Abstract interface behavior of an object-oriented language with futures and promises

Erika Ábrahám[1], Immo Grabe[2], Andreas Grüner[2], and Martin Steffen[3]

Albert-Ludwigs-University Freiburg, Germany

Christian-Albrechts University Kiel, Germany

University of Oslo, Norway

Credo-meeting, Amsterdam 2007

Sept. 2007

Structure

Introduction

Futures and promises

Related work

λ-calculus with futures

Open systems

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

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)

Futures

- introduced in the concurrent Multilisp language [10]:
- proposed by Baker & Hewitt [3]
- originally: transparent concurrency compiler annotation
- `future e`:
  - evaluated potentially in in parallel with the rest ⇒ 2 threads (producer and consumer)
  - future variable dynamically generated
  - *e* evaluated: future identified with value
- lazy cons [9], also ACT-1
- wait-by-necessity [4] [5]

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:[1] claiming the future + suspension points

$$claim@(n, o) \quad \text{and} \quad suspend(n)$$

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

$$grab(n) \quad \text{and} \quad release(n)$$

[1] user level

Syntax

$$
\begin{array}{rcl}
C & ::= & \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).}C \mid n[\![O]\!] \mid \underline{n[n, F, L]} \mid \underline{n\langle t\rangle} \\
O & ::= & F, M \\
M & ::= & l = m, \ldots, l = m \\
F & ::= & l = f, \ldots, l = f \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t \\
f & ::= & \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'} \\
t & ::= & v \mid \mathrm{stop} \mid \mathrm{let}\ x{:}T = e\ \mathrm{in}\ t \\
e & ::= & t \mid \mathrm{if}\ v = v\ \mathrm{then}\ e\ \mathrm{else}\ e \mid \mathrm{if}\ \mathit{undef}(v.l())\ \mathrm{then}\ e\ \mathrm{else}\ e \\
 & \mid & v@l(v, \ldots, v) \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v \\
 & \mid & \mathrm{new}\ n \mid \mathit{claim}@(n, n) \mid \underline{\mathit{get}@n} \mid \mathit{suspend}(n) \mid \underline{\mathit{grab}(n)} \mid \underline{\mathit{release}(n} \\
v & ::= & x \mid n \mid () \\
L & ::= & \bot \mid \top
\end{array}
$$

Type system

- judgments of the form (component level)

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

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

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![(\ldots, l{:}\vec{T} \to T, \ldots)]\!] \quad \Gamma;\Delta \vdash v_i : T_i}{\Gamma;\Delta \vdash v@l(\vec{v}) : [T]} \text{ T-CALLA}$$

$$\frac{\Gamma;\Delta \vdash n : [T]}{\Gamma;\Delta \vdash claim@(n, o) : T} \text{ T-CLAIM} \qquad \frac{\Gamma;\Delta \vdash n : [T]}{\Gamma;\Delta \vdash get@n : T} \text{ T-GET}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash suspend(o) : \text{Unit}} \text{ T-SUSPEND} \qquad \frac{}{\Gamma;\Delta \vdash grab(o) : \text{Unit}} \text{ T-GRAB} \qquad \frac{}{\Gamma;\Delta}$$

Operational semantics

- strict separation of internal and external behavior

⇒ 2 stages
  - internal steps (modulo congruence)
  - external, interface steps (labelled transitions)

- internal one: rather standard OS.

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

$n\langle \text{let}\, x_2{:}T_2 = (\text{let}\, x_1{:}T_1 = e_1 \,\text{in}\, e) \,\text{in}\, t\rangle \rightsquigarrow n\langle \text{let}\, x_1{:}T_1 = e_1 \,\text{in}\, (\text{let}\, x_2{:}T_2 = e \,\text{in}\, t)\rangle$ LET

$n\langle \text{let}\, x{:}T = (\text{if}\, v = v \,\text{then}\, e_1 \,\text{else}\, e_2) \,\text{in}\, t\rangle \rightsquigarrow n\langle \text{let}\, x{:}T = e_1 \,\text{in}\, t\rangle$ COND_1

$n\langle \text{let}\, x{:}T = (\text{if}\, v_1 = v_2 \,\text{then}\, e_1 \,\text{else}\, e_2) \,\text{in}\, t\rangle \rightsquigarrow n\langle \text{let}\, x{:}T = e_2 \,\text{in}\, t\rangle$ COND_2

$n\langle \text{let}\, x{:}T = (\text{if}\, \text{undef}(\bot_{c'}) \,\text{then}\, e_1 \,\text{else}\, e_2) \,\text{in}\, t\rangle \rightsquigarrow n\langle \text{let}\, x{:}T = e_1 \,\text{in}\, t\rangle$ $\text{COND}_1^{\bot}$

