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

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

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

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

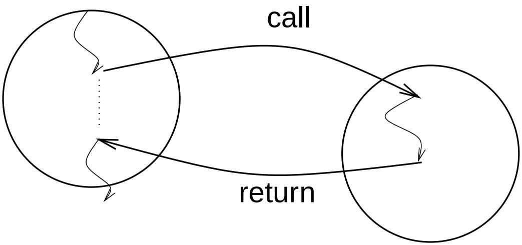
Interface description

Results & conclusion

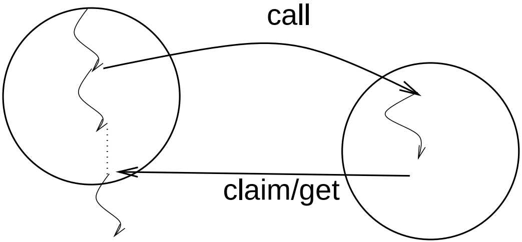
Futures

- 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) +

$$e \quad ::= \quad \ldots \mid o@l(v,\ldots,v) \mid claim@(n,o) \mid \underline{get@n} \mid \ldots$$

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.

$$n\langle \mathsf{let}\, x{:}T = v \;\mathsf{in}\; t\rangle \rightsquigarrow n\langle t[v/x]\rangle \qquad \textsc{Red}$$

$$c[\![(F, M)]\!] \parallel n\langle \mathsf{let}\, x{:}c = \mathsf{new}\, c \;\mathsf{in}\; t\rangle \rightsquigarrow$$
$$c[\![(F, M)]\!] \parallel \nu(o{:}c).(\quad o[c, F, \bot] \parallel n\langle \mathsf{let}\, x{:}c = o \;\mathsf{in}\; t\rangle \quad) \qquad \textsc{NewO}_i$$

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

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

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

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

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

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

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

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

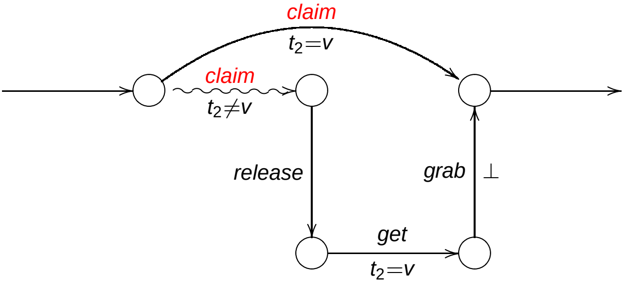
$$n_1\langle v\rangle \parallel n_2\langle \text{let}\, x : T = get@n_1\, \text{in}\, t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \text{let}\, x : T = v\, \text{in}\, t\rangle \qquad \text{GET}_i$$

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

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

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

Claiming a future



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.

$$e \quad ::= \quad \ldots \mid \mathit{promise}\ T \mid \mathit{bind}\, o.l(\vec{v}) : T \hookrightarrow n \mid \ldots$$

[2] or a handle to the future.

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

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

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

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

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

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

[3] user level

Syntax

$$
\begin{array}{rcll}
C & ::= & \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).}C \mid n[\![O]\!] \mid \underline{n[n, F, L]} \mid \underline{n\langle t\rangle} & \text{program} \\
O & ::= & F, M & \text{object} \\
M & ::= & l = m, \ldots, l = m & \text{method suite} \\
F & ::= & l = f, \ldots, l = f & \text{fields} \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t & \text{method} \\
f & ::= & \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'} & \text{field} \\
t & ::= & v \mid \text{stop} \mid \text{let}\, x{:}T = e \,\text{in}\, t & \text{thread} \\
e & ::= & t \mid \text{if}\, v = v \,\text{then}\, e \,\text{else}\, e \mid \text{if}\, \mathit{undef}(v.l()) \,\text{then}\, e \,\text{else}\, e & \text{expr.} \\
 & \mid & \mathit{promise}\ T \mid \mathit{bind}\ n.l(\vec{v}) : T \hookrightarrow n \mid \underline{\mathit{get}\ v \mapsto n} \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v & \\
 & \mid & \text{new}\, n \mid \mathit{claim}@(n, n) \mid \underline{\mathit{get}@n} \mid \mathit{suspend}(n) \mid \underline{\mathit{grab}(n)} \mid \underline{\mathit{release}(n)} & \\
v & ::= & x \mid n \mid () & \text{values} \\
L & ::= & \bot \mid \top & \text{lock status}
\end{array}
$$

