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

Erika Ábrahám¹, Immo Grabe², Andreas Grünér², and Martin Steffen³

Research Centre Jülich, Germany
Christian-Albrechts University Kiel, Germany
University of Oslo, Norway

NWPT'07, Oslo
Oct. 2007
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
- 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

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

\[
e ::= \ldots \mid \texttt{o@l(v, \ldots, v)} \mid \texttt{claim@}(n, o) \mid \texttt{get@n} \mid \ldots
\]
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.
\[ n\langle \text{let } x : T = v \text{ in } t \rangle \rightsquigarrow n\langle t[v/x] \rangle \quad \text{RED} \]

\[ c[(F, M)] \parallel n\langle \text{let } x : c = \text{new } c \text{ in } t \rangle \rightsquigarrow \]
\[ c[(F, M)] \parallel \nu(o : c). (\quad o[c, F, \bot] \parallel n\langle \text{let } x : c = o \text{ in } t \rangle \quad ) \quad \text{NEWO} \]
\[ n\langle\text{let } x: [T] = o@l(\vec{v}) \text{ in } t \rangle \leadsto \nu(n': [T]). (n\langle\text{let } x: [T] = n' \text{ in } t \rangle \parallel n' \langle\text{let } x: T = \text{grab}(o); \ M.l(o)(\vec{v}) \text{ in release}(o); x \rangle) \]

\[ n_1\langle \vec{v} \rangle \parallel n_2\langle\text{let } x : T = \text{claim}@\langle n_1, o \rangle \text{ in } t \rangle \leadsto n_1\langle \vec{v} \rangle \parallel n_2\langle\text{let } x : T = \vec{v} \text{ in } t \rangle \]

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

\[
n_1\langle v\rangle \parallel n_2\langle\text{let } x : T = \text{claim}@ (n_1, o) \text{ in } t\rangle \leadsto n_1\langle v\rangle \parallel n_2\langle\text{let } x : T = v \text{ in } t\rangle \quad \text{CLAIM}_1
\]

\[
\frac{t_2 \neq v}{n_2\langle t_2\rangle \parallel n_1\langle\text{let } x : T = \text{claim}@ (n_2, o) \text{ in } t'_1\rangle \leadsto n_2\langle t_2\rangle \parallel n_1\langle\text{let } x : T = \text{release}(o); \text{get}@ n_2 \text{ in } \text{grab}(o); t'_1\rangle} \quad \text{CLAIM}_2
\]
\[
\begin{align*}
n\langle \text{let } x : [T] = o \@ l(\vec{v}) \text{ in } t \rangle & \rightsquigarrow \\
\nu(n' : [T]). (n\langle \text{let } x : [T] = n' \text{ in } t \rangle \parallel n'\langle \text{let } x : T = \text{grab}(o); M.l(o)(\vec{v}) \text{ in release}(o); x \rangle)
\end{align*}
\]

\[
\begin{align*}
n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = \text{claim}@ (n_1, o) \text{ in } t \rangle & \rightsquigarrow n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t \rangle \quad \text{CLAIM}\]
\end{align*}
\]

\[
\begin{align*}
t_2 \neq v
\end{align*}
\]

\[
\begin{align*}
n_2\langle t_2 \rangle \parallel n_1\langle \text{let } x : T = \text{claim}@ (n_2, o) \text{ in } t'_1 \rangle & \rightsquigarrow \\
n_2\langle t_2 \rangle \parallel n_1\langle \text{let } x : T = \text{release}(o); \text{get}@ n_2 \text{ in grab}(o); t'_1 \rangle
\end{align*}
\]

\[
\begin{align*}
n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = \text{get}@ n_1 \text{ in } t \rangle & \rightsquigarrow n_1\langle v \rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t \rangle \quad \text{GET}\]
\end{align*}
\]
\[ n\langle \text{suspend}(o); t \rangle \leadsto n\langle \text{release}(o); \text{grab}(o); t \rangle \quad \text{SUSPEND} \]
\[ o[c, F, \bot] \parallel n\langle \text{grab}(o); t \rangle \xrightarrow{\tau} o[c, F, \top] \parallel n\langle t \rangle \quad \text{GRAB} \]
\[ o[c, F, \top] \parallel n\langle \text{release}(o); t \rangle \xrightarrow{\tau} o[c, F, \bot] \parallel n\langle t \rangle \quad \text{RELEASE} \]
Claiming a future

\[ t_2 \neq v \]

\[ t_2 = v \]

\[ claim \]

\[ release \]

\[ get \]

\[ grab \]

\[ \bot \]
Futures and promises

- terminology is not so clear
- relation to handled futures
- promises [9], I-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 it\(^1\)

- good for delegation

\(^1\)as in for async. calls
Syntax (promise)

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

\[
e ::= \ldots | \text{promise } T | \text{bind } o.l(\vec{v}) : T \leftrightarrow n \mid \ldots
\]

\(^2\)or a handle to the future.
\[ n' \langle \text{let } x : T' = \text{promise } T \text{ in } t \rangle \leadsto \nu(n : T') . (n' \langle \text{let } x : T' = n \text{ in } t \rangle) \]

\[ \ldots n_1 \langle \text{let } x : T = \text{bind } o.l(\vec{v}) : T_2 \leftrightarrow n_2 \text{ in } t_1 \rangle \xrightarrow{\tau} \]
\[ \ldots n_1 \langle \text{let } x : T = n_2 \text{ in } t_1 \rangle \]
\[ \| (n_2 \langle \text{let } x : T_2 = \text{grab}(o); M.l(o)(\vec{v}) \text{ in } \text{release}(o); x \rangle) \]
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: 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 $release(n)$

