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
  Typeful programming
  Relevant concepts
  Theory and practice
  Why types?
  Why subtypes?
  Why polymorphism?

Typed λ-calculi and universe levels
  Notation(s)
  Levels
    One type
    Many types
    One kind
    Many kinds
    Kinds are types
    Types are values
  Universes/Predicativity

The Quest language; fundamentals
  Goals/Approach
  Why a new language?
  Language Overview
  Language levels
  Binary vs n-ary quantifiers
  Generalized correspondence principle

The Quest language; constructions
  Simple values
  Simple declarations
  Function declarations
  Recursive declarations
  Tuples
  Tuples and functions
  Type declarations
  Type operators
  Polymorphic functions
  Polymorphic lists
Abstract types
Information hiding
Parametric abstract types

Formal systems
General principles
First-order typed λ-calculus
  Types, terms
  Free variables, substitution
  Environments, judgements
  General rules
  Specific typing rules
  Specific subtyping rules
  Specific computation rules

Quest core formal system
Syntax
Judgements
Notation
  Congruence, extension, subsumption
Signatures, bindings
Kinds, Types, Operators
Values
SubSignatures, SubKinds, SubTypes

Quest typing
  Function types
  Parametric polymorphism
Type operators
Simple tuples
Abstract types
Polymorphism + abstract types

Other Quest design topics
  Quantifier closures
  Quantifiers: pick your own
  Phase distinctions
Subtyping
  Abstract types and the dot notation
Alpha conversion
Compilation techniques
Syntax
Quest subtyping
  Power types
  Records
  Higher-order subtypes
  The subsumption rule
  Subtyping + polymorphism
  Subtyping + abstract types
  Set types
  Subtyping + set types
  Classes and methods

Modules and interfaces
  Programming in the large
  Modules and interfaces
  Separate compilation and linking
  Diamond import

Module combination
  System modelling
  Major approaches
  A different approach
    Open systems
    Closed systems
    Sealed systems

System Programming
  Low-level programming
  Dynamic types
  Type violations
  Unsound features

Conclusions
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 $\lambda$-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 $\lambda$-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.
Why subtypes? (a methodological view)

To aid in the extension 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)

<table>
<thead>
<tr>
<th>construct</th>
<th>introduction</th>
<th>elimination (may fail)</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>constant</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>fun(x) b</td>
<td>b(a)</td>
</tr>
<tr>
<td>tuple</td>
<td>tuple l₁=a₁, ..., lₙ=aₙ end</td>
<td>b.l</td>
</tr>
</tbody>
</table>

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*:

\[
(f\text{un}(x) \ \text{tuple} \ \text{fst}=x \ \text{snd}=x \ \text{end}) \ (3) \\
(f\text{un}(x) \ \text{tuple} \ \text{fst}=x \ \text{snd}=x \ \text{end}) \ (\text{true})
\]

preserved by *parametric polymorphism*.

\[
(f\text{un}(x) \ x.t) \ (\text{tuple} \ t=3 \ \text{end}) \\
(f\text{un}(x) \ x.t) \ (\text{tuple} \ t=3, \ u=\text{true} \ \text{end})
\]

preserved by *subtype polymorphism*.
Typed $\lambda$-calculi and universe levels
### Notation(s)

<table>
<thead>
<tr>
<th>Programming</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{fun}(x:A) \ x )</td>
<td>( \lambda x:A. \ x )</td>
<td>( \lambda x \in A. \ x )</td>
</tr>
<tr>
<td>( f(a) )</td>
<td>( (f \ a) )</td>
<td>( f(a) )</td>
</tr>
<tr>
<td><strong>Polymorphic Func.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{fun}(A) \ \text{fun}(x:A) \ x )</td>
<td>( \lambda A. \ \lambda x:A. \ x )</td>
<td>( \lambda A \in \text{Type}. \ b )</td>
</tr>
<tr>
<td>( \text{fun}(A::\text{TYPE}) \ b )</td>
<td>( \lambda A::\text{Type}. \ b )</td>
<td>( f(A) )</td>
</tr>
<tr>
<td>( f[A] )</td>
<td>( f[A] )</td>
<td></td>
</tr>
<tr>
<td>( f(:A) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuples</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 3, 'x' )</td>
<td>( \langle 3, 'x' \rangle )</td>
<td>( \langle 3, 'x' \rangle )</td>
</tr>
<tr>
<td>( \text{tuple} \ a=3, \ b='x' \ \text{end} )</td>
<td>( \text{fst}(t) \ \text{snd}(t) )</td>
<td>( \text{let} \ \langle x, y \rangle = t )</td>
</tr>
<tr>
<td>( \text{let} \ (x, y) = t )</td>
<td></td>
<td>( \pi_1(t) )</td>
</tr>
<tr>
<td>( t.a )</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Function Spaces</strong></td>
<td>( A \rightarrow B )</td>
<td>( A \rightarrow B )</td>
</tr>
<tr>
<td>( \text{All}(x:A) B )</td>
<td>( A \supset B )</td>
<td>( A \rightarrow B )</td>
</tr>
<tr>
<td><strong>Cartesian Prod.</strong></td>
<td>( A # B )</td>
<td>( A \times B )</td>
</tr>
<tr>
<td>( \text{Tuple} \ f:A, \ s:B \ \text{end} )</td>
<td>( A \land B )</td>
<td>( A \times B )</td>
</tr>
<tr>
<td><strong>Disjoint Union</strong></td>
<td>( A + B )</td>
<td>( A + B )</td>
</tr>
<tr>
<td>( \text{Variant} \ a:A, \ b:B \ \text{end} )</td>
<td>( A + B )</td>
<td>( A + B )</td>
</tr>
<tr>
<td><strong>Universal Quant.</strong></td>
<td>( \text{All}[A] \ A \rightarrow A )</td>
<td>( \forall A. \ A \rightarrow A )</td>
</tr>
<tr>
<td>( \text{All} (A) \ A \rightarrow A )</td>
<td></td>
<td>( \Pi A. \ A \rightarrow A )</td>
</tr>
<tr>
<td><strong>Existential Quant.</strong></td>
<td>( \text{Some}[A] \ A # (A \rightarrow B) )</td>
<td>( \exists A. \ A \times (A \rightarrow B) )</td>
</tr>
<tr>
<td>( \text{Tuple} \ A, \ f:A \rightarrow B \ \text{end} )</td>
<td>( \Sigma A. \ A \times (A \rightarrow B) )</td>
<td></td>
</tr>
</tbody>
</table>
Levels
(values, types and operators, kinds)

One type
(untyped \(\lambda\)-calculus)

\[ \text{let id} = \text{fun}(x) \times \]

id(3)
Many types
(first-order typed $\lambda$-calculus)

\[
\begin{center}
\begin{tikzcd}
& T \arrow{dr} \\
V &
\end{tikzcd}
\end{center}
\]

