

Behavioral interface description of an object-oriented language with futures and promises

Erika Ábrahám¹, Immo Grabe², Andreas Grüner², and Martin Steffen³

Research Centre Jülich, Germany

Christian-Albrechts University Kiel, Germany

University of Oslo, Norway

Sintef, Oslo

Feb. 2008



Structure

Introduction

Futures and promises

Interface description

Results & conclusion

Introduction

Futures and promises

Interface description

Results & conclusion

Motivation

- we are interested in
 - (observable) interface behavior of
 - components (i.e. open programs)
 - written in oo languages, here *Creol*
- interface behavior:
 - component/environment interactions
 - synchronous message passing (method invocations)
 - $\Rightarrow$ (interface) **traces** = sequences of calls and returns
- so far: objects, classes, threads, thread classes, monitors
...
- **now:** **asynchronous** method calls, **futures**, **promises**

Creol: a concurrent object model

- executable oo modeling language **concurrent** objects
- formal semantics in **rewriting logics** /Maude
- strongly **typed**
- method invocations: synchronous or **asynchronous**
- recently: concurrent objects by (first-class) futures
- **dynamic reprogramming**: class definitions may *evolve at runtime*

Road map

- characterization of the interface behavior
 - design goals
 - (preferably) seamless extension of the calculus with an
 - eye to compositionality
- ⇒ clean separation of concerns between
- assumptions* vs. *commitments*
- intuitively:
 - enabledness of input must depend only on the environment (= assumption)
 - enabledness of output must depend only on the component (= commitments)
 - interface *trace* must contain all relevant information relevant (and not part of the internal state(s))
 - easy comparison with Java multi-threading

Semantics

1. operational semantics
2. remember the design-goals
3. two stages
 - internal semantics
 - closed system
 - spec. of the “virtual machine”
 - external semantics
 - interaction with environment via
 - message passing (calls/returns)

Introduction

Futures and promises

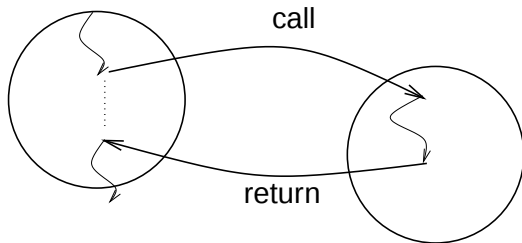
Interface description

Results & conclusion

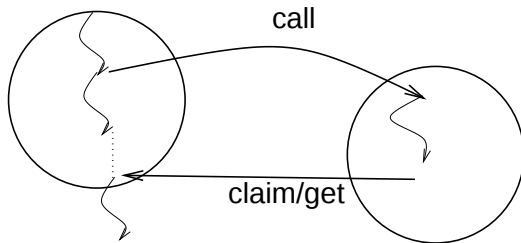
Futures

- introduced in the concurrent Multilisp language [7] [3]
- originally: **transparent** concurrency compiler annotation
- future e:
 - evaluated potentially **in parallel** with the rest $\Rightarrow$ 2 threads (producer and consumer)
 - future variable **dynamically generated**
 - when evaluated: future **identified** with value
- **wait-by-necessity** [4] [5]
- supported by Oz, Alice, MultiLisp, ... (shared state concurrency), Io, Joule, E, and most actor languages (Act1/2/3 ..., ASP), Java

Async. method calls and futures



Async. method calls and futures



Syntax

- $o@l(\vec{v})$: asynchronous method call, non-blocking
- execution:
 1. create a “placeholder”/reference to the eventual result: future reference
 2. initiate execution of method body
 3. continue to execute (= non-blocking, asynchronous) +

$e ::= \dots \mid o@l(v, \dots, v) \mid \textit{claim}@(\textcolor{red}{n}, o) \mid \underline{\textit{get}@n} \mid \dots$

Operational semantics

- strict separation of **internal** and **external** behavior
- ⇒ 2 stages
- internal steps (modulo congruence)
 - external, interface steps (labeled transitions)
- internal one: rather standard OS.