$n\langle \text{let}\, x{:}T = (\text{if}\, \text{undef}(v) \,\text{then}\, e_1 \,\text{else}\, e_2) \,\text{in}\, t\rangle \rightsquigarrow n\langle \text{let}\, x{:}T = e_2 \,\text{in}\, t\rangle$ $\text{COND}_2^{\bot}$

$n\langle \text{let}\, x{:}T = \text{stop} \,\text{in}\, t\rangle \rightsquigarrow n\langle \text{stop}\rangle$ STOP

$o[c, F', L] \parallel n\langle \text{let}\, x{:}T = o.l() \,\text{in}\, t\rangle \xrightarrow{\tau}$

$o[c, F', L] \parallel n\langle \text{let}\, x{:}T = F'.l(o)() \,\text{in}\, t\rangle$ FLOOKUP

$o[c, F, L] \parallel n\langle \text{let}\, x{:}T = o.l := v \,\text{in}\, t\rangle \xrightarrow{\tau} o[c, F.l := v, n'] \parallel n\langle \text{let}\, x{:}T = o \,\text{in}\, t\rangle$ FUPDATE

$c[\![F, M]\!] \parallel n\langle \text{let}\, x{:}c = \text{new}\, c \,\text{in}\, t\rangle \rightsquigarrow$

$c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F, \bot] \parallel n\langle \text{let}\, x{:}c = o \,\text{in}\, t\rangle)$ NEWO_i

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

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

$$\frac{t_2 \neq v}{\begin{array}{l} n_2\langle t_2\rangle \parallel n_1\langle \text{let } x : T = claim@(n_2, o) \text{ in } t_1'\rangle \rightsquigarrow \\ \qquad n_2\langle t_2\rangle \parallel n_1\langle \text{let } x : T = release(o); get@n_2 \text{ in } grab(o); t_1'\rangle \end{array}} \text{ CLAIM}_i^2$$

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

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

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

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

Claiming a future

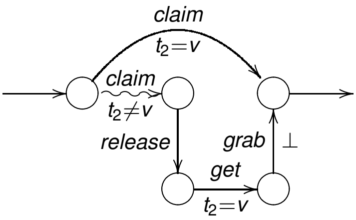

Figure: Claiming a future

Busy waiting

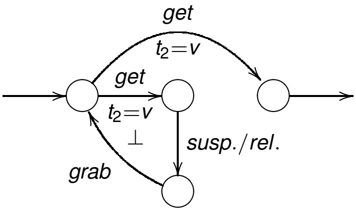

Figure: Claiming a future (busy wait)

Introduction

Futures and promises

Related work

λ-calculus with futures

Open systems

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 $C \stackrel{t}{\Longrightarrow} \acute{C}$?
  - rather: consider C in a context / environment

$$C \parallel E \underset{\bar{t}}{\stackrel{t}{\Longrightarrow}} \acute{C} \parallel \acute{E}$$

for some environment E

⇒ open semantics

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash C : \acute{\Theta}$$

- assumptions Δ abstracts environments E

One step further: legal traces

- open semantics

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash C : \acute{\Theta}$$

abstracts the environment

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

$$C \parallel E \underset{\bar{t}}{\stackrel{t}{\Longrightarrow}} \acute{C} \parallel \acute{E}$$

for some component C + some environment E

$\Rightarrow$ legal trace

$$\Delta \vdash t : \mathit{trace}$$

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

$$\frac{\begin{array}{c} a = \nu(\Xi').\ n\langle \mathit{call}\ o.l(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + a \quad \Xi' \vdash n \\ \acute{\Xi} \vdash o.l? : \vec{T} \to \_ \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_ \quad \acute{\Xi} \vdash s : \mathit{trace} \end{array}}{\Xi \vdash a\ s : \mathit{trace}} \ \text{L-CALLI}$$

$$\frac{\begin{array}{c} a = \nu(\Xi').n\langle \mathit{get}(v)\rangle? \quad \acute{\Xi} = \Xi + a \quad \Delta \vdash n : [T]^{+} = \bot \\ \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T \quad \acute{\Xi} \vdash s : \mathit{trace} \end{array}}{\Xi \vdash a\ s : \mathit{trace}} \ \text{L-GETI}_1$$

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

Remark

- simpler are for multi-threading
- can be seen as a special case
- no re-entrancy
- context-free description vs. memory-less

$$\Xi \vdash r \triangleright s : \mathit{trace}$$

- comparison to Java monitors: *much* simpler

Futures and promises

- terminology is not so clear
- relation to handled futures
- promises [12], I-structures [2]
- special case of logic variables

⇒ 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[2]
- good for delegation

[2] as in for async. calls

Syntax (promise)

- instead of $o@l(\vec{v})$
- split into
  1. create a promise[3]
  2. fullfill the promise = bind code to it.

$$e \quad ::= \quad \ldots \mid \textit{promise}\ T \mid \textit{bind}\ o.l(\vec{v}) : T \hookrightarrow n \mid \underline{\textit{get}\ v \mapsto n} \mid \ldots$$

[3] or a handle to the future

Main problem

- promises can be passed around [4]
- requirement: write-once discipline.