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

id(3)
```
One kind
(second-order typed $\lambda$-calculus)
(second-order polymorphic $\lambda$-calculus)

\[
\begin{align*}
\text{let } \text{id} : & \text{All}(A) \ A \to A = \\
& \text{fun}(A) \ \text{fun}(x:A) \times \\
\text{id}(:\text{Int})(3)
\end{align*}
\]
Many kinds
(higher-order typed λ-calculus)

Let \( \text{Endo} :: \text{TYPE} \rightarrow \text{TYPE} = \)
\( \text{All}(A :: \text{TYPE}) \ A \rightarrow A \)

let \( \text{id} : \text{All}(A :: \text{TYPE}) \ \text{Endo}(A) = \)
\( \text{fun}(A :: \text{TYPE}) \ \text{fun}(x :: A)\ x \)

\( \text{id}(: \text{Int})(3) \)
Kinds are types
(higher-order typed $\lambda$-calculus with Type:Type)

\[
\begin{array}{c}
\text{K,T} \\
\text{v}
\end{array}
\]

let Endo : Type $\rightarrow$ Type =
fun(A:Type) A $\rightarrow$ A

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

id(:Int)(3)

Problem: *static levelchecking* becomes impossible.

\[
\text{fun(A:Type) fun(x:A) x} \quad \text{is A a type or a kind?}
\]

\[
\text{is x a type or a value?}
\]

id (:Int) (3)     id (:Type) (:Int)
comp \uparrow run     comp \uparrow run
Types are values
(untyped \(\lambda\)-calculus with Type:Value)

\[ K, T, V \]

```ml
let Endo = fun(A) A->A

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

id(Int)(3)
```

Problem: static typechecking becomes impossible.

```ml
id (Int) (true) id (3) (4)
```
Universes

The predicative hierarchy

Values 3 ∈ U0
Types Int, Int→Int, (Int→Int)→Int ∈ U1
Polytypes All(A::U1)A→A
Operators U1→U1 ∈ U2
Higher operators U2→U2 ∈ U3 ...

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

The impredicative hierarchy

Values 3 ∈ Value
Types Int, Int→Int, (Int→Int)→Int ∈ Type
Polytypes All(A::TYPE)A→A ∈ Kind
Operators Type→Type
Higher operators (Type→Type)→Type

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

KINDS

Level 2

TYPE

ALL(A::TYPE)TYPE

POWER(B)

Types

Int

All(A::TYPE)All(x:A)A

Pair A::TYPE x:A end

Fun(A::TYPE)All(x:A)A

Operators

A <: B

values

Level 0

3

fun(A::TYPE)fun(x:A)x

pair Let A=Int let x=3 end

a
- **All(x:A)B**  \( \text{(e.g.:} \quad \text{fun(x:} \text{Int})x \) \)

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**  \( \text{(e.g.:} \quad \text{fun(A::TYPE})\text{fun(x:A)}x \) \)

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**  \( \text{(e.g.:} \quad \text{Fun(A::TYPE})A \) \)

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**  \( \text{(e.g.:} \quad \text{pair let x=true let y=3 end} \) \)

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**  \( \text{(e.g.:} \quad \text{pair Let X=Int let y:X=3 end} \) \)

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

\[ g : \text{All}(A::\text{TYPE}) \text{ All}(a:A) \text{ All}(f:A\rightarrow\text{Int}) \text{ Int} \]

\[ = \text{fun}(A::\text{TYPE}) \text{ fun}(a:A) \text{ fun}(f: A\rightarrow\text{Int}) f(a) \]

\[ g(:\text{Int})(3)(\text{succ}) \]

\textit{vs.}

\[ g : \text{All}(A::\text{TYPE} \ a:A \ f:A\rightarrow\text{Int}) \text{ Int} \]

\[ = \text{fun}(A::\text{TYPE} \ a:A \ f:A\rightarrow\text{Int}) f(a) \]

\[ g(:\text{Int} \ 3 \ \text{succ}) \]
t: \textit{Pair}(A::\textsc{TYPE}) \textit{Pair}(a:A) A\to\textsc{Int}

= \textit{pair}(A::\textsc{TYPE}=\textsc{Int}) \textit{pair}(a:A=3) \textsc{succ} \text{ end end}

\textit{i.e.}: t = \textit{pair} :\textsc{Int} \textit{pair} 3 \textsc{succ} \text{ end end}

\textsc{snd}(\textsc{snd}(t))(\textsc{fst}(\textsc{snd}(t))): \textsc{Int}

\textsc{fst}(\textsc{snd}(t)) : \textsc{Fst}(t)

\textit{vs.}

t: \textit{Tuple} A::\textsc{TYPE} a:A f:A\to\textsc{Int} \text{ end}

= \textit{tuple}

Let A::\textsc{TYPE} =\textsc{Int} \ \text{let} \ a:A = 3 \ \text{let} \ f:A\to\textsc{Int} = \textsc{succ} \text{ end}

\textit{i.e.}: t = \textit{tuple} :\textsc{Int} 3 \textsc{succ} \text{ end}

t.f(t.a) : \textsc{Int}

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

$$A::\text{TYPE} \ a:A \ f:A\rightarrow\text{Int}$$

*n-ary universals:* \hspace{1cm} $\text{All}(S) \ A$

*n-ary existentials:* \hspace{1cm} $\text{Tuple} \ S \ \text{end}$

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

$$\text{Let} \ A::\text{TYPE} = \text{Int} \ \text{let} \ a:A = 3 \ \text{let} \ f:A\rightarrow\text{Int} = \text{succ}$$

*n-ary application* \hspace{1cm} $f(D)$

*n-ary products* \hspace{1cm} $\text{tuple} \ D \ \text{end}$
Generalized correspondence principle
(Burstall-Lampson)

\[
\text{let } f(S_1)\ldots(S_n):B = b \quad \Rightarrow \quad \text{let } f = \text{fun}(S_1) \ldots \text{fun}(S_n):B \ b
\]

\[
f(S_1)\ldots(S_n):B \quad \Rightarrow \quad f: \text{All}(S_1) \ldots \text{All}(S_n):B
\]

Signatures in:

Declarations

Let \( A::\text{TYPE} = \text{Int} \) let \( a:A = 3 \) let \( f(x:A):\text{Int} = x+1 \)

Formal parameters

let \( f(A::\text{TYPE} \ a:A \ f(x:A):\text{Int}):A = ... \)

Types

\[
\text{All}(A::\text{TYPE} \ a:A \ f(x:A):\text{Int}) A
\]

Tuple \( A::\text{TYPE} \ a:A \ f(x:A):\text{Int} \) end

Interfaces

interface I import ...
export
\[
A::\text{TYPE}
\]
\[
a:A
\]
\[
f(x:A):\text{Int}
\]
end
 Bindings in:

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

\[
\text{Let } A::\text{TYPE} = \text{Int} \quad \text{let } a:A = 3 \quad \text{let } f(x:A):\text{Int} = x+1 \\
:\text{Int} \quad 3 \quad \text{fun}(x:\text{Int})x+1
\]

**Actual parameters**

\[
f(\text{Let } A::\text{TYPE} = \text{Int} \quad \text{let } a:A = 3 \quad \text{let } f(x:A):\text{Int} = x+1) \\
f(:\text{Int} \quad 3 \quad \text{fun}(x:\text{Int})x+1)
\]

**Tuples**

\[
\text{tuple} \\
\text{Let } A::\text{TYPE} = \text{Int} \quad \text{let } a:A = 3 \quad \text{let } f(x:A):\text{Int} = x+1 \\
\text{end} \\
\text{tuple} \quad :\text{Int} \quad 3 \quad \text{fun}(x:\text{Int})x+1 \quad \text{end}
\]

**Modules**

\[
\text{module } m::I \text{ import } ... \\
\text{export} \\
\text{Let } A::\text{TYPE} = \text{Int} \\
\text{let } a:A = 3 \\
\text{let } f(x:A):\text{Int} = x+1 \\
\text{end}
\]
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                                 simultaneous declarations
and b = 5;

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

The successor function

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

A function of no arguments

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

A function of two arguments

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

A curried function

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

Partial application