$$n\langle \text{let } x:T = \textcolor{red}{v} \text{ in } t \rangle \rightsquigarrow n\langle t[v/x] \rangle \quad \text{RED}$$

$$c[[F, M]] \parallel n\langle \text{let } x:c = \textcolor{red}{\text{new } c} \text{ in } t \rangle \rightsquigarrow \\ c[[F, M]] \parallel \textcolor{red}{\nu(o:c)}. (\quad o[c, F, \perp] \parallel n\langle \text{let } x:c = \textcolor{red}{o} \text{ in } t \rangle \quad) \quad \text{NEW}_i$$

$$n\langle \text{let } x:[T] = o@l(\vec{v}) \text{ in } t \rangle \rightsquigarrow \\ \nu(n':[T]). (n\langle \text{let } x:[T] = n' \text{ in } t \rangle \parallel n'\langle \text{let } x:T = \text{grab}(o); M.l(o)(\vec{v}) \text{ in } \text{release}(o); x \rangle)$$

$$n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = \text{claim}@ (n_1, o) \text{ in } t \rangle \rightsquigarrow n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t \rangle \quad \text{CLAIM}$$

$$n\langle \text{let } x:[T] = o@l(\vec{v}) \text{ in } t \rangle \rightsquigarrow \\ \nu(n':[T]). (n\langle \text{let } x:[T] = n' \text{ in } t \rangle \parallel n'\langle \text{let } x:T = \text{grab}(o); M.l(o)(\vec{v}) \text{ in } \text{release}(o); x \rangle)$$

$$n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = \text{claim}@ (n_1, o) \text{ in } t \rangle \rightsquigarrow n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t \rangle \quad \text{CLAIM}$$

$$\frac{t_2 \neq v}{n_2\langle t_2 \rangle \parallel n_1\langle \text{let } x : T = \text{claim}@ (n_2, o) \text{ in } t'_1 \rangle \rightsquigarrow n_2\langle t_2 \rangle \parallel n_1\langle \text{let } x : T = \text{release}(o); \text{get}@_{n_2} \text{ in } \text{grab}(o); t'_1 \rangle} \text{CLAIM}_I^2$$

$$n\langle \text{let } x:[T] = o@l(\vec{v}) \text{ in } t \rangle \rightsquigarrow \\ \nu(n':[T]). (n\langle \text{let } x:[T] = n' \text{ in } t \rangle \parallel n'\langle \text{let } x:T = \text{grab}(o); M.l(o)(\vec{v}) \text{ in } \text{release}(o); x \rangle)$$

$$n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = \text{claim}@ (n_1, o) \text{ in } t \rangle \rightsquigarrow n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t \rangle \quad \text{CLAIM}$$

$$\frac{t_2 \neq v}{n_2\langle t_2 \rangle \parallel n_1\langle \text{let } x : T = \text{claim}@ (n_2, o) \text{ in } t'_1 \rangle \rightsquigarrow n_2\langle t_2 \rangle \parallel n_1\langle \text{let } x : T = \text{release}(o); \text{get}@n_2 \text{ in } \text{grab}(o); t'_1 \rangle} \text{CLAIM}_i^2$$

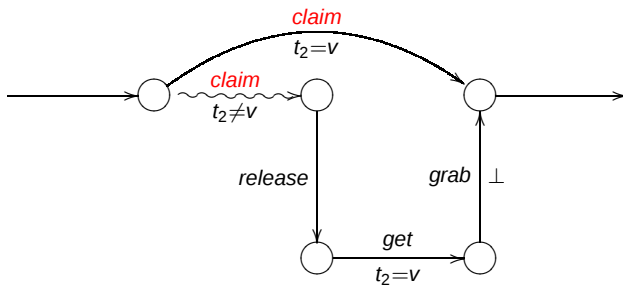
$$n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = \text{get}@n_1 \text{ in } t \rangle \rightsquigarrow n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t \rangle \quad \text{GET}_i$$

$n\langle \textit{suspend}(o); t \rangle \rightsquigarrow n\langle \textit{release}(o); \textit{grab}(o); t \rangle$ SUSPEND

$o[c, F, \perp] \parallel n\langle \textit{grab}(o); t \rangle \xrightarrow{\tau} o[c, F, \top] \parallel n\langle t \rangle$ GRAB

$o[c, F, \top] \parallel n\langle \textit{release}(o); t \rangle \xrightarrow{\tau} o[c, F, \perp] \parallel n\langle t \rangle$ RELEASE

Claiming a future



Futures and promises

- terminology is not so clear
- relation to **handled** futures
- **promises** [9], l-structures [2]

⇒ 2 aspects of future var:

- **write** = value of e “stored” to future
- **read** by the clients
- *promises*: **separating** the creation of future-reference from attaching code to it¹
- good for delegation

¹as in for async. calls

Syntax (promise)

- instead of $o@l(\vec{v})$
- split into
 1. **create** a promise²
 2. **fulfill** the promise = **bind** code to it.

