An Abstract and Reusable Programming Language Architecture

Hassan Aït–Kaci

ILOG

Outline

- Motivation
- Overview
- Syntax
- Kernel
- Types
- Backend
- Conclusion

Motivation

- Synthesis of language front-ends to dedicated engines
- Syntactic interface between several application domains
- Scripting tools
- Rapid prototyping, academic experimentation, *etc.*, ...

Basic idea: Factoring out . . .

. . . syntax analysis,

. . . type checking,

. . . code generation.

Seeking an open design—*i.e.*, easily adaptable and extensible.

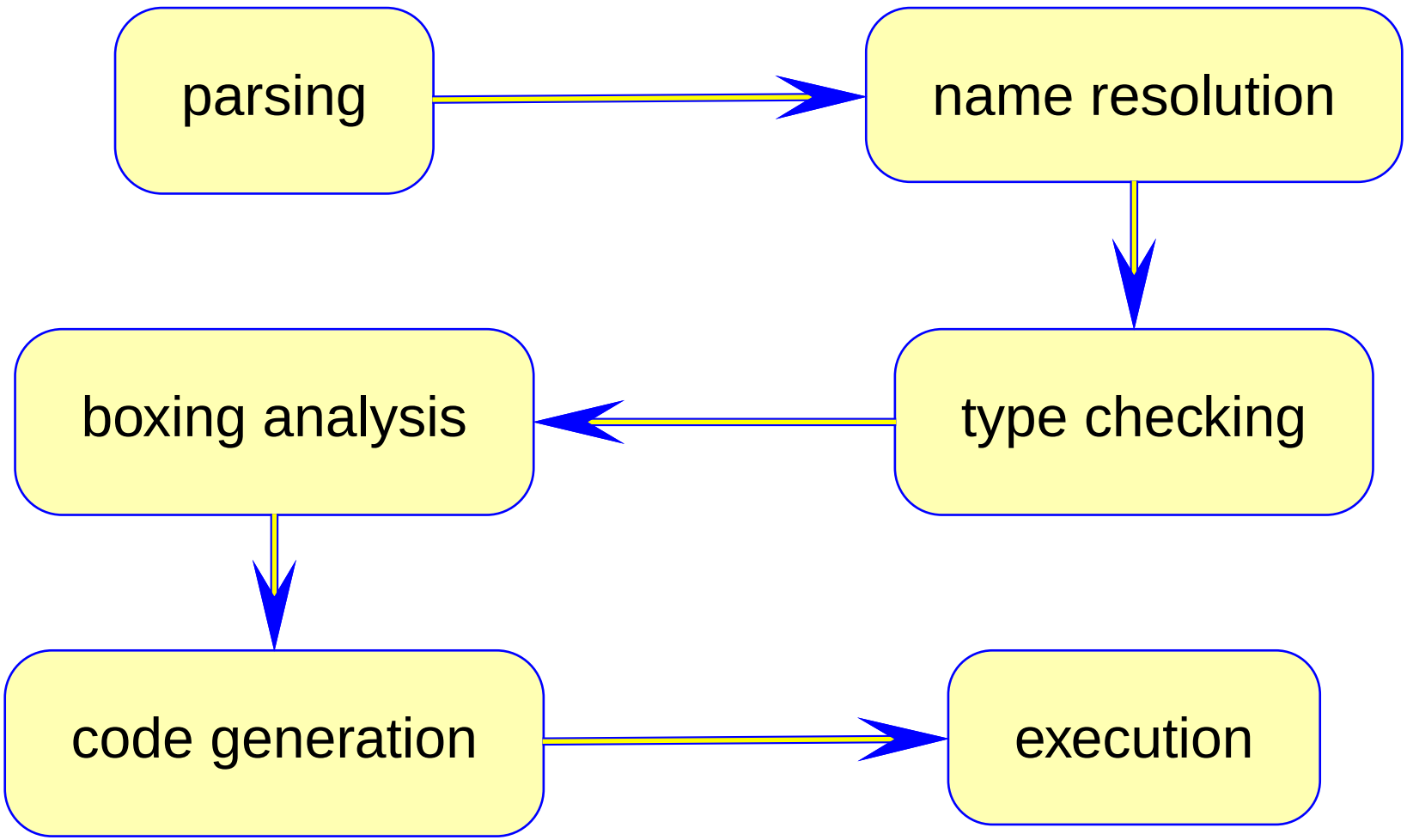
Overview

The design consists in Java packages and classes that offer a basic service.