```haskell
let it = twice(succ);
it(3);
  = 5
```
Recursive declarations

Recursive functions

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

Mutually recursive functions

```plaintext
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

```plaintext
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

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

A legal application

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

A tuple with function components

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

Selection and application

```plaintext
q.succ(3);
```
Type declarations

The Ok type
\[
\begin{align*}
\text{let } \text{Ok}:\text{TYPE} &= \text{Tuple end}; \\
\text{let } \text{ok}:\text{Ok} &= \text{tuple end};
\end{align*}
\]

An integer pair type
\[
\begin{align*}
\text{let } \text{IntPair}:\text{TYPE} &= \\
&\quad \text{Tuple fst:Int snd:Int end};
\end{align*}
\]

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

The integer function type
\[
\begin{align*}
\text{let } \text{IntFun}:\text{TYPE} &= \text{All}(\text{:Int})\text{Int}; \\
\text{let } \text{f}:\text{IntFun} &= \text{succ};
\end{align*}
\]
Type operators

Cartesian product

Let $\#(A,B:\text{TYPE})::\text{TYPE} =$
  Tuple fst:A snd:B end;

Function space

Let $\rightarrow(A,B::\text{TYPE})::\text{TYPE} =$
  All(:A) B;

Ex.: \[(\text{Int } \# \text{ Int} ) \rightarrow \text{Int} :: \text{TYPE}\]

Homogeneous lists

Let List(A::\text{TYPE})::\text{TYPE} =$
  Rec (B::\text{TYPE})$
    Option
      nil
      cons with hd:A tl:B end
  end;

Ex.: List(\text{Int}) :: \text{TYPE}
Polymorphic functions

The type of the integer identity

\[
\texttt{let } \text{IntId::TYPE } = \\
\text{Int } \rightarrow \text{Int};
\]

The integer identity

\[
\texttt{let intId(a: Int)::Int } = \\
a
\]

Usage of integer identity

\[
\text{intId(3)};
\]

The type of the polymorphic identity

\[
\texttt{let Id::TYPE } = \\
\text{All(A::TYPE) A } \rightarrow \text{A};
\]

The polymorphic identity

\[
\texttt{let id(A::TYPE)(a:A)::A } = a;
\]

.. application of a polymorphic function

\[
\text{id(\text{\text{:Int})(3)});
\]

.. abbreviated application

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

Specialized identities

\[
\texttt{let intId: Int--\rightarrow Int } = \text{id(\text{\text{:Int)});}
\]

\[
\texttt{let boolId: Bool--\rightarrow Bool } = \text{id(\text{\text{:Bool)};}
\]
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
  obj:T
  op:T->Int
end;

An algebra (implementation)

let alg1 =
  tuple
  Let T::TYPE = Int
  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 $\text{alg1}$ can be defined totally independently of $\text{Alg}$. 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).

\begin{verbatim}
Let $\text{Alg}'::\text{TYPE} =$
    $\text{Tuple}$
    $T::\text{TYPE} \ \text{obj}:T \ \text{op}:T\to\text{Int}$
end;
\end{verbatim}

$\text{Alg}'$ matches $\text{Alg}$ Hence $\text{alg1:Alg}'$, although $\text{Alg}'$ was defined after $\text{alg1}$ was created. $\text{Alg}'$ can impersonate $\text{Alg}$. 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)

\( \Pi(X::K)L\{X\} \) is a kind (functions from types to types)
Kinds are closed under quantification over kinds.

\( \Pi(x:A)L\{x\} \) is a kind (functions from values to types)
Kinds are closed under quantification over types.

\( \Pi(X::K)B\{X\} \) is a type (functions from types to values)
Types are closed under quantification over kinds.

\( \Pi(x:A)B\{x\} \) is a type (functions from values to values)
Types are closed under quantification over types.

(Similarly for \( \Sigma \).)
Quantifiers: pick your own

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

<table>
<thead>
<tr>
<th>Quest</th>
<th>ELF</th>
<th>Fo</th>
<th>Constructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL(X::K) L</td>
<td>-</td>
<td>K⇒L</td>
<td>Π(X::K) L</td>
</tr>
<tr>
<td></td>
<td>Π(x:A) L</td>
<td>-</td>
<td>Π(x:A) L</td>
</tr>
<tr>
<td>All(X::K) B</td>
<td>-</td>
<td>A∀(X::K) B</td>
<td>Π(X::K) B</td>
</tr>
<tr>
<td>A → B</td>
<td>Π(x:A) B</td>
<td>A → B</td>
<td>Π(x:A) B</td>
</tr>
</tbody>
</table>

derivable:

<table>
<thead>
<tr>
<th>(could add)</th>
<th>-</th>
<th>-</th>
<th>Σ(X::K) L</th>
<th>a kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair(X::K) B</td>
<td>-</td>
<td>-</td>
<td>Σ(X::K) B</td>
<td>a type</td>
</tr>
<tr>
<td>A × B</td>
<td>-</td>
<td>-</td>
<td>Σ(x:A) B</td>
<td>a type</td>
</tr>
</tbody>
</table>
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 \[ E \vdash a : A \] (a,A are symbolic expressions)

\[ \downarrow \]

translation

2nd \[ E \vdash a = b : A \] (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:

\[
\text{(fun}(A::\text{TYPE}) \text{ fun}(a:A) a) (\text{Int}) (3) \quad \text{(1st 2nd)}
\]

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

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

\[
\begin{align*}
\Pi(x:A)B & \quad \text{has } B\{x \leftarrow a\} \text{ in the elim rule (keep } A \rightarrow B\text{).} \\
\Pi(x:A)L & \quad \text{has } L\{x \leftarrow a\} \text{ in the elim rule.} \\
 & \quad \quad \text{(}A \rightarrow L \text{ has } B\{x \leftarrow a\} \text{ in reduction rule.)}
\end{align*}
\]

\[
\begin{align*}
\Sigma(x:A)B & \quad \text{has } B\{x \leftarrow \text{Ift}(a)\} \text{ in the elim rule (keep } A \times B\text{).} \\
\Sigma(x:A)L & \quad \text{has } L\{x \leftarrow \text{Ift}(a)\} \text{ in the elim rule.} \\
 & \quad \quad \text{(could keep } A \times L, \text{ but it is isom. to } L \times A.)
\end{align*}
\]

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 lose type information.

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

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

