Core ABS

Hats meeting, Sept. 2009

http://www.hats-project.eu

Principles

- object-oriented
- concurrent
- simple core
- typed
- amendable to verification
- formal/executable semantics
- "Java"-like "syntactic feeling"
- functional sub-part

Object-orientation

- currently rudimentary:
  - objects,
  - classes,
  - interfaces
  - method calls

  (and that's it)
- no "code-reuse" yet (inheritance, traits, mixins, feature composition).

$$
\begin{array}{rcll}
D & ::= & I \mid D_C \ldots & \text{declarations} \\
I & ::= & \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{m} : \overline{T}\} & \text{interface} \\
D_C & ::= & \texttt{class}\ C(\overline{f} : \overline{T})\ \texttt{implements}\ \overline{I}\{f{:}T = \overline{ei}; \overline{MD}\} & \text{class def.} \\
MD & ::= & T\ m(\overline{x} : \overline{T})\{e\} & \text{method defin} \\
 & & & \\
e & ::= & & \text{full expressio} \\
 & \mid & \ldots \mid \texttt{this} \mid \texttt{this}.f \mid \texttt{this}.f := e' \mid & \\
 & \mid & e.m(\overline{e}) \mid e!m(\overline{e}) \ldots &
\end{array}
$$

Method calls

- asynchronous and synchronous
- caller decides
- asynchronous: core of the concurrency model
- synchronous: inside 1 object (group of objects)
- note: synchronous calls $\neq$ async. call + immediate insisting on getting the value.

$$e \quad ::= \quad \ldots \qquad e.m(\overline{e}) \mid e!m(\overline{e}) \ldots$$

Concurrency

- "active" objects (à la Creol)
- field are "private"
- objects = unit of concurrency and of state
- futures for "returning" results
- one object: acts as monitor
  - mutex + cooperative scheduling
  - release points
    - when claiming results from async. method calls (futures)
    - explicitly (`yield`)
- two ways of claiming a result from future ref.
  - insistive : no release
  - polite : release, when value not available

$$e \quad ::= \quad \ldots e.\texttt{await} \mid e.\texttt{get} \mid \texttt{yield} \ldots$$

Functional/data part

- side-effect free expressions
- simple language for inductive data types
- "generic"[1]
- pattern matching
- part of the assertion language

[1] No generics for classes currently

$$
\begin{array}{rcll}
D & ::= & \ldots \mid ADT \mid F & \text{declarations} \\
F & ::= & \texttt{def}\ fn\ \langle \overline{X} \rangle (\overline{x} : \overline{T}) : T = ea & \text{functions} \\
ADT & ::= & \texttt{data}\ dn \langle \overline{X} \rangle = constrlist & \\
constrlist & ::= & constr\ "|"\ constrlist & \\
constr & ::= & cn(\overline{T}) & \text{constructor} \\
e & ::= & \ldots fn(e) \mid cn(\overline{e}) & \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ branchlist\ \texttt{end} & \\
branchlist & ::= & e \mid branch\ "|"\ branchlist & \\
branch & ::= & pattern \rightarrow e & \\
pattern & ::= & x \mid "\_" \mid cn(\overline{pattern}) &
\end{array}
$$

AOG ("co-boxes")

- "active object groups"[2]
- moderate extension of Creol
- "lightweight" component notion
- "unit of concurrency": group of objects
- synchronous calls: allowed inside an AOG
- instantiator determines whether an "ordinary" object or an "AOG" is created.
- not nested
- not referrable at user-level

$$e_i \quad ::= \quad \text{new } C(\overline{ea}) \mid ea \mid \text{new aog } C(\overline{e})$$

[2] more lucid names welcome

Imperative/sequential core

- in tendency:
  - imperative variables = fields = heap
  - non-imperative variables = "stack", substitution semantics
- wish:
  - unproblematic initialization
  - no nil-pointer-exceptions

$$
\begin{array}{lll}
e & ::= & \text{full expression} \\
& \mid & x \mid \texttt{this} \mid \texttt{this}.f \mid \texttt{this}.f := e' \mid fn(e) \\
& \mid & e = e \mid \text{let } x{:}T = e \text{ in } e \mid \texttt{var}x := e \\
& \mid & \texttt{case } e \texttt{ of } \textit{branchlist} \texttt{ end}
\end{array}
$$

- three level of imperativeness
  1. side effect free/functional
  2. usable in constructors and initializing expressions
  3. statements
- without $\texttt{var}x := e$: substitution semantics

Type system

- classes/class names are no types (for the user)
- no subtyping on class types
- interface as types
- "multiple subtyping" for interfaces (nominal)
- universal polymorphism ("generics") for the data type language, not for classes
- no type inference, no let-polymorphism
- no first-class functions in the data language
- poor man's exceptions

$$\begin{array}{rcl} T & ::= & dn\langle\overline{T}\rangle \mid I \mid X \mid C \\ T_S & ::= & \overline{T} \to T \end{array}$$

Proposal for syntax

P	::=	$\overline{D}\ e$	program
D	::=	$I \mid D_C \mid ADT \mid F$	declaration
I	::=	$\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{m} : \overline{T_S}\}$	interface
D_C	::=	$\texttt{class}\ C(\overline{f} : \overline{T})\ \texttt{implements}\ \overline{I}\{\overline{f}{:}\overline{T} = \overline{ei}; \overline{MD}\}$	class declaration
F	::=	$\texttt{def}\ fn\ \langle\overline{X}\rangle(\overline{x} : \overline{T}){:}T = ea$	functions
MD	::=	$T\ m(\overline{x} : \overline{T})\{e\}$	method definition
ADT	::=	$\texttt{data}\ dn\langle\overline{X}\rangle = constrlist$	
$constrlist$	::=	$constr\ "\mid"\ constrlist$	
$constr$	::=	$cn(\overline{T})$	constructor
e	::=		full expression
	$\mid$	$x \mid \texttt{this} \mid \texttt{this}.f \mid \texttt{this}.f := e' \mid fn(e)$	
	$\mid$	$e.m(\overline{e}) \mid e!m(\overline{e}) \mid e = e \mid \texttt{let}\, x{:}T = e\, \texttt{in}\, e \mid cn(\overline{e})$	
	$\mid$	$[\texttt{var}x := e]$	
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ branchlist\ \texttt{end}$	
	$\mid$	$e.\texttt{await} \mid e.\texttt{get} \mid \texttt{yield}$	
e_i	::=	$ea \mid \text{new}\ C(\overline{ea}) \mid \text{new}\ \texttt{aog}\ C(\overline{e})$	initialising expressions
ea	::=	$\ldots$	side-effect free expressions
$branchlist$	::=	$e \mid branch\ "\mid"\ branchlist$	
$branch$	::=	$pattern \rightarrow e$	
$pattern$	::=	$x \mid "\_" \mid cn(\overline{pattern})$	
T	::=	$dn\langle\overline{T}\rangle \mid I \mid X \mid C$	
T_S	::=	$\overline{T} \rightarrow T$	

Syntactic sugar

- core language: rather minimal
- surface language: more sugar welcome.
- examples
  - if-then-else
  - while-loop?
  - ...

What has been left out compared to Creol

- "code reuse" (inheritance etc)
- more complex "monitor statements" (free `await` ...)
- promises
- co-interfaces
-