption
`x:Object`.

```
age(:Car)(myCar);
  = 3:Int
```

`age(myCar);` *the usual abbreviation*

```
  = 3:Int
```

```
let objId : All(A<:Object) A → A =
  fun(A<:Object) fun(x:A) x;
```

```
objId(:Car)(myCar);
  = myCar:Car
```

```
objId(:Car)(myCar).speed;
  = 120:Int
```

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

Subtyping + abstract types

Abstract subtypes

```
Let Package::Type =
  Tuple A::TYPE a:A end;

Let ExtendedPackage::Type =
  Tuple A::TYPE a:A f:A→Int end;
```

Partially abstract types

```
Let ObjectPackage::Type =
  Tuple A<:Object f:A→Int end;

let objectPackage:ObjectPackage =
  tuple Let A=Car let f=speed end;
```

Set types

(Buneman-Ohori)

```
Let AllCars::TYPE =
  Set Car end;
```

(type of all cars)

```
let allCars: AllCars =
  set myCar yourCar end
```

(extension of all cars)

```
allCars |><| set record age=3 end end;
  = set myCar end
```

$$\frac{S \vdash B_1 \text{ type} \quad \dots \quad S \vdash B_m \text{ type} \qquad \forall i \in 1..n\ \exists j \in 1..m\ S \vdash a_i : Bj}{S \vdash \text{set } a_1, \dots, a_n \text{ end} : \text{Set } B_1, \dots, B_m \text{ end}}$$

Subtyping + set types

(Buneman-Ohori)

```
Let AllObjects::TYPE =
  Set Object end;
Let AllVehiclesAndMachines::TYPE =
  Set Vehicle Machine end;
Let AllCars::TYPE =
  Set Car end;

  AllCars<:AllVehiclesAndMachines<:AllObjects

let allCars:AllCars =
  set myCar yourCar end;

allCars |><| set record age=3 end end;
  = set myCar end


LET RELATION = POWER(Set Record end end)

AllObjects :: RELATION

Join :: ALL(A,B::RELATION)RELATION
|><| : All(A,B::RELATION)(A#B)->Join(A B)
```

$$\frac{\forall i \in 1..n\ \exists j \in 1..m\ S \vdash A_i <: B_j}{S \vdash \text{Set } A_1, \ldots, A_n \text{ end} <: \text{Set } B_1, \ldots, B_m \text{ end}}$$

Classes and methods (Hint)

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

```
Let Rec Counter::TYPE =
  Record
    var count: Int
    fetch: Ok -> Int
    incr: Int -> Counter
  end;
```

A *class* (= *object generator*):

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

An *object* (= *object generator*):

```
let count = newCounter(0);
```

Invocation of *methods*:

```
count.incr(3).fetch();
```

Modules and interfaces

Programming in the large

The usefulness, even necessity, of typeful programming is most apparent in the construction of large systems

Large programs have the property that no single person can understand or remember all of their details at the same time.

Large programs must be split into *modules*, for better understanding and maintenance.

Module boundaries are called *interfaces*. They declare the types (or kinds) of identifiers supplied by modules; that is they describe how modules plug together to form systems.

An interface may provide:
- A collection of types to be used by many modules.
- A collection of related routines.
- One or more abstract types with operations.

Both interfaces and modules may import other interfaces or modules.

Both interfaces and modules export a set of identifiers.

State of the art: Modula2/Modula3:
- Advantages:
  - Nice and simple.
- Problems: does not support multiple implementations, parametric modules, or first-class modules.

Modules and interfaces

In Quest, each interface, say A, can be implemented by many modules, say b and c. Each module specifies the interface it implements:

```
interface A                  module b:A
import ..                      import ..
export                       export
  ..                              ..
end;                            end;
```

The following line imports:
- interfaces C, D, and E;
- module c implementing C;
- modules d1 and d2 both implementing D.

```
import  c:C  d1,d2:D  :E
```

Imported modules are just tuples (hence first-class).

Imported interfaces are just tuple types.

Separate compilation and linking

Interfaces can be *separately compiled* (after the interfaces they recursively import) (imported modules do not matter).

Modules can be *separately compiled* (after the interfaces they recursively import) (imported modules do not matter).

Modules are *linked* by importing them at the top level (after all the involved modules and interfaces are compiled):

Version checking ensures consistency.

```
import b:A;

  let b:A = ..
```

The result is the definition at the top-level of a tuple b, of type A, from which values and types can be extracted in the usual fashion.