t : Pair(A <: Int) A x (Int -> A) x (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 \[ A :: \text{POWER}(B) \]

\[ \text{All}(A <: B) \ C \] abbreviates \[ \text{All}(A :: \text{POWER}(B)) \ C \]
(i.e. we no longer need special bounded quantifiers)
From "bind (x₁...xₙ)=a in .. xᵢ .." to "a.xᵢ"

The notation "bind (x₁...xₙ) = a in b" is too sensitive to changes in the type of "a" (when software is evolving).

Also, simple selectors like "bind (x₁...xₙ) = a in xᵢ" are very cumbersome to write.

Programming languages universally use the dot notation "a.xᵢ" (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.xᵢ" in an adequate (i.e. large enough) context of the form:

\[
\text{let } y = c \text{ in } \ldots \ y.xᵢ \ \ldots \ y.x_j \ \ldots \ \text{end}
\]

which can then be interpreted as equivalent to:

\[
\text{bind } x₁...xₙ = c \text{ in } \ldots \ xᵢ \ \ldots \ x_j \ \ldots \ \text{end}
\]

i.e., the type rules for "a.xᵢ" 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 $\alpha$-conversion

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

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

which can then be transformed into:

\[
\text{bind } x_1...x_n = c \text{ in } ... \text{ x$_i$} ... \text{ x$_j$} ... \text{ end}
\]

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

I.e. $\alpha$-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 \(\alpha\)-conversion. I.e. while it is clear that (with the usual free-variable restrictions):

\[
\text{Some}(x:A) \ B[x] \ \leftrightarrow_{\alpha} \ \text{Some}(y:A) \ B[y]
\]

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

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

\[
\begin{align*}
\text{let } f &= \text{fun}(t: \text{Tuple } x: \text{Int } y: \text{Int } \text{end}) t.x + t.y \\
\text{let } a &= \text{tuple } \text{let } y=3 \ \text{let } x=4 \ \text{end}; \\
\text{f(a)}
\end{align*}
\]

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 \(\alpha\)-convert. Any such matching up to \(\alpha\)-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 $\alpha$-conversion.

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

We however adopt a weaker form of $\alpha$-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] ["elseif" 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 elseif 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) $\lambda$-calculi can be described as formal systems based on judgements:

$$E \vdash \Phi$$

Where $E$ is a list of assumptions for variables free in $\Phi$, and $\Phi$ 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 \textit{relation} $\vdash$ (entailment) between assumptions and conclusions, which determines the \textit{valid} conclusions.

This relation is normally defined inductively by \textit{axioms} and \textit{inference rules}:

\[
E_0 \vdash \Phi_0
\]

\[
E_1 \vdash \Phi_1 \quad \ldots \quad E_n \vdash \Phi_n
\]

\[
E_{n+1} \vdash \Phi_{n+1}
\]

(and nothing else is in the $\vdash$ relation)
First-order typed $\lambda$-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)
\[ \text{Rec}(X) A \]  recursive types

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

\[ x \]
\[ k_{ij} \]
\[ \text{fun}(x:A) b \]  \[ b(a) \]

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

Free variables  (omitted)

Substitution  (omitted)
Environments \((E)\)

\[\emptyset\]
\[E, \, x:A\]

**extensions**

\[E, \, X \text{ type} \quad \text{for recursion}\]
\[E, \, x:A \quad \text{for subtyping}\]

**Judgements**

\[\vdash E \text{ env} \quad E \text{ is a well-formed environment}\]
\[E \vdash A \text{ type} \quad A \text{ is a legal type (trivial)}\]
\[E \vdash a:A \quad a \text{ has type } A\]
\[E \vdash A \leftrightarrow B \quad A \text{ and } B \text{ are equivalent types (trivial)}\]
\[E \vdash a \leftrightarrow b:A \quad a \text{ and } b \text{ equivalent terms of type } A\]

**extensions**

\[E \vdash A <: B \quad A \text{ is a subtype of } B\]
\[E \vdash C \downarrow X \quad C \text{ is contractive in } X\]
General rules

Pure calculus

Environments
\[ \vdash \emptyset \text{ env} \]
\[ \vdash E \text{, } x:A \text{ env} \]

Type congruence
(\(\leftrightarrow\) is a substitutive equivalence relation over well-formed types)
\[ E \vdash A \leftrightarrow B \]
\[ E \vdash B \leftrightarrow A \]
\[ E \vdash \text{K}_i \text{ type} \]
\[ E \vdash \text{K}_i \leftrightarrow \text{K}_i \]

Congruence
(\(\leftrightarrow\) is a substitutive equivalence relation over the syntax of terms)
(omitted)
Recursion extensions

Environments

\[\vdash E \text{ env} \quad X \notin \text{Dom}(E)\]
\[\vdash E, X \text{ type env}\]

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

\[E, X \text{ type } \vdash K \text{ type}\]
\[E \vdash K \downarrow X\]
\[E, X \text{ type } \vdash Y \text{ type} \quad Y \neq X\]
\[E \vdash Y \downarrow X\]
\[E, X \text{ type } \vdash A \text{ type} \quad E, X \text{ type } \vdash B \text{ type}\]
\[E \vdash (A \rightarrow B) \downarrow X\]
\[E, Y \text{ type } \vdash A \downarrow X \quad E, X \text{ type } \vdash A \downarrow Y \quad Y \neq X\]
\[E \vdash (\text{Rec}(Y)A) \downarrow X\]

Type Equivalence

\[E \vdash \text{Rec}(X)A \quad Y \notin \text{Dom}(E)\]
\[E \vdash \text{Rec}(X)A \leftrightarrow \text{Rec}(Y)A[X \leftrightarrow Y]\]
\[E \vdash \text{Rec}(X)A \quad Y \notin \text{Dom}(E)\]
\[E \vdash \text{Rec}(X)A \leftrightarrow A[X \leftrightarrow \text{Rec}(X)A]\]
\[E \vdash A \leftrightarrow C[X \leftrightarrow A]\]
\[E \vdash B \leftrightarrow C[X \leftrightarrow B]\]
\[E \vdash C \downarrow X\]
\[E \vdash A \leftrightarrow B\]

Retyping

\[E \vdash a:A \quad E \vdash a:A \leftrightarrow B\]
\[E \vdash a:B\]
Subtyping extensions

Environments

\[ E \vdash A \text{ type} \quad X \notin \text{Dom}(E) \]
\[ \vdash E, X <: A \quad \text{env} \]