$e ::= \dots \mid \textit{promise } T \mid \textit{bind } o.l(\vec{v}) : T \hookrightarrow n \mid \dots$

²or a **handle** to the future.

$$n' \langle \text{let } x:T' = \textit{promise } T \text{ in } t \rangle \rightsquigarrow \nu(n:T').(n' \langle \text{let } x:T' = n \text{ in } t \rangle) \quad \text{PROM}$$

$$\begin{aligned} & \dots n_1 \langle \text{let } x:T = \textit{bind } o.l(\vec{v}) : T_2 \hookrightarrow n_2 \text{ in } t_1 \rangle \xrightarrow{\tau} \\ & \quad \dots n_1 \langle \text{let } x:T = n_2 \text{ in } t_1 \rangle \quad \text{BIND}_i \\ & \quad \parallel (n_2 \langle \text{let } x:T_2 = \textit{grab}(o); M.l(o)(\vec{v}) \text{ in } \textit{release}(o); x \rangle) \end{aligned}$$

Calculus

- based on an concurrent **object calculus**
- almost no changes to the representation of *Java*-like threading
- on level of “components”: objects o , classes c , “activities”
- $n\langle t \rangle$: code t + “**future reference**” n
- basically (syntactically): two additions:³ **claiming** the future + **suspension** points

$claim@n, o$ and $suspend(n)$

- few more additions of run-time syntax: $get@n$ + lock handling

$grab(n)$ and $release(n)$

³user level

Syntax

C	$::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n:T).C} \mid n\llbracket O \rrbracket \mid \underline{n[n, F, L]} \mid \underline{n\langle t \rangle} \mid \underline{n\langle \bullet \rangle}$	component
O	$::= F, M$	object
M	$::= l = m, \dots, l = m$	method suite
F	$::= l = f, \dots, l = f$	fields
m	$::= \varsigma(n:T).\lambda(x:T, \dots, x:T).t$	method
f	$::= \varsigma(n:T).\lambda().v \mid \varsigma(n:T).\lambda().\perp_{n'}$	field
t	$::= v \mid \text{stop} \mid \text{let } x:T = e \text{ in } t$	thread
e	$::= t \mid \text{if } v = v \text{ then } e \text{ else } e \mid \text{if } \text{undef}(v.l()) \text{ then } e \text{ else } e$ $\mid \text{promise } T \mid \text{bind } n.l(\vec{v}) : T \hookrightarrow n \mid v.l() \mid v.l := \varsigma(s:n).\lambda().v$ $\mid \text{new } n \mid \text{claim}@ (n, n) \mid \underline{\text{get}@n} \mid \text{suspend}(n) \mid \underline{\text{grab}(n)} \mid \underline{\text{release}(n)}$	expr.
v	$::= x \mid n \mid ()$	values
L	$::= \perp \mid \top$	lock status

Introduction

Futures and promises

Interface description

Results & conclusion

Interface description: Task

- characterize possible interface behavior
- possible = adhering to the restriction of the language
 - well-typed
- basis of a trace logic / interface description
- abstraction process:
 - not $C \xRightarrow{t} \acute{C}$?
 - rather: consider C in a context / environment

$$C \parallel E \xRightarrow[\bar{t}]{t} \acute{C} \parallel \acute{E}$$

for some environment E

⇒ open semantics

$$\Delta \vdash C : \Theta \xRightarrow{t} \acute{\Delta} \vdash C : \acute{\Theta}$$

- assumptions Δ abstracts environments E

One step further: legal traces

- open semantics

$$\Delta \vdash C : \Theta \xRightarrow{t} \Delta' \vdash C : \Theta'$$

abstracts the environment

- existential abstraction of component, as well:
- characterization of *principally possible* interface behavior

$$C \parallel E \xRightarrow[\bar{t}]{t} C' \parallel E'$$

for **some** component C + **some** environment E

$\Rightarrow$ **legal trace**

$$\Delta \vdash t : trace :: \Theta$$

Type system

- judgments of the form (component level)

$$\Delta \vdash C : \Theta$$

- assures absence of “message-not-understood” etc errors
- static typing/subject reduction
- simple form of subtyping