Diamond import

A module d imports two modules c and b which both import a module a. Then the types flowing from a to d through two different import paths are made to interact in d.

```
interface A
export
  T::TYPE
  new(x:Int):T
  int(x:T):Int
end;
```

```
module a:A
export
    Let T::TYPE = Int
    let new(x:Int):T = x
    and int(x:T):Int = x
  end;
```

```
interface B
import a:A
export
  x:a.T
end;
```

```
module b:B
  import a:A
export
    let x = a.new(0)
  end;
```

```
interface C
import a:A
export
  f(x:a.T):Int
end;
```

```
module c:C
  import a:A
export
    let f(x:a.T):Int =
      a.int(x)+1
  end;
```

```
interface D
export
  z:Int
end;
```

```
module d:D
import b:B c:C
  export
    let z = c.f(b.x)
  end;
```

Note that the application `c.f(b.x)` in module d typechecks because the a imported by b and the a imported by c are the "same" implementation of the interface A, since a is a global external name.

To illustrate the correspondence between interfaces and signatures, and between modules and bindings, we can rephrase the diamond import example as follows.

```
Let A::TYPE =           let a:A =
  Tuple                     tuple
    T::TYPE                   Let T::TYPE = Int
    new(x:Int):T              let new(x:Int):T = x
    int(x:T):Int              and int(x:T):Int = x
  end;                      end;

Let B::TYPE =           let b:B =
  Tuple                     tuple
    x:a.T                     let x = a.new(0)
  end;                      end;

Let C::TYPE =           let c:C =
  Tuple                     tuple
    f(x:a.T):Int              let f(x:a.T):Int =
  end;                          a.int(x)+1
                            end;

Let D::TYPE =           let d:D =
  Tuple                     tuple
    z:Int                     let z = c.f(b.x)
  end;                      end;
```

In this case, `c.f(b.x)` typechecks because the types of `c.f` and `b.` both refer to the same variable `a` which is lexically in the scope of both `c` and `b`.

Module combination

System modelling

When programs first started becoming *large* (hundreds of procedures), it became necessary to split them into pieces. Eventually this trend led to modules and interfaces.

Today programs are starting to become *huge* (hundreds of interfaces). Unfortunately, interface systems have a *flat* structure (this is also an advantage).

It would clearly be desirable to be able to group interfaces into systems which could then be grouped into larger systems, and so on.

Major approaches:

Unix "make"

Properties:

Language independent.

Problems:

Cannot know about true dependencies.

Unreliable (hand-generated).

Pebble

Properties:

Parametric modules (universally quantified).

First-class modules, multiple implementations.

Module composition obtained by (dependent) function application.

Problems: the interface of each parametric module may have to express the *entire* module hierarchy above it. Practically unusable without additional assistance.

Standard ML

Properties:

Parametric modules (existentially quantified, predicative hierarchy).

Multiple implementations.

"Sharing constraints" to fix the Pebble problem.

Module composition obtained by (dependent) function application.

Problems:

Modules are not first-class.

Needs additional notion: "sharing constraints".

A different approach

Motivated by ease of reconfiguration of subsystems.

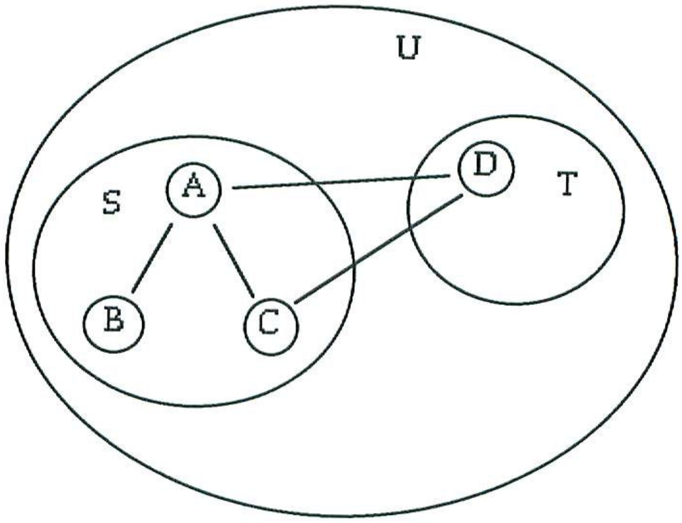
Classify systems as *open, closed,* and *sealed,* based on *membership* and *visibility* restrictions.

Open systems have no restriction regarding membership or visibility. Each module can *claim* membership to one or more (open) systems and can import from any (open) system.
System structure can be reorganized very easily just by changing membership claims and without affecting unrelated parts; this flexibility is important in initial stages of development. At the same time, the membership claims provide some degree of structuring.