One can thus import (some of) these packages and derive (some of) of these classes and (possibly) complete them with specific features to:

- Specify a grammar for a surface syntax;
- Possibly augment the basic design with new:
  - kernel expressions;
  - built-in types;
  - built-in instructions;
  - runtime objects and structures.
- Translate AST's into kernel language expressions.

Processing diagram



Syntax front-end

JACC - Just Another Compiler Compiler:

- “100% Pure Java” LALR(1) parser generator *à la* yacc
- AST construction and manipulation
- partial parsing
- dynamically (re-)definable operators *à la* Prolog

Jacc dynamic operators

```
%token '!'

%dynamic op1
%op1 '!' yf 200

%dynamic op2
%op2 '!' yfx 500

%%
expression : expression1 _op1_ expression1
           | expression2 _op2
           | '!' expression
           ;
```

Kernel Expressions

- constants
- names
- functions
- control
- indexed structures
- tuples
- objects
- collections
- updates
- nonstandard constructs
- monoids

Kernel Expressions (ctd.)

- constants
  - `Char`
  - `Int`
  - `Real`
  - `StringConstant`
  - `BuiltinObjectConstant`
- names
  - `Unresolved`
  - `Global`
  - `Local`
  - `Parameter`

Kernel Expressions (ctd.)

- functions
  - `Abstraction` $\mathfrak{function}\ x_1, \ldots, x_n \cdot e$
  - `Application` $f(e_1, \ldots, e_n)$
  - `Let` $\mathfrak{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathfrak{in}\ e$
- control
  - `Sequence` $\{\ e_1;\ \ldots; e_n\ \}$
  - `IfThenElse` $\mathfrak{if}\ e_1\ \mathfrak{then}\ e_2\ \mathfrak{else}\ e_3$
  - `And` $e_1\ \mathfrak{and}\ e_2$
  - `Or` $e_1\ \mathfrak{or}\ e_2$
  - `Loop` $\mathfrak{while}\ e_1\ \mathfrak{do}\ e_2$

Kernel Expressions (ctd.)

- indexed structures
  - `NewArray` $\mathfrak{new}\ type[indexer]\ldots[indexer]$
  - `ArraySlot` $e[e_1]\ldots[e_n]$
  - `ArrayExtension` $\#[e_1,\ldots,e_n]\#$ or $\#[i_1 : e_1,\ldots,i_n : e_n]\#$
  - `ArrayInitializer` `NewArray` $=$ `ArrayExtension`
  - `ArrayToMap` $e!indexer$
- tuples
  - `Tuple` $\langle e_1,\ldots,e_n\rangle$
  - `NamedTuple` $\langle p_1 := e_1,\ldots,p_n := e_n\rangle$
  - `TupleProjection` $e@p$

Kernel Expressions (ctd.)

- objects
  - `NewObject` $\mathfrak{new}\ C$ or $\mathfrak{new}\ C(T_1, \ldots, T_n)$
- collections
  - `NewSet` $\mathfrak{new}\ \{T\}$ or $\mathfrak{set}\{e_1, \ldots, e_n\}$
  - `NewList` $\mathfrak{new}\ \mathfrak{list}\{T\}$ or $\mathfrak{list}\{e_1, \ldots, e_n\}$
  - `NewBag` $\mathfrak{new}\ \mathfrak{bag}\{T\}$ or $\mathfrak{bag}\{e_1, \ldots, e_n\}$

Kernel Expressions (ctd.)