Reflexivity

\[ E \vdash A \leftrightarrow B \]
\[ E \vdash A <: B \]

Transitivity

\[ E \vdash A <: B \quad E \vdash B <: C \]
\[ E \vdash A <: C \]

Subsumption

\[ E \vdash a : A \quad E \vdash A <: B \]
\[ E \vdash a : B \]

(Retyping is now derivable)
## Specific typing rules

### Formation

<table>
<thead>
<tr>
<th>E type</th>
<th>A type</th>
<th>B type</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ E env</td>
<td>⊢ E env</td>
<td>⊢ E env</td>
</tr>
<tr>
<td>E, X type ⊢ X type</td>
<td>E, x:A ⊢ x:A</td>
<td>E ⊢ k_{i}; K_{i}</td>
</tr>
</tbody>
</table>

### Introduction

<table>
<thead>
<tr>
<th>E type</th>
<th>A type</th>
<th>B type</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ A type</td>
<td>⊢ B type</td>
<td>⊢ b:A→B</td>
</tr>
<tr>
<td>⊢ A→B type</td>
<td>⊢ fun(x:A) b : A→B</td>
<td>⊢ b(a) : B</td>
</tr>
</tbody>
</table>

### Elimination

<table>
<thead>
<tr>
<th>E type</th>
<th>A type</th>
<th>B type</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ b:A→B</td>
<td>⊢ a:A</td>
<td></td>
</tr>
</tbody>
</table>

### Extensions

<table>
<thead>
<tr>
<th>E type</th>
<th>A type</th>
<th>B type</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ A type</td>
<td>⊢ B type</td>
<td>⊢ c:A×B</td>
</tr>
<tr>
<td>⊢ A×B type</td>
<td>⊢ &lt;a,b&gt; : A×B</td>
<td>⊢ lft(c):A</td>
</tr>
<tr>
<td>⊢ A_{i} type</td>
<td>⊢ a:A_{i}</td>
<td>⊢ rht(c):B</td>
</tr>
<tr>
<td>⊢ (l_{i}; A_{i}) type</td>
<td>⊢ (l_{i}=a_{i}) : (l_{i}; A_{i})</td>
<td>⊢ c:(l_{i}; A_{i})</td>
</tr>
<tr>
<td>⊢ A_{i} type</td>
<td>⊢ a:A_{i}</td>
<td>⊢ c:[l_{i}; A_{i}]</td>
</tr>
<tr>
<td>⊢ [l_{i}; A_{i}] type</td>
<td>⊢ [l_{i}=a_{i}] as [l_{i}; A_{i}] : [l_{i}; A_{i}]</td>
<td>⊢ x_{i}:A_{i} ⊢ b_{i}:C</td>
</tr>
<tr>
<td>⊢ A↓X</td>
<td>E, x:A ⊢ a:A</td>
<td>⊢ case c : [l_{i}=x_{i}] b_{i} : C</td>
</tr>
<tr>
<td>⊢ Rec(X) A type</td>
<td>⊢ rec(x:A) a : A</td>
<td></td>
</tr>
</tbody>
</table>
Specific subtyping rules

Subtyping

\[ \Gamma \vdash E \text{ env} \quad \text{for some pairs } i,j \]
\[ E \vdash K_i <: K_j \]
\[ E \vdash A'<: A \quad E \vdash B <: B' \]
\[ E \vdash A \rightarrow B <: A' \rightarrow B' \]

Extensions

\[ E \vdash A <: A' \quad E \vdash B <: B' \]
\[ E \vdash A \times B <: A' \times B' \]
\[ E \vdash A_i <: B_i \quad E \vdash A_j \text{ type} \]
\[ E \vdash \langle i; A, j; A_j \rangle <: \langle i; B, j \rangle \]
\[ E \vdash A_i <: B_i \quad E \vdash B_j \text{ type} \]
\[ E \vdash [i; A_i] <: [i; B_i] \]
\[ E, Y \text{ type}, X <: Y \vdash A <: B \quad X \neq Y \quad X \notin \text{FV}(B) \quad Y \notin \text{FV}(A) \]
\[ E \vdash \text{Rec}(X) A <: \text{Rec}(Y) B \]
Specific computation rules

**Computation**

(omitted)

\[ \text{E, } x:A \vdash b:B \quad y \notin \text{Dom(E)} \quad \text{E, } x:A \vdash b:B \quad \text{E} \vdash a:A \]

\[ \text{E} \vdash \text{fun}(x:A)b \leftrightarrow \text{fun}(y:A)b\{x \leftarrow y\} : B \quad \text{E} \vdash (\text{fun}(x:A)b)(a) \leftrightarrow b\{x \leftarrow a\} : B \]

**Extensions**

(omitted)

(omitted)

(omitted)

(omitted)

\[ \text{E, } x:A \vdash a:A \quad y \notin \text{Dom(E)} \quad \text{E, } x:A \vdash a:A \]

\[ \text{E} \vdash \text{rec}(x:A)a \leftrightarrow \text{rec}(y:A)a\{x \leftarrow y\} : A \quad \text{E} \vdash \text{rec}(x:A)a \leftrightarrow a\{x \leftarrow \text{rec}(x:A)a\} : A \]
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 l₁:A₁, ..., lₙ:Aₙ end
Variant l₁:A₁, ..., lₙ:Aₙ end
Set A₁, ..., Aₙ 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 l₁=a₁, ..., lₙ=aₙ end
a.t
variant l=a end
case a ... when(lᵢ=xᵢ) bᵢ ... end
set a₁, ..., aₙ end
rec(x:A)a

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

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

Formation
\[ \vdash S \text{ sig} \quad \text{S is a signature} \]
\[ S \vdash K \text{ kind} \quad \text{K is a kind} \]
\[ S \vdash A \text{ type} \quad \text{A is a type} \]
\[ \quad \text{(same as } S \vdash A::\text{TYPE}) \]

Equivalence
\[ S \vdash S'::<::>S'' \quad \text{equivalent signatures} \]
\[ S \vdash K::<::>L \quad \text{equivalent kinds} \]
\[ S \vdash A::<::>B \quad \text{equivalent types} \]

Inclusion
\[ S \vdash S'::<::>S'' \quad \text{S' is a subsignature of S''} \]
\[ S \vdash K::<::>L \quad \text{K is a subkind of L} \]
\[ S \vdash A::<::>B \quad \text{A is a subtype of B} \]
\[ \quad \text{(same as } S \vdash A::\text{POWER}(B)) \]

Membership
\[ S \vdash D::<::>S' \quad \text{D has signature S'} \]
\[ S \vdash A::K \quad \text{A has kind K} \]
\[ S \vdash a:A \quad \text{a has type A} \]
Notation

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

Signatures and bindings are ordered, however we freely use the notation $X \in \text{Dom}(S)$ (type $X$ is defined in $S$), $x \in \text{Dom}(S)$ (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 $D$.

$E\{X \leftarrow A\}$ 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 $X \in \text{Dom}(D)$ and $x \in \text{Dom}(D)$ 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

\[ S \vdash S' \leftrightarrow S'' \quad S \vdash K \leftrightarrow L \quad S \vdash A \leftrightarrow B \]

\[ S \vdash S' \leftrightarrow S'' \quad S \vdash K \leftrightarrow L \quad S \vdash A \leftrightarrow B \]

Subsumption

\[ S \vdash D :: S' \quad S \vdash S' \leftrightarrow S'' \quad S \vdash A :: K \quad S \vdash K \leftrightarrow L \quad S \vdash a : A \quad S \vdash A \leftrightarrow B \]

\[ S \vdash D :: S'' \quad S \vdash A :: L \quad S \vdash a : B \]

Conversion

\[ S, X :: K \vdash B :: L \quad S \vdash A :: K \]

\[ S \vdash (\text{Fun}(X :: K)B)(A) \leftrightarrow B[X \leftarrow A] \]
Signatures

\[ \vdash \emptyset \text{ sig} \]
\[ S \vdash K \text{ kind } \quad x \notin \text{Dom}(S) \]
\[ \vdash S, X :: K \text{ sig} \]