---

$^{3}$user level
Syntax

\[
C ::= 0 | C \parallel C | \nu(n:T).C | n[O] | n[n, F, L] | n\langle t \rangle \\
O ::= F, M \\
M ::= l = m, \ldots, l = m \\
F ::= l = f, \ldots, l = f \\
m ::= \varsigma(n:T).\lambda(x:T, \ldots, x:T).t \\
f ::= \varsigma(n:T).\lambda().v | \varsigma(n:T).\lambda().\perp_n' \\
t ::= v \mid \text{stop} \mid \text{let } x:T = e \text{ in } t \\
e ::= t \mid \text{if } v = \nu \text{ then } e \text{ else } e \mid \text{if } \text{undef}(v.l()) \text{ then } e \text{ else } e \\
\mid \text{promise } T \mid \text{bind } n.l(\bar{\nu}) : T \leftrightarrow n \mid \text{get } v \leftrightarrow n \mid v.l() \mid v.l := \varsigma(s:n).\lambda().v \\
\mid \text{new } n \mid \text{claim}(n, n) \mid \text{get}(n) \mid \text{suspend}(n) \mid \text{grab}(n) \mid \text{release}(n) \\
v ::= x \mid n \mid () \\
L ::= \perp \mid T
\]
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 \( C \xrightarrow{t} \hat{C} \)?
  - rather: consider \( C \) in a context / environment
    \[ C \parallel E \xrightarrow{t} \hat{C} \parallel \hat{E} \]
    for some environment \( E \)
  - open semantics
    \[ \Delta \vdash C : \Theta \xrightarrow{t} \hat{\Delta} \vdash C : \hat{\Theta} \]
- assumptions \( \Delta \) abstracts environments \( E \)
One step further: legal traces

- open semantics

\[ \Delta \vdash C : \Theta \xrightarrow{t} \hat{\Delta} \vdash C : \hat{\Theta} \]

abstracts the environment

- existential abstraction of component, as well:

- characterization of *principally possible* interface behavior

\[ C \parallel E \xrightarrow{t} \dot{C} \parallel \dot{E} \]

for *some* component \( C \) + *some* environment \( E \)

\[ \Rightarrow \text{legal trace} \]

\[ \Delta \vdash t : \text{trace} :: \Theta \]
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
\[\Xi \vdash \epsilon : \text{trace} \quad \text{L-EMPTY}\]

\[
a = \nu(\Xi'). n\langle \text{call o.l}(\bar{v})\rangle? \quad \Xi' = \Xi + a \quad (\Xi' \vdash n \lor \Delta \vdash n: [0^+] \wedge) \quad \Xi \vdash o.l? : \bar{T} \rightarrow T \quad \Xi \vdash [a] : \bar{T} \rightarrow \_ \quad \Xi \vdash s : \text{trace}
\]

\[\Xi \vdash a s : \text{trace}\]

\[
a = \nu(\Xi'). n\langle \text{get}(v)\rangle? \quad \Xi' = \Xi + a \quad \Delta \vdash n = \bot \quad \Xi \vdash [a] : \_ \rightarrow T \quad \Xi \vdash s : \text{trace}
\]

\[\Xi \vdash a s : \text{trace} \quad \text{L-GETI}_1\]

\[
a = n\langle \text{get}(v)\rangle? \quad \Delta \vdash n = v \quad \Xi \vdash s : \text{trace}
\]

\[\Xi \vdash a s : \text{trace} \quad \text{L-GETI}_2\]
\[ \Theta \vdash n \]
\[ \Xi \not\vdash n \]
\[ \Theta \vdash n : [T]^+ = \bot \]
\[ \Theta \vdash n : [T]^+ = \nu \]
\[ \Theta \vdash n : [T]^+ = v \]
\[ \Theta \vdash n \vdash n: [T]^+ = \bot \]

\[ \Theta \vdash n: [T]^+ = \nu \]
Promises: main problem

• promises can be passed around

• requirement: write-once discipline.

⇒ each promise must be bound at-most once

• writing twice: write-error

\footnote{\textit{like 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

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

- 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

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

• 2 typical rules

\[
\begin{align*}
\Gamma; \Delta, n: [T]^+ & \vdash o : c & & \Gamma; \Delta, n: [T]^+ & \vdash c : [\ldots, l: \bar{T} \rightarrow T, \ldots] \\
\Gamma; \Delta, n: [T]^+ & \vdash v_i : T_i & & \Gamma; \acute{\Delta} = \Gamma; \Delta \setminus (\bar{v} : \bar{T}) \\
\Gamma; \Delta, n : [T]^{+-} & \vdash \text{bind} \ o. l(\bar{v}) : T & & \Gamma; \acute{\Delta}, n: [T]^+ \\
\end{align*}
\]
“linear” type system

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

⇒ resource-aware (linear) type system

- data-flow analysis

- types

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

- 2 typical rules

\[ \Gamma_1; \Delta_1 \vdash v_1 : T_1 \quad \Gamma_1; \Delta_1 \vdash v_2 : T_1 \]

\[ \Gamma_1; \Delta_1 \vdash e_1 : T_2 :: \Gamma_2; \Delta_2 \quad \Gamma_2; \Delta_1 \vdash e_2 : T_2 :: \Gamma_2; \Delta_2 \]

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

Abstract interface behavior of object-oriented languages with monitors.

I-structures: Data-structures for parallel computing.

The incremental garbage collection of processes.

Service, asynchrony and wait-by-necessity.

Towards a method of object-oriented concurrent programming.

A complete guide to the future.

Multilisp: A language for concurrent symbolic computation.
Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. 

Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. 

A concurrent lambda-calculus with futures. 
Preprint submitted to TCS.

Alice through the looking glass. 

Safe futures in Java. 
In SIGPLAN Notices.