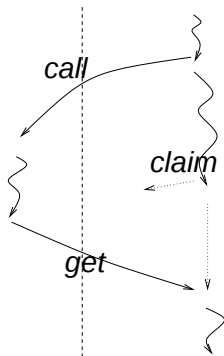
$$\begin{array}{c}
\frac{}{\Delta \vdash \mathbf{0} : ()} \text{T-EMPTY} \quad \frac{\Delta_1, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta_2, \Theta_1 \vdash C_2 : \Theta_2}{\Delta_1 \oplus \Delta_2 \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2} \text{T-PAR} \quad \frac{\Delta \vdash C : \Theta, n:T}{\Delta \vdash \nu(n:T).C : \Theta} \text{T-NU} \\
\\
\frac{; \Delta, c:T \vdash \llbracket O \rrbracket : T}{\Delta \vdash c \llbracket O \rrbracket : (c:T)} \text{T-NCLASS} \quad \frac{; \Delta \vdash c : \llbracket T_F, T_M \rrbracket \quad ; \Delta, o:c \vdash [F] : [T_F]}{\Delta \vdash o[c, F, l] : (o:c)} \text{T-NOBJ} \\
\\
\frac{; \Delta, n:[T]^+ \vdash t : T}{\Delta \vdash n \langle t \rangle : (n:[T]^+)} \text{T-NTHREAD} \quad \frac{}{\Delta \vdash n \langle \bullet \rangle : (n:[T]^{+-})} \text{T-NTHREAD} \\
\\
\frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'} \text{T-SUB}
\end{array}$$

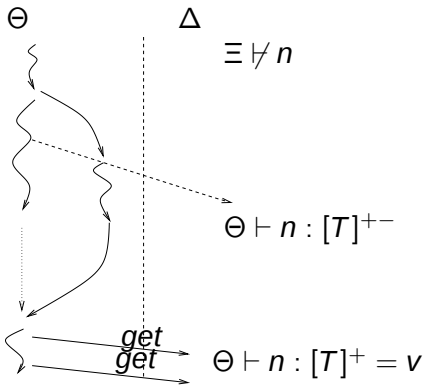
$\Xi \vdash \epsilon : \text{trace}$ L-EMPTY

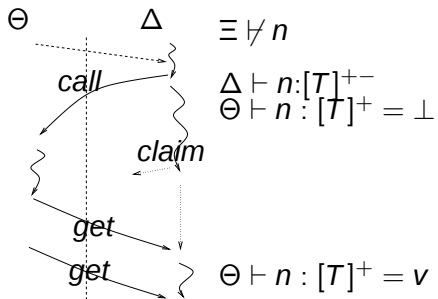
$$\frac{\begin{array}{l} a = \nu(\Xi'). n\langle \text{call } o.l(\vec{v}) \rangle? \quad \Xi' = \Xi + a \quad (\Xi' \vdash n \vee \Delta \vdash n : \boxed{\text{red}}^{+-}) \\ \Xi' \vdash o.l? : \vec{T} \rightarrow \mathcal{T} \quad \Xi' \vdash [a] : \vec{T} \rightarrow _ \quad \Xi' \vdash s : \text{trace} \end{array}}{\Xi \vdash a \ s : \text{trace}} \text{ L-CALLI}$$

$$\frac{a = \nu(\Xi'). n\langle \text{get}(v) \rangle? \quad \Xi' = \Xi + a \quad \Delta \vdash n = \text{red} \quad \Xi' \vdash [a] : _ \rightarrow \mathcal{T} \quad \Xi' \vdash s : \text{trace}}{\Xi \vdash a \ s : \text{trace}} \text{ L-GETI}_1$$

$$\frac{a = n\langle \text{get}(v) \rangle? \quad \Delta \vdash n = \text{red} \quad \Xi \vdash s : \text{trace}}{\Xi \vdash a \ s : \text{trace}} \text{ L-GETI}_2$$

Θ Δ  $\Xi \not\models n$ $\Theta \vdash n : [T]^+ = \perp$ $\Theta \vdash n : [T]^+ = v$





Promises: main problem

- promises can be passed around⁴
- requirement: write-once discipline.

⇒ each promise must be bound at-most once

- writing twice: write-error

⁴like first-class futures, as well.

“linear” type system

- write-once, read-many
- unfulfilled promise = consumable **resource**, bind consumes it

⇒ resource-aware (linear) type system

- **data-flow** analysis
- types

$$n : [T]^{+-} \quad \text{and} \quad n : [T]^+ \quad \text{and} \quad n : [T]^+ = v \quad (1)$$

- 2 typical rules

“linear” type system

- write-once, read-many
- unfulfilled promise = consumable **resource**, bind consumes it

⇒ resource-aware (linear) type system

- **data-flow** analysis
- types

$$n : [T]^{+-} \quad \text{and} \quad n : [T]^+ \quad \text{and} \quad n : [T]^+ = v \quad (1)$$

