Funcons for threads and processes

Peter D. Mosses

Swansea University (emeritus)
TU Delft (visitor)

WG 2.2 meeting, September 2018
Brno, Czech Republic

CBS: Component-Based Semantics

Main goal:

Make formal semantics as popular as BNF !

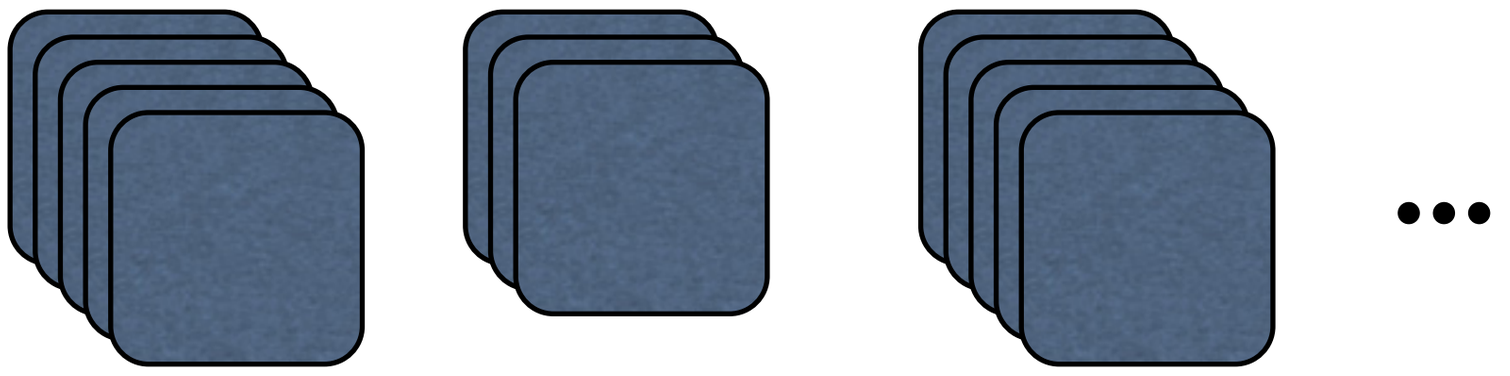
Encourage language developers *to use formal semantics:*

- *documentation* of language features, design decisions
- *generation* of (prototype) implementations

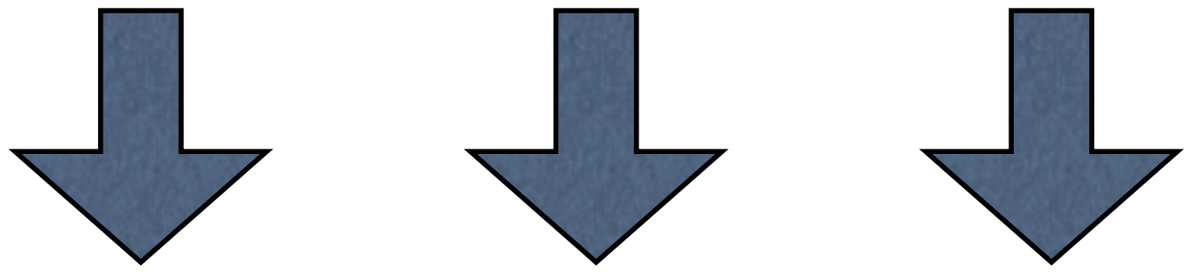
Component-based semantics

evolving

programming languages

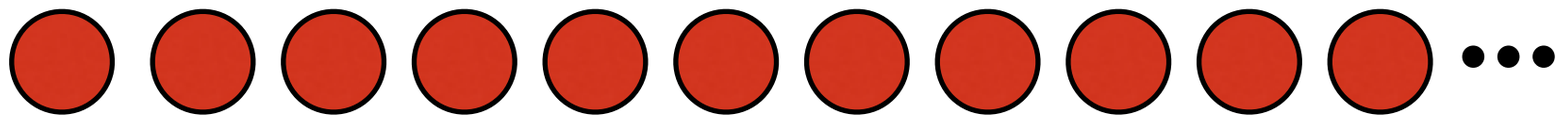


translation



stable reusable components

funcons



open-ended repository

Conjecture

Using ***component-based*** semantics can significantly reduce the effort of language specification

… leading to improved programming languages

CBS beta-release

plancomps.github.io/CBS-beta

Funcons-beta currently under **review**

- *those funcons will then be **fixed** (more can be added)*

Languages-beta illustrates CBS and use of Funcons-beta

- *simple languages: IMP, SIMPLE, SL*
- *sub-languages: MiniJava, OCaml Light*
- *language specifications **may evolve***

Concurrency concepts

Threads: shared state

- synchronisation (mutexes, condition variables, barriers), scheduling, weak/strong atomicity, POSIX/OpenMP, …

Processes: separate state

- asynchrony, message-passing, channels, rendezvous, MPI, …
- heavyweight (most OS) or lightweight (e.g., Erlang)

Logical Methods in Computer Science
Vol. 6 (4:2) 2010, pp. 1–39
www.lmcs-online.org

Submitted Sep. 17, 2010
Published Oct. 20, 2010

A MODEL OF COOPERATIVE THREADS *

MARTÍN ABADI [a] AND GORDON D. PLOTKIN [b]

[a] Microsoft Research, Silicon Valley; University of California, Santa Cruz
e-mail address: abadi@microsoft.com

[b] Microsoft Research, Silicon Valley; LFCS, University of Edinburgh
e-mail address: gdp@inf.ed.ac.uk

Abstract. We develop a model of concurrent imperative programming with threads. We focus on a small imperative language with cooperative threads which execute without interruption until they terminate or explicitly yield control. We define and study a trace-based denotational semantics for this language; this semantics is fully abstract but mathematically elementary. We also give an equational theory for the computational effects that underlie the language, including thread spawning. We then analyze threads in terms of the free algebra monad for this theory.

Syntax for threads [Abadi & Plotkin]

$$
\begin{array}{rcll}
b & \in & \text{BExp} & = \quad \ldots \\
e & \in & \text{NExp} & = \quad \ldots \\
C, D & \in & \text{Com} & = \quad \texttt{