\[ S \vdash A \text{ type } \quad x \notin \text{Dom}(S) \]
\[ \vdash S, x :: A \text{ sig} \]

Bindings

\[ \vdash S \text{ sig} \]
\[ S \vdash \emptyset :: \emptyset \]
\[ S \vdash D :: S' \quad S \vdash A[D] :: K[D] \]
\[ S \vdash D, \text{Let } X :: K = A \quad :: S', X :: K \]
\[ S \vdash D :: S' \quad S \vdash a[D] :: A[D] \]
\[ S \vdash D, \text{let } x :: A = a \quad :: S', x :: A \]

Ex.

\[ S \vdash \text{Int} :: \text{TYPE} \]
\[ S \vdash \emptyset :: \emptyset \quad S \vdash \text{Int} \{ \emptyset \} :: \text{TYPE} \{ \emptyset \} \quad S \vdash 3 :: \text{Int} \]
\[ S \vdash \emptyset, \text{Let } A :: \text{TYPE} = \text{Int} \quad :: \emptyset, A :: \text{TYPE} \quad S \vdash 3 \{ \emptyset, \text{Let } A = \text{Int} \} :: A \{ \emptyset, \text{Let } A = \text{Int} \} \]
\[ S \vdash \emptyset, \text{Let } A :: \text{TYPE} = \text{Int}, \text{let } a :: A = 3 \quad :: \emptyset, A :: \text{TYPE}, a :: A \]
Kinds

\[ \vdash S \text{ sig} \]
\[ S \vdash \text{TYPE kind} \]
\[ S \vdash K \text{ kind} \quad S, X::K \vdash L \text{ kind} \]
\[ S \vdash \text{ALL}(X::K)L \text{ kind} \]
\[ S \vdash A \text{ type} \]
\[ S \vdash \text{POWER}(A) \text{ kind} \]

Types and Operators

\[ \vdash S \text{ sig} \quad X \in \text{Dom}(S) \]
\[ S \vdash X :: S(X) \]
\[ SS' \vdash A \text{ type} \]
\[ S \vdash \text{All}(S')A \text{ type} \]
\[ \vdash SS' \text{ sig} \]
\[ S \vdash \text{Tuple } S' \text{ end type} \]
\[ S, X::K \vdash B::L \]
\[ S \vdash \text{Fun}(X::K)B :: \text{ALL}(X::K)L \]
\[ S \vdash B::\text{ALL}(X::K)L \quad S \vdash A::K \]
\[ S \vdash B(A) :: L[X \leftarrow A] \]
\[ S \vdash A_1 \text{ type} \quad S \vdash A_n \text{ type} \]
\[ S \vdash \text{Record } l_1::A_1, ..., l_n::A_n \text{ end type} \]
\[ S \vdash A_1 \text{ type} \quad S \vdash A_n \text{ type} \]
\[ S \vdash \text{Variant } l_1::A_1, ..., l_n::A_n \text{ end type} \]
\[ S \vdash A_1 \text{ type} \quad S \vdash A_n \text{ type} \]

\[ S \vdash \text{Set } A_1, \ldots, A_n \text{ end } \text{ type} \]

\[ S, X::\text{TYPE} \vdash A \text{ type} \]

\[ S \vdash \text{Rec}(X::\text{TYPE}) A \text{ type} \]

**Values**

\[ \vdash S \text{ sig } x \in \text{Dom}(S) \]

\[ S \vdash x : S(x) \]

\[ S S' \vdash a : A \]

\[ S \vdash \text{fun}(S') a : \text{All}(S') A \]

\[ S \vdash a : \text{All}(S') A \quad S \vdash D::S' \]

\[ S \vdash a(D) : A[D] \]

\[ S \vdash D::S' \]

\[ S \vdash \text{tuple } D \text{ end} : \text{Tuple } S' \text{ end} \]

\[ S \vdash a : \text{Tuple } S' \text{ end} \quad S \vdash B \text{ type} \quad S S' \vdash b : B \]

\[ S \vdash \text{bind } S' = a \text{ in } b \text{ end } : B \text{ end} \]

\[ S \vdash a_i : A_1 \quad \ldots \quad S \vdash a_n : A_n \]

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

\[ S \vdash a : \text{Record } t_i : A_1, \ldots, t_n : A_n \text{ end } \quad i \in 1..n \]

\[ S \vdash a.l_i : A_i \]

\[ S \vdash B_1 \text{ type} \quad \ldots \quad S \vdash B_m \text{ type} \quad \forall i \in 1..n \quad \exists j \in 1..m \quad S \vdash a_i : B_j \]

\[ S \vdash \text{set } a_1, \ldots, a_n \text{ end} : \text{Set } B_1, \ldots, B_m \text{ end} \]
S, x:A ⊢ a:A
S ⊢ rec(x:A)a : A
Ex.

\[ S \vdash \text{alg} : \text{Tuple} \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} \; \text{end} \]
\[ S \vdash \text{Int} :: \text{TYPE} \]
\[ S, \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} \vdash f(a) : \text{Int} \]
\[ S \vdash \text{bind} \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} = \text{alg} \; \text{in} \; f(a) \; \text{end} : \; \text{Int} \]

\[ S \vdash \text{alg} : \text{Tuple} \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} \; \text{end} \]
\[ S \vdash \text{Int} :: \text{TYPE} \]
\[ S, \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} \vdash a+1 : \text{Int} \]
\[ S \vdash \text{bind} \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} = \text{alg} \; \text{in} \; a+1 \; \text{end} : \; \text{Int} \]

\[ S \vdash \text{alg} : \text{Tuple} \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} \; \text{end} \]
\[ S \vdash A :: \text{TYPE} \]
\[ S, \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} \vdash a : A \]
\[ S \vdash \text{bind} \; A::\text{TYPE}, \; a:A, \; f:A \rightarrow \text{Int} = \text{alg} \; \text{in} \; a \; \text{end} : \; A \]
SubSignatures

\[ S \vdash S' <: S'' \quad S S' \vdash K <: L \]
\[ S \vdash S', X :: K < :: S'', X :: L \]
\[ S \vdash S' <: S'' \quad S S' \vdash A :: B \]
\[ S \vdash S', x :: A < :: S'', x :: B \]

SubKinds

\[ S \vdash K' <: K \quad S, X :: K' \vdash L <: L' \]
\[ S \vdash \text{ALL}(X :: K)L < :: \text{ALL}(X :: K')L' \]
\[ S \vdash A :: B \]
\[ S \vdash \text{POWER}(A) < :: \text{POWER}(B) \]
\[ S \vdash A \text{ type} \]
\[ S \vdash \text{POWER}(A) < :: \text{TYPE} \]

SubTypes

\[ S \vdash S'' <: S' \quad S, S'' \vdash A' :: A'' \]
\[ S \vdash \text{All}(S')A' < :: \text{All}(S'')A'' \]
\[ \vdash SS'S'' \text{ sig} \quad S \vdash S' <: S'' \]
\[ S \vdash \text{Tuple} S'S'' \text{ end} < :: \text{Tuple} S'' \text{ end} \]
\[ S \vdash A_1 :: B_1 \ldots S \vdash A_n :: B_n \ldots S \vdash A_m \text{ type} \]
\[ S \vdash \text{Record} l_1 :: A_1, \ldots, l_n :: A_n, \ldots, l_m :: A_m \text{ end} < :: \text{Record} l_1 :: B_1, \ldots, l_n :: B_n \text{ end} \]
\[ S \vdash A_1 :: B_1 \ldots S \vdash A_n :: B_n \ldots S \vdash B_m \text{ type} \]
\[ S \vdash \text{Variant} l_1 :: A_1, \ldots, l_n :: A_n \text{ end} < :: \text{Variant} l_1 :: B_1, \ldots, l_n :: B_n, \ldots, l_m :: B_m \text{ end} \]
∀i∈1..n ∃j∈1..m S ⊢ A_i ⇐ B_j
S ⊢ Set A_1, ..., A_n end ⇐ Set B_1, ..., B_m end
Quest typing
Function types

An object of type $\text{All}(x:A)B$ (or simply $A \rightarrow B$) is a function $\text{fun}(x:A)b$ of argument $x:A$ and result $b:B$.