- updates
  - `Definition` $\mathfrak{def}\ id\ =\ e$ or $\mathfrak{def}\ id(v_1, \ldots, v_n)\ =\ e$
  - `Assignment` $l\ =\ e$
  - `UnresolvedAssignment`
  - `GlobalAssignment`
  - `LocalAssignment`
  - `ArraySlotUpdate` $e[e_1] \ldots [e_n]\ =\ e'$
  - `TupleUpdate` $t@p\ =\ e$
  - `FieldUpdate` $o.f\ =\ e$

Kernel Expressions (ctd.)

- nonstandard constructs
  - `ExitWithValue` — $\mathfrak{return}\ e$
  - `UndecidedExpression` — $e_1\ ?\ e_2$
  - `HideType` — $e\ \mathfrak{as}\ T$
  - `OpenType` — $\$e$
- monoids
  - `Homomorphism`
  - `Comprehension`
  - `FilterHomomorphism`

Functions

$$\frac{\Gamma[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash \mathfrak{function}\ x_1, \ldots, x_n \cdot e : T_1, \ldots, T_n \to T}$$

$$\frac{\Gamma \vdash e_1 : T_1, \cdots, \Gamma \vdash e_n : T_n, \Gamma \vdash f : T_1, \ldots, T_n \to T}{\Gamma \vdash f(e_1, \ldots, e_n) : T}$$

Implicit currying:

$$\frac{S_1 = T_1,\ \ldots,\ S_n = T_n,\ S = T_{n+1}, \ldots, T_{n+k} \to T}{S_1, \ldots, S_n \to S = T_1, \ldots, T_n, T_{n+1}, \ldots, T_{n+k} \to T}$$

Let

$$\frac{\Gamma \vdash e_1 : T_1, \cdots, \Gamma \vdash e_n : T_n, \Gamma[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash \mathfrak{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathfrak{in}\ e : T}$$

NB: this $\mathfrak{let}$ is *not* polymorphic and amounts to:

$$\mathfrak{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathfrak{in}\ e$$

$$\stackrel{\text{def}}{=}$$

$$\big(\mathfrak{function}\ x_1, \ldots, x_n\ \cdot\ e\big)(e_1, \ldots, e_n)$$

Control expressions

$$\frac{\Gamma \vdash e_1 : T_1, \ldots, \Gamma \vdash e_n : T_n}{\Gamma \vdash \{ e_1; \ldots; e_n \} : T_n}$$

$$\frac{\Gamma \vdash c : \mathfrak{Boolean}, \Gamma \vdash e_1 : T, \Gamma \vdash e_2 : T}{\Gamma \vdash \mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2 : T}$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{Boolean}, \Gamma \vdash e_2 : \mathfrak{Boolean}}{\Gamma \vdash e_1\ \mathfrak{and/or}\ e_2 : \mathfrak{Boolean}}$$

$$\frac{\Gamma \vdash c : \mathfrak{Boolean}, \Gamma \vdash e : T}{\Gamma \vdash \mathfrak{while}\ c\ \mathfrak{do}\ e : \mathfrak{Void}}$$

Monoid homomorphisms

Monoid: data type with associative binary operation and identity

Type	Operation	Identity
$\mathfrak{Int}$	$+$	0
$\mathfrak{Int}$	$*$	1
$\mathfrak{Real}$	$+$	0.0
$\mathfrak{Real}$	$*$	1.0
$\mathfrak{Int}$	$\mathfrak{max}$	$-\infty$
$\mathfrak{Int}$	$\mathfrak{min}$	$+\infty$
$\mathfrak{Boolean}$	$\mathfrak{or}$	$\mathfrak{false}$
$\mathfrak{Boolean}$	$\mathfrak{and}$	$\mathfrak{true}$
$\mathfrak{Set}$	$\mathfrak{union}$	$\{\}$
$\mathfrak{List}$	$\mathfrak{append}$	$[]$
⋮	⋮	⋮

Monoid homomorphism

A monoid homomorphism expresses an *declarative iteration*.