Closed systems explicitly export interfaces, and only these interfaces are visible from the outside. However membership is still unrestricted.
The latter property facilitates access to "friends" of the system developers, while limiting visibility to the "public". This reflects intermediate stages of development.

Sealed systems have a membership list, as well as an interface export list.
Sealed systems can implement large abstract types composed of many modules, and are protected from interference. This reflects the final stages of development.

Consider the following system organization.

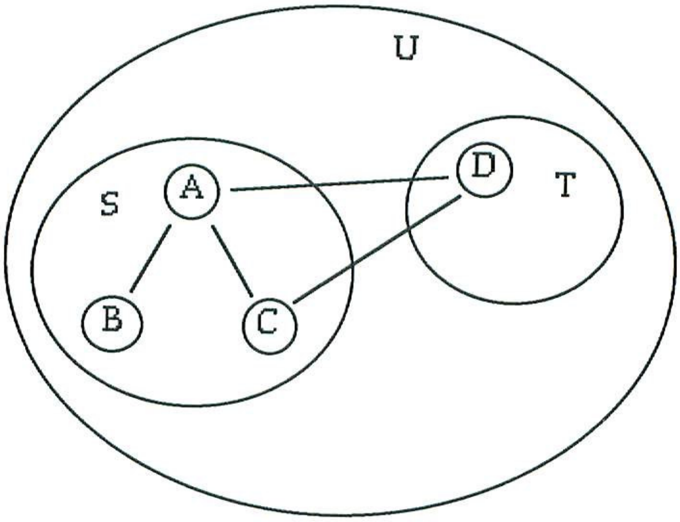


We express this arrangement by the following notation:

```
system U
end;

system S of U
end;

system T of U
end;

interface A of S
import :B :C
export ...
end;

interface B of S
export ...
end;

interface C of S
export ...
end;

interface D of T
import :A :C
export ...
end;
```

A new interface E could join system T just by claiming to belong to it.

Closed systems



Closing systems `U` and `S`:

```
system U
export :A of S
end;
```

```
system S of U
export :A :B
end;
```

```
interface D of T
import :A
export ...
end;
```

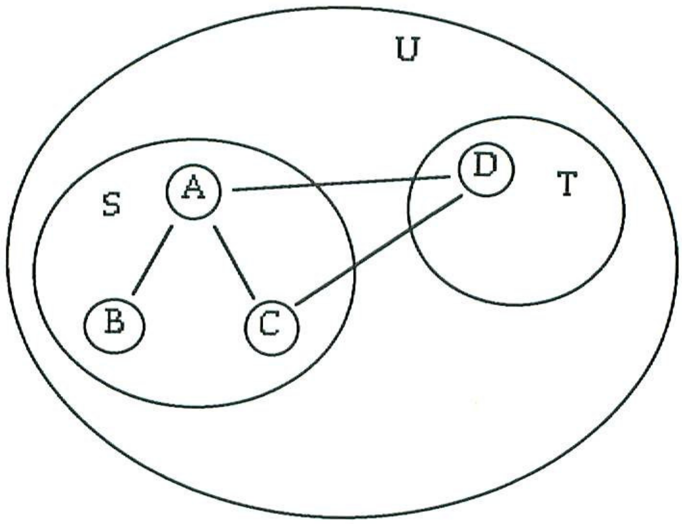
Prevents D from importing C.

D could counteract by claiming to belong to S too, thereby being able again to import C

```
interface D of T,S
import :A :C
export ...
end;
```

Joining a system one explicitly declares the intention of depending on its internal structure, while simply importing an interface provided by a system declares the intention of not depending on any implementation details of that system.

Note that we still have a single name space for interfaces and modules. This is a doubtful feature, but this way interfaces and modules can be moved from one system to another without having to modify all their clients.



Sealing system S:

```
system S of U
components a:A b:B :C
export :A :C
end;
```

Now D is again cut out of S and prevented access to B, although D could be added to the component list of S if desired.

The process of closing a system may reveal unintentional dependencies that may have accumulated during development. The process of sealing a system may reveal deficiencies in the system interface that have to be fixed.

It is expected that, during its evolution, a software system will start as open to facilitate initial development. Then it will be closed when relatively stable interfaces have been developed and the system is ready to be released to clients.

However, at this stage developers may still want to have easy access to the closed system, and they can do so by joining it. When the system is finally quite stable it can be sealed, effectively forming a large, structured abstract type, for example an operating system or file system interface.

System Programming

Low-level programming

As we mentioned in the introduction, a language cannot be considered "real" unless it allows some form of low-level programming; for example a "real" language should be able to express its own compiler, including its own run-time system (memory allocation, garbage collection, etc.).