⇒ Each promise must be bound at-most once

- writing twice: write-error 1

[4] like first-class futures.

Semantics, interface behavior

- rest of the story: quite similar (only a bit trickier)
  - interal semantics: straighforward[5]
  - interface description: very little difference by adding promises

[5] The (quite) tricky bit is: formulate a “good” way to split assumptions and commitments.

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

$$c[\![(F', M)]\!] \parallel o[c, F, I] \parallel n_1\langle \text{let } x{:}T = bind\ o.l(\vec{v}) : T_2 \hookrightarrow n_2 \text{ in } t_1\rangle \xrightarrow{\tau}$$
$$c[\![(F', M)]\!] \parallel o[c, F, I] \parallel n_1\langle \text{let } x{:}T = n_2 \text{ in } t_1\rangle$$
$$\parallel \nu(n'{:}\text{Unit}).(n'\langle \text{let } x{:}T_2 = grab(o); M.l(o)(\vec{v}) \text{ in } release(o);$$

$$n'\langle get\ v \mapsto n_1\rangle \parallel n_2\langle \text{let } x : T = claim@(n_1, o) \text{ in } t\rangle \rightsquigarrow n'\langle get\ v \mapsto n_1\rangle \parallel n_2\langle$$

$$\frac{t_2 \neq v}{\begin{array}{l} n'\langle get\ t_2 \mapsto n_2\rangle \parallel n_1\langle \text{let } x : T = claim@(n_2, o) \text{ in } t_1'\rangle \rightsquigarrow \\ \quad n'\langle get\ t_2 \mapsto n_2\rangle \parallel n_1\langle \text{let } x : T = release(o); get@n_2 \text{ in } grab(o); t_1'\rangle \end{array}} \ \text{CLAIM}_i^4$$

$$n'\langle get\ v \mapsto n_1\rangle \parallel n_2\langle \text{let } x : T = get@n_1 \text{ in } t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \text{let } x : T = v \text{ in } t$$

Promises vs. handled futures

- basically the same
- write-once linear type discipline

⇒ handled futures better

Syntax comparison

- difference: thread creation: where comes the code from
- threadlets here a executed inside an object

Handled futures, I-structures, and promises

- also: handled futures
- safe concurrent access
- write-once
- I-Var from dataflow languages, in *Id* and also *Concurrent ML*
- form of restricted logic variable
- associated with future

Related work

- [13], *Alice ML*[14] [11]

Futures in *Java* etc

- *safe* futures (for *FJ*) [16]: transparent use of futures

Introduction

Futures and promises

Related work

λ-calculus with futures

Open systems

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

Future work

- testing
- common semantics of multi-threading / futures, comparison
- comparison with monitor -semantics

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] S. Conchon and F. L. Fessant.
JoCaml: mobile agents for Objective Caml.
In *First International Symposium on Agent Systems and Applications ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages 22–29, 1999.

[7] 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*. Springer-Verlag, 2007.

References II

[8] C. Fournet and G. Gonthier.
Th join calculus: A language for distributed mobile programming.
In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *APPSEM 2000*, volume 2395, 2002.

[9] D. Friedman and D. Wise.
CONS should not evaluate its arguments.
Communications of the ACM, 1976.

[10] R. H. Halstead, Jr.
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, Oct. 1985.

[11] 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 Mulit-Language Infrastructure and Interoperability (BABEL'01)*, Electronic Notes in Theoretical Computer Science, Sept. 2001.

[12] B. Liskov and L. Shrira.
Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems.
SIGPLAN Notices, 23(7):260–267, 1988.

[13] J. Niehren, J. Schwinghammer, and G. Smolka.
A concurrent lambda-calculus with futures.
Theoretical Computer Science, 2006.
Preprint submitted to TCS.

[14] 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.

References III

[15] J. Schwinghammer.
A concurrent λ-calculus with promises and futures.
Diplomarbeit, Universität des Saarlandes, Feb. 2002.

[16] 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*.