Example (list):

$$\mathfrak{hom}^{\mathbb{1}}_{\star}(f)([]) = \mathbb{1}$$

$$\mathfrak{hom}^{\mathbb{1}}_{\star}(f)([H|T]) = f(H) \star \mathfrak{hom}^{\mathbb{1}}_{\star}(f)(T)$$

Clearly, this scheme extends a function f to a homomorphism of monoids, from the monoid of lists to the monoid defined by $\langle \star, \mathbb{1} \rangle$.

Primitive homomorphisms

A *primitive homomorphism* computing a value of type T consists of:

- a collection iterated over—of type $coll(T')$;
- a function applied to each element—of type $T' \rightarrow T$;
- a monoid operation—of type $T, T \rightarrow T$;
- an identity—of type T.

Collection homomorphisms

A *collection homomorphism* expression constructing a collection of type $coll(T)$ consists of:

- a collection iterated over—of type $coll'(T')$;
- a function applied to each element—of type $T' \rightarrow coll(T)$;
- an operation "adding" an element to a collection—of type $T, coll(T) \rightarrow coll(T)$
- an identity—of type $coll(T)$.

Monoid comprehension

$$\langle \oplus, \mathbb{1} \rangle \{ e \mid q_1, \ldots, q_n \}$$

- $\langle \oplus, \mathbb{1} \rangle$ is a monoid,
- e is an expression,
- q_i is a *qualifier*:
  - an expression e, or
  - a pair $x \leftarrow e$, where x is a variable and e is an expression.

```
[+,set{}] { <x,y> | x <- 1..5, y <- x..6, (x+y)%2 == 0 };
```

```
{<1,1>,<1,3>,<1,5>,<2,2>,<2,4>,<2,6>,<3,3>,<3,5>,<4,4>,<4,6>,<5,5>} : {<int,int>}
```

Monoid comprehension

A monoid comprehension is defined in terms of homomorphisms:

[Fegaras-Maier: TODS-1996]

$$\langle \oplus, \mathbb{1} \rangle \{ e \mid \} \quad \stackrel{\text{def}}{=} \quad e \oplus \mathbb{1}$$

$$\langle \oplus, \mathbb{1} \rangle \{ e \mid x \leftarrow e', Q \} \quad \stackrel{\text{def}}{=} \quad \mathfrak{hom}^{\mathbb{1}}_{\oplus} \big(\lambda x. \langle \oplus, \mathbb{1} \rangle \{ e \mid Q \} \big) (e')$$

$$\langle \oplus, \mathbb{1} \rangle \{ e \mid c, Q \} \quad \stackrel{\text{def}}{=} \quad \mathfrak{if}\ c\ \mathfrak{then}\ \langle \oplus, \mathbb{1} \rangle \{ e \mid Q \}\ \mathfrak{else}\ \mathbb{1}$$

Monoid comprehensions

Monoid comprehensions (based on homomorphisms) allow to express a complete formal *Object Query Calculus* [Fegaras-Maier: TODS-1996].

Monoid comprehensions

- facilitate optimization through expression normalization
- complete naturally the λ-calculus
- enable efficient declarative iteration using RDB techniques

We adapted the comprehension calculus to work in a language setting.