- 2 typical rules

$$\frac{\begin{array}{c} \Gamma; \Delta, n:[T]^+ \vdash o : c \quad \Gamma; \Delta, n:[T]^+ \vdash c : [(\dots, l:\vec{T} \rightarrow T, \dots)] \\ \Gamma; \Delta, n:[T]^+ \vdash v_i : T_i \quad \acute{\Gamma}; \acute{\Delta} = \Gamma; \Delta \setminus (\vec{v} : \vec{T}) \end{array}}{\Gamma; \Delta, \textcolor{red}{n} : [\textcolor{red}{T}]^{+-} \vdash \textit{bind } o.l(\vec{v}) : T \hookrightarrow n : [T]^+ :: \acute{\Gamma}; \acute{\Delta}, \textcolor{red}{n} : [\textcolor{red}{T}]^+} \text{T-BIND}$$

“linear” type system

- write-once, read-many
- unfulfilled promise = consumable **resource**, bind consumes it

⇒ resource-aware (linear) type system

- **data-flow** analysis
- types

$$n : [T]^{+-} \quad \text{and} \quad n : [T]^+ \quad \text{and} \quad n : [T]^+ = v \quad (1)$$

- 2 typical rules

$$\frac{\Gamma_1; \Delta_1 \vdash v_1 : T_1 \quad \Gamma_1; \Delta_1 \vdash v_2 : T_1 \quad \Gamma_1; \Delta_1 \vdash e_1 : T_2 :: \Gamma_2; \Delta_2 \quad \Gamma_2; \Delta_1 \vdash e_2 : T_2 :: \Gamma_2; \Delta_2}{\Gamma_1; \Delta_1 \vdash \text{if } v_1 = v_2 \text{ then } e_1 \text{ else } e_2 : T_2 :: \Gamma_2; \Delta_2} \text{ T-COND}$$

“linear” type system

- write-once, read-many
- unfulfilled promise = consumable **resource**, bind consumes it

⇒ resource-aware (linear) type system

- **data-flow** analysis
- types

$$n : [T]^{+-} \quad \text{and} \quad n : [T]^+ \quad \text{and} \quad n : [T]^+ = v \quad (1)$$

- 2 typical rules
further: sending a promise ⇒ loosing the permission ...

Introduction

Futures and promises

Interface description

Results & conclusion

Results

- proposing an extension of Creol with promises (on top of first-class futures)
- type system for futures, especially **resource aware** (linear) type system for promises
- standard soundness results (subject reduction, ...)
- formulation of an **open semantics** plus characterization of **possible interface behavior** by abstracting the environment
- **soundness** of the abstractions

Related work

- Hoare-logic of *Creol* plus first-class futures (but not promises) in [6]
- λ -calc. with futures + promises [10], *Alice ML* [11] [8]
- *safe* futures (for *FJ*) [12]: transparent use of futures

Future work

- Mock-testing framework
- common semantics of multi-threading / futures, comparison
- full abstraction and observational theory

References I

- [1] E. Ábrahám, A. Grüner, and M. Steffen.
Abstract interface behavior of object-oriented languages with monitors.
In R. Gorrieri and H. Wehrheim, editors, *FMOODS '06*, volume 4037 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 2006.
- [2] Arvind, R. S. Nikhil, and K. K. Pingali.
I-structures: Data-structures for parallel computing.
ACM Transactions on Programming Languages and Systems, 11(4):598–632, 1989.
- [3] H. Baker and C. Hewitt.
The incremental garbage collection of processes.
ACM Sigplan Notices, 12:55–59, 1977.
- [4] D. Caromel.
Service, asynchrony and wait-by-necessity.
Journal of Object-Oriented Programming, 2(4):12–22, Nov. 1990.
- [5] D. Caromel.
Towards a method of object-oriented concurrent programming.
Communications of the ACM, 36(9):90–102, Sept. 1993.
- [6] F. S. de Boer, D. Clarke, and E. B. Johnsen.
A complete guide to the future.
In R. de Nicola, editor, *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria.*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.
- [7] R. H. Halstead, Jr.
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, Oct. 1985.

References II

- [8] L. Kornstaedt.
Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language.
In Proceedings of the First Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Theoretical Computer Science, Sept. 2001.
- [9] B. Liskov and L. Shriram.
Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems.
SIGPLAN Notices, 23(7):260–267, 1988.
- [10] J. Niehren, J. Schwinghammer, and G. Smolka.
A concurrent lambda-calculus with futures.
Theoretical Computer Science, 64(3):338–356, Nov. 2006.
- [11] A. Rossberg, D. L. Botland, G. Tack, T. Brunklaus, and G. Smolka.
Alice through the looking glass.
In Vol. 5 of Trends in Functional Programming, chapter 6. Intellect Books, Bristol, 2006.
- [12] A. Welc, S. Jagannathan, and A. Hosking.
Safe futures in Java.
In Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '05. ACM, 2005.
In SIGPLAN Notices.