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

- existential 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

⇒ 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

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-EMPTY} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \qquad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-PAR} \qquad \frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \text{T-NU}$$

$$\frac{;\Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NCLASS} \qquad \frac{;\Delta \vdash c : [\![T_F, T_M]\!] \quad ;\Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F, l] : (o{:}c)}\ \text{T-NOBJ}$$

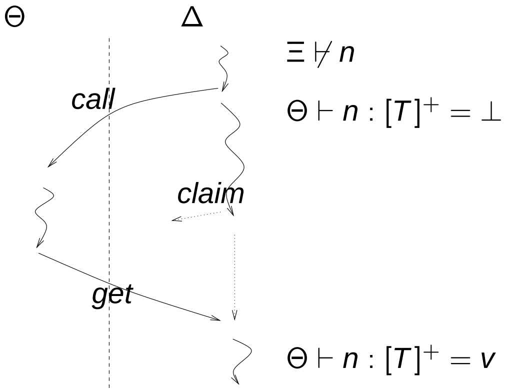
$$\frac{;\lfloor\Delta\rfloor, n{:}[T]^+ \vdash t : T}{\Delta \vdash n\langle t\rangle : (n{:}[T]^+)}\ \text{T-NTHREAD} \qquad \frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-SUB}$$

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

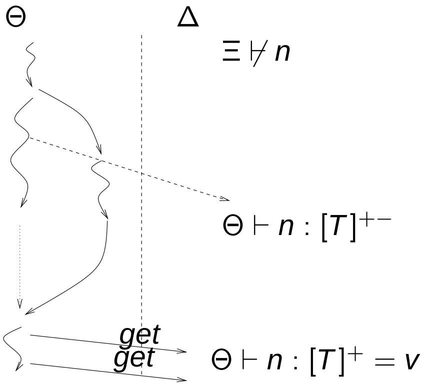
$$\frac{a = \nu(\Xi').\ n\langle \mathit{call}\ o.l(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + a \quad (\Xi' \vdash n \vee \Delta \vdash n : [\,]^{+-}) \quad \acute{\Xi} \vdash o.l? : \vec{T} \to T \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_ \quad \acute{\Xi} \vdash s : \mathit{trac}}{\Xi \vdash a\ s : \mathit{trace}}$$

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

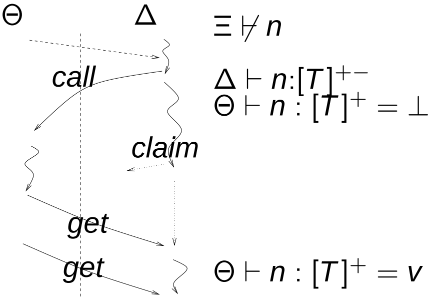
$$\frac{a = n\langle \mathit{get}(v)\rangle? \quad \Delta \vdash n = v \quad \Xi \vdash s : \mathit{trace}}{\Xi \vdash a\ s : \mathit{trace}} \text{ L-GETI}_2$$



Θ
Δ
$\Xi \not\vdash n$
call
$\Theta \vdash n : [T]^+ = \bot$
claim
get
$\Theta \vdash n : [T]^+ = v$



Θ
Δ
Ξ ⊬ n
Θ ⊢ n : [T]+−
get
get
Θ ⊢ n : [T]+ = v


Θ
Δ
Ξ ⊬ n
call
Δ ⊢ n:[T]+−
Θ ⊢ n : [T]+ = ⊥
claim
get
get
Θ ⊢ n : [T]+ = v

Promises: main problem

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

⇒ each promise must be bound at-most once

- writing twice: write-error

[4] 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

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

- 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

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

- 2 typical rules

$$\frac{\Gamma;\Delta, n{:}[T]^{+} \vdash o : c \quad \Gamma;\Delta, n{:}[T]^{+} \vdash c : [\![\ldots, l{:}\vec{T} \rightarrow T, \ldots]\!] \qquad \Gamma;\Delta, n{:}[T]^{+} \vdash v_i : T_i \quad \acute{\Gamma};\acute{\Delta} = \Gamma;\Delta \setminus (\vec{v} : \vec{T})}{\Gamma;\Delta, n : [T]^{+-} \vdash \mathit{bind}\ o.l(\vec{v}) : T \hookrightarrow n : [T]^{+} :: \acute{\Gamma};\acute{\Delta}, n{:}[T]^{+}} \ \text{T-BIND}$$

“linear” type system

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

⇒ resource-aware (linear) type system

- data-flow analysis
- types

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

- 2 typical rules

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

“linear” type system

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

⇒ resource-aware (linear) type system

- data-flow analysis
- types

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

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

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

[9] B. Liskov and L. Shrira.
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, 2006.
Preprint submitted to TCS.

[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*.