Types

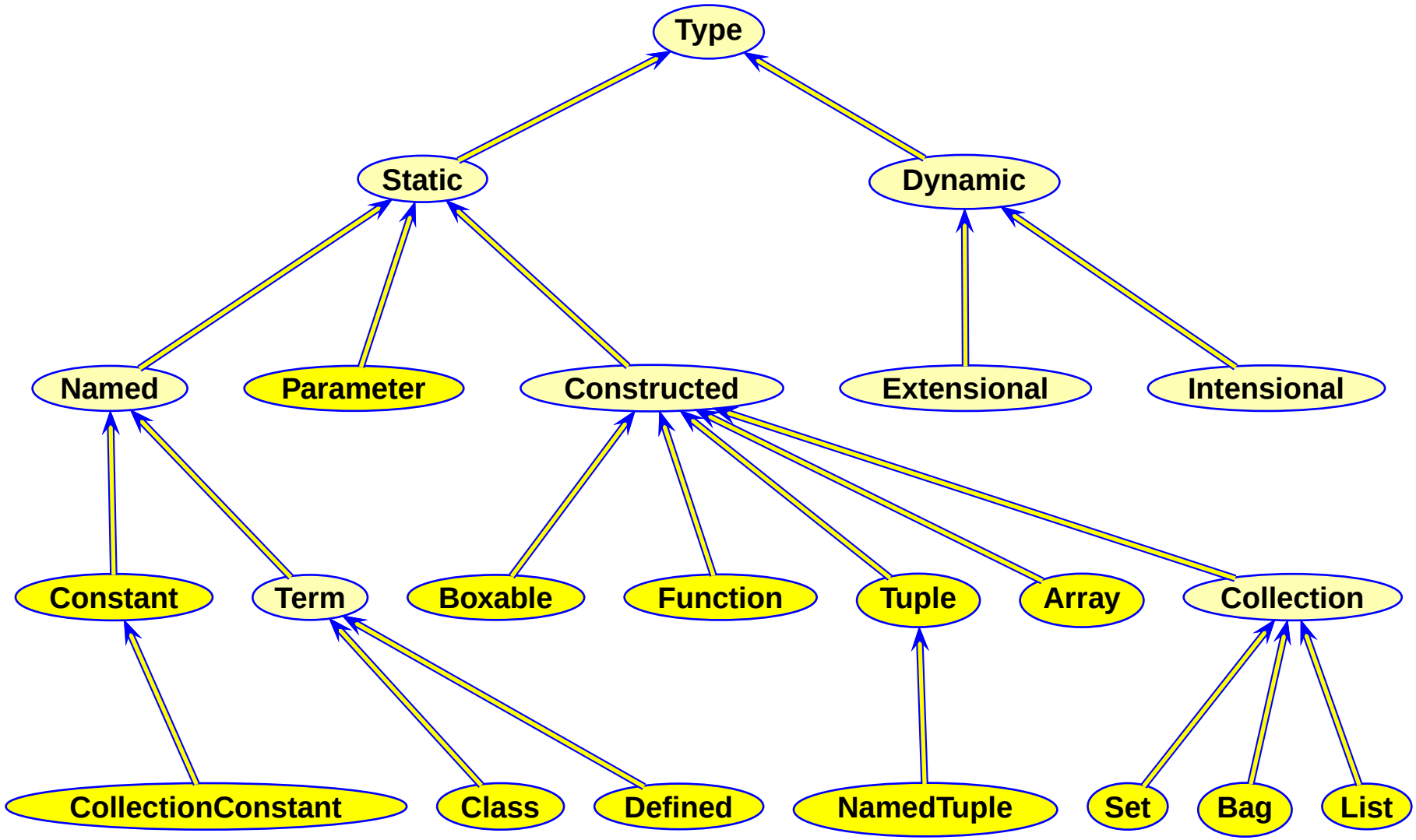
- polymorphism
- overloading
- currying
- boxing/unboxing
- dynamic types
- classes
- type definitions

Polymorphism

ML-polymorphism (*i.e.*, 2*nd*-order universal)—*e.g.*:

$$
\begin{array}{lcl}
\mathit{Type} & ::= & \mathit{SimpleType} \mid \mathit{TypeScheme} \\
\mathit{SimpleType} & ::= & \mathit{BasicType} \mid \mathit{FunctionType} \mid \mathit{TypeParameter} \\
\mathit{BasicType} & ::= & \mathfrak{Int} \mid \mathfrak{Real} \mid \mathfrak{Boolean} \mid \ldots \\
\mathit{FunctionType} & ::= & \mathit{SimpleType} \rightarrow \mathit{SimpleType} \\
\mathit{TypeParameter} & ::= & \alpha \mid \alpha' \mid \ldots \mid \beta \mid \beta' \mid \ldots \\
\mathit{TypeScheme} & ::= & \forall\ \mathit{TypeParameter}\ .\ \mathit{Type}
\end{array}
$$