Most interesting systems, at some point, must get down to the bit level. One can always arrange that these parts of the system are written in some other language, but this is unsatisfactory.

A better solution is to allow these low-level parts to be written in some variation (a subset with some special extensions) of the language at hand, and be clearly marked as such.

One can find solutions that are relatively or completely implementation-independent, that provide good checking, and that localize responsibility when unpredicted behavior results. Some such mechanisms are considered here.

Explicity polymorphic typing turns out to be handy in expressing some of these features.

Dynamic types

Static typechecking cannot cover all situations. One problem is with giving a type to an eval function, or to a generic print function.

A more common problem is handling in a type-sound way data that lives longer than any activation of the compiler.

These problems can be solved by introducing a (static) type of dynamically typechecked data.

Objects of type Dynamic_T should be imagined as pairs of a type and an object of that type. The type component of a Dynamic_T object can be tested at run-time.

One can construct dynamic objects as follows:

```
let d3:Dynamic_T = dynamic.new(:Int 3);
```

These objects can then be *narrowed* to a given type:

```
dynamic.be(:Int d3);
  3: Int

dynamic.be(:Bool d3);
  Exception: dynamicError
```

The matching rules for narrowing and inspecting are the same as for static typechecking, except that the check happens at run-time.

Since an object of type dynamic is self-describing it can be saved to a file and then read back and narrowed, maybe in a separate programming session:

```
let wr = writer.file("d3.dyn");
dynamic.extern(wr d3);      (Write d3 to file)
writer.close(wr);
...
let rd = reader.file("d3.dyn");
let d3 = dynamic.intern(rd); (Read d3)
```

The operations extern and intern preserve sharing and circularities within a single dynamic object, but not across different objects. All values can be made into dynamics, including functions and dynamics. All dynamic values can be externed, except readers and writers; in general it is not meaningful to extern objects that are bound to input/output devices.

Type violations

Most system programming languages allow arbitrary type violations, some indiscriminately, some only in restricted parts of a program. Operations that involve type violations are called unsound.

Type violations fall in several classes:

Basic-value coercions.

Bit and word operations.

Address arithmetic.

Memory mapping.

Metalevel operations.

Unsound features

Here is the Quest mechanism for type violations:

```
unsound interface Value
export
  T::TYPE
    (An arbitrary value)
  error:Exception(Ok)
    (Raised when an operation cannot be carried out)
  new(A::TYPE a:A):T
    (Convert anything to a value)
  be(A::TYPE v:T):A
    (Convert a value to anything.)
  fetch(addr:T displ:Int):T
    (Fetch the value at location addr+displ in memory)
  store(addr:T displ:Int w:T):Ok
    (Store a value at location addr+displ in memory)
end;
```

Whatever the type violation mechanisms are, they need to be controlled somehow, lest they compromise the reliability of the entire language. Hence the keyword "unsound".

Following Cedar-Mesa and Modula3:

Operations that may violate run-time system invariants are called unsound. Unsound operations can only be used in modules that are explicitly declared unsound. If a module is declared sound, the compiler checks that (a) its body contains no unsound operations, and (b) it imports no unsound interfaces.

Unsound modules may advertise an unsound interface. However, unsound modules can also have ordinary interfaces. In the latter case, the programmer of the unsound module guarantees (i.e. proves or, more likely, trusts) that the module is not actually unsound, from an external point of view, although internally it uses unsound operations.

This way, low-level facilities can be added to the system without requiring all the users of such facilities to declare their modules unsound just because they import them.

The main advantage of this scheme is that if something goes very wrong the responsibility can be restricted to unsound modules.

Conclusions

In order to manage large and complex systems, many programming paradigms show convergence in at least one area: typing.

Vice versa, type theory is evolving to cover the typing needs of diverse programming styles and methodologies.

The result is *typeful programming*, a combination of language features, programming methodologies, and formal system that is relatively independent of the control-flow paradigms of the underlying language.

Comparison with other programming styles;

- Typeless programming
- Type-free programming
- Functional programming
- Imperative programming
- Object-oriented programming
- Relational programming
- Algebraic programming
- Concurrent programming
- Programming in the large
- System programming
- Database programming

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

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

Issues of predicativity, impredicativity and stratifications guide the design. Impredicativity leads to more flexibility. Stratification helps in distinguishing compile-time and run-time phases, and in introducing updatable state.

Active research areas:

Models of subtyping
(+ quantifiers, recursion, recursive types).

Meaning of subsumption and coercions,
translation to a subsumption-free calculus.

Typechecking techniques
(reduction, matching, inference).

Modelling of objects and classes.

Module systems.