Let $\text{IntId}::\text{TYPE} =$

$\text{All}(x:\text{Int})\ \text{Int};$

let $\text{intId}::\text{IntId} =$

$\text{fun}(x:\text{Int})\ x;$

$\text{intId}(3);$
Parametric polymorphism
(Girard-Reynolds)

An object of type $\text{All}(X:K)B\{X\}$ is a polymorphic function $\text{fun}(X:K)b$ of argument $A:K$ and result $b:B\{X\leftarrow A\}$

Let $Id::\text{TYPE} =$
$$\text{All}(A::\text{TYPE}) \text{All}(x:A) A;$$

let $id::Id =$
$$\text{fun}(A::\text{TYPE}) \text{fun}(x:A) x;$$

$id(:,\text{Int});$
$$= \text{fun}(x:\text{Int}) x = \text{intId}$$
$$: \text{All}(x:\text{Int}) \text{Int} = \text{IntId}$$

$id(\text{Int})(3);$
$$= 3 : \text{Int}$$

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 \texttt{ALL}(X : :K)L\{X\} is an operator \texttt{Fun}(X : :K)b of argument A : :K and result B : :L\{X \leftarrow A\}

\begin{verbatim}
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
\end{verbatim}
Simple tuples

An object of type $\text{Tuple x:A, y:B end}$ is a pair of left component $a:A$ and right component $b:B$.

Let $\text{IntPackage::TYPE} =$
  $\text{Tuple x:Int f:Int→Int end}$;

let $\text{intPackage::IntPackage} =$
  $\text{tuple let x=0 let f=succ end}$;

intPackage.f(intPackage.x);
  $= 1:\text{Int}$
Abstract types
(Mitchell-Plotkin)

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

\begin{verbatim}
Let Package::TYPE =
   Tuple
      A::TYPE
      a:A f:A→Int
   end;

let package:Package =
   tuple
      Let A=Int
      let a=0 let f=succ
   end;
\end{verbatim}

An element of an abstract type can be implemented by \textit{its own} type

\begin{verbatim}
let package1:Package =
   tuple
      Let A=Package
      let a=package
      let f(p:Package)=p.f(p.a)
   end;
\end{verbatim}

uses impredicativity in an essential way
Polymorphism + abstract types

Let Stack :: \textsc{All}(A::\textsc{Type})\textsc{Type} =
\textsc{Fun}(A::\textsc{Type})
\textsc{Tuple}
  \textsc{S::Type}
  \text{empty: } S
  \text{push: } \textsc{All}(a:A \ s:S) \textsc{S}
  \text{top: } \textsc{All}(s:S)A
  \text{pop: } \textsc{All}(s:S)S
end;

:Stack(Int)
Quest subtyping
Power Kinds

For any type $A$, $\text{POWER}(A)$ is the kind of all the subtypes of $A$.

$A : \text{POWER}(B)$ means that $A <: B$ (A is a subtype of B)

$A <: B \approx A : \text{POWER}(B)$
$\text{fun}(A <: B) \ c \approx \text{fun}(A : \text{POWER}(B)) \ c$
$\text{All}(A <: B) \ C \approx \text{All}(A : \text{POWER}(B)) \ C$

Formation

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

Introduction

$S \vdash A <: B$

Elimination

$S \vdash a : A \quad S \vdash A <: B$

$S \vdash a : B$

Power-Power

$S \vdash A <: B$

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

Power-Type

$S \vdash A \text{ type}$

$S \vdash \text{POWER}(A) <: : \text{TYPE}$
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  <:   Object
      Car     <:   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*

\[
S \vdash A_1 <: B_1 \quad S \vdash A_n <: B_n \quad S \vdash A_m \text{ type}
\]
\[
S \vdash \text{Record } t_1:A_1, \ldots, t_n:A_n, \ldots, t_m:A_m \text{ end} <: \text{Record } t_1:B_1, \ldots, t_n:B_n \text{ end}
\]
Higher-order Subtypes

\[ S \vdash A' <: A \quad S \vdash B <: B' \]
\[ S \vdash A \rightarrow B <: A' \rightarrow B' \]
\[ S \vdash S'' <: S' \quad S, S'' \vdash A' <: A'' \]
\[ S \vdash \text{All}(S')A' <: \text{All}(S'')A'' \]

\textbf{let} speed: Vehicle \rightarrow Int = ... ;
\hspace{1em}Vehicle \rightarrow Int <: Car \rightarrow Int
\hspace{2.5em}(\text{speed takes cars})

\textbf{let} speed': \text{All}(A <: \text{Vehicle})A \rightarrow Int = ... ;
\hspace{1em}\text{All}(A <: \text{Vehicle})A \rightarrow Int <: \text{All}(A <: \text{Car})A \rightarrow Int

\textbf{let} serialNo: Int \rightarrow Car = ... ;
\hspace{1em}Int \rightarrow Car <: Int \rightarrow \text{Vehicle}
\hspace{3em}(\text{serialNo returns vehicles})

\textbf{let} f: Vehicle \rightarrow Vehicle = ... ;
\hspace{1em}Vehicle \rightarrow Vehicle <: \text{Car} \rightarrow \text{Object}

age(f(myCar));

\textbf{let} f': \text{All}(A <: \text{Vehicle}) A \rightarrow A = ... ;
\hspace{1em}(f': \text{Vehicle} \rightarrow \text{Vehicle})

age(:Car)(f': :Car)(myCar));
\hspace{2.5em}(f', \text{used as Car} \rightarrow \text{Object})

age(f'(myCar));
\hspace{3em}(\text{abbreviated})
The subsumption rule

\[ S \vdash a : A \quad S \vdash A \leq B \]
\[ S \vdash a : B \]

Then \texttt{myCar:Car} implies \texttt{myCar:Object}.

\texttt{let age': Object \rightarrow Int =}
\texttt{ fun(x:Object) x.age;}

\texttt{age'(myCar);}  
\texttt{= 3:Int}

The subsumption rule is useful but not sufficient by itself:

\texttt{let objId': Object \rightarrow Object =}
\texttt{ fun(a:Object) a;}

\texttt{objId'(myCar);}  
\texttt{= myCar:Object}

\texttt{objId'(myCar).speed; \quad Wrong!}
Subtyping + polymorphism

let age: \textbf{All}(A:\textless:\text{Object}) A \rightarrow \text{Int} =
\textbf{fun}(A:\textless:\text{Object}) \textbf{fun}(x:A) x.\text{age};

Here we must check $A:\textless:\text{TYPE}$, but $A:\textless:\text{POWER}($Object$)$. Hence we use embedding ($\text{POWER}(X)\textless:\text{TYPE}$). Then we must check that $x$ has a record type in $x.\text{age}$, but $x:A$. However $A:\textless:\text{Object}$, hence by subsumption $x:\text{Object}$.

age(:Car)(myCar);
\hspace{1em} = 3:\text{Int}

age(myCar); \hspace{1em} \text{the usual abbreviation}
\hspace{1em} = 3:\text{Int}

let objId : \textbf{All}(A:\textless:\text{Object}) A \rightarrow A =
\textbf{fun}(A:\textless:\text{Object}) \textbf{fun}(x:A) x;

objId(:Car)(myCar);
\hspace{1em} = myCar:Car

objId(:Car)(myCar).speed;
\hspace{1em} = 120:\text{Int}
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
```
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