The Type System



Class type

Declaring a class type and defining its implementation causes the following:

- the name of the class is entered with a new type for it in the type symbol table;
- each field of a distinct type is assigned an offset in an array of slots (per sort);
- each method and field expression is name-resolved, type-checked, after closing it into an abstraction taking $\mathfrak{this}$ as first argument;

Class type

- each method definition is compiled into a global definition, and each field is compiled into a global function corresponding to accessing its value from the appropriate offset;
- finally, each field's initialization expression is compiled and recorded to be used at object creation time.

An object may be created at run-time (using the `new` operator followed by a class name).

Class type

```
class classname { interface } [ { implementation } ]
```

The *interface* block specifies the type signatures of the *members* (*fields* and *methods*) of the class and possibly initial values for fields.

The *implementation* block is optional and gives the definition of (some or all of) the methods.

Class type

```
class Counter{ val : Int = 1;
               method set : Int → Counter;
             }
             { set(val : Int) : Counter = (this.val = val);
             }

def set(x : Counter, n : Int) : Counter = (x.val = n);

c = new Counter;

c.set(c.val + 2);
write(c.val);

set(c, val(c) + 2);
write(val(c));
```

The Type Checker

- Backtracking prover that establishes goal objects.
- *e.g.*: a `TypingGoal` consists of an expression and a type:

  proving a `TypingGoal` amounts to unifying its expression component's type with its type component.
- Such goals are spawned in the typechecker by the type checking method of expressions as per their type checking rules.

The Type Checker (ctd.)

- Some globally defined symbols may have multiple types: keep choices and backtrack to alternative types upon failure.
- the typechecker maintains all the necessary structures for undoing:
  - type variable binding,
  - function type currying,
  - application expression currying.

Some typing goals

- EmptyGoal
- TypingGoal
- UnifyGoal
- GlobalTypingGoal
- SubTypeGoal
- BaseTypeGoal
- ArrayIndexTypeGoal
- PruningGoal
- PushExitableGoal
- PopExitableGoal
- CheckExitableGoal
- ResiduatedGoal
- ShadowUnifyGoal
- UnifyBaseTypeGoal
- NoVoidTypeGoal

Boxing/Unboxing

Polymorphic code must work for either:

- primitive unboxed types (*e.g.*, $\mathfrak{Int}$, $\mathfrak{Real}$, *etc.*)
- boxed types

Problem: how to compile a polymorphic function into code that knows the actual runtime sorts of the function's runtime arguments and returned value, *before the function type is actually instantiated*?

The problem was addressed by Xavier Leroy 11 years [POPL'92].

Boxing/Unboxing

Leroy's method:

- type annotation enabling a source-to-source transformation;
- source transformation: generate of *wrappers* and *unwrappers* for boxing and unboxing expressions whenever necessary;
- compile the transformed source as usual.

We adapt and improve the main idea from Leroy's method:

- type annotation and rules are greatly simplified;
- no source-to-source transformation is needed;
- un/wrappers generation is done at code-generation time.

This saves a great amount of space and time.

Backend system

Intermediate code is executed in the context of *Runtime*.

A runtime state (encapsulated by an object in this class) is that of a stack automaton supporting the computations of a higher-order functional language with lexical closures.

This may be viewed as an optimized variant of Peter Landin's SECD machine in the same spirit as Luca Cardelli's Functional Abstract Machine (FAM).

The backend system also defines runtime objects for tuples, maps, sets, *etc.*, ...

Conclusion

This architecture offers a compromise between formal executable specification systems and pragmatic language definition (a *poor man's language kit?...*).

It enables low-cost development of programming languages with basic and advanced features.

Importantly, it is *open* and favors ease of extension and interoperability.

Much more remains to be done (*e.g.*, namespaces, access management, rule-based programming, logic programming, type logics, *etc.*, ...)

Thank You Very Much !