S ⊨ B_1 type ... S ⊨ B_m type  ∀i∈1..n ∃j∈1..m S ⊨ a_i:B_j

S ⊨ set a_1, ..., a_n end : Set B_1, ..., B_m end
**Subtyping + set types**
(Buneman-Ohori)

Let \texttt{AllObjects::TYPE} = 
  \texttt{Set Object end};
Let \texttt{AllVehiclesAndMachines::TYPE} = 
  \texttt{Set Vehicle Machine end};
Let \texttt{AllCars::TYPE} = 
  \texttt{Set Car end};

\texttt{AllCars} <= \texttt{AllVehiclesAndMachines} <= \texttt{AllObjects}

let \texttt{allCars:AllCars =}
  \texttt{set myCar yourCar end};

\texttt{allCars} |>>| \texttt{set record age=3 end end;}
  = \texttt{set myCar end}

\texttt{LET RELATION} = \texttt{POWER(Set Record end end)}

\texttt{AllObjects :: RELATION}

\texttt{Join :: ALL(A,B::RELATION)RELATION}
|>>| : \texttt{ALL(A,B::RELATION)(A#B)->Join(A B)}

\[ \forall i \in 1..n \exists j \in 1..m \quad S \vdash A_i <: B_j \]

\[ S \vdash \text{Set } A_1, \ldots, A_n \text{ end } <= \text{Set } B_1, \ldots, B_m \text{ end} \]
Classes and methods (Hint)

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

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

A class (= object generator):

```plaintext
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):

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

Invocation of methods:

```plaintext
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
import ..
export ..
end;
```

```
module b: A
import ..
export ..
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.

\[
\text{import } b: A; \\
\text{let } b: A = ..
\]

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.

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

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

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

interface D
export
z:Int
end;

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

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

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

\[
\begin{align*}
\text{Let } A::\text{TYPE} &= \text{Tuple}
\text{ T::\text{TYPE}} \\
&\quad \text{new}(x:\text{Int}):\text{T} \\
&\quad \text{int}(x:\text{T}):\text{Int} \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{let } a:A &= \text{tuple} \\
&\quad \text{Let } T::\text{TYPE} = \text{Int} \\
&\quad \text{let } \text{new}(x:\text{Int}):\text{T} = x \\
&\quad \text{and } \text{int}(x:\text{T}):\text{Int} = x \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{Let } B::\text{TYPE} &= \text{Tuple}
\text{ x:a.T} \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{let } b:B &= \text{tuple} \\
&\quad \text{let } x = a.\text{new}(0) \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{Let } C::\text{TYPE} &= \text{Tuple}
\text{ f(x:a.T):Int} \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{let } c:C &= \text{tuple} \\
&\quad \text{let } f(x:a.T):\text{Int} = a.\text{int}(x)+1 \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{Let } D::\text{TYPE} &= \text{Tuple}
\text{ z:Int} \\
&\quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{let } d:D &= \text{tuple} \\
&\quad \text{let } z = c.f(b.x) \\
&\quad \text{end;}
\end{align*}
\]

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.
Open systems

Consider the following system organization.

![Diagram]

We express this arrangement by the following notation:

```plaintext
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  
import :A :C
export ...
end;

interface C of S  
import :B :C
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 $S$:

```plaintext
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

```plaintext
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.
Sealed systems

Sealing system $S$:

```plaintext
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.

Explicitly 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:

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

These objects can then be narrowed to a given type:

```haskell
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:

```plaintext
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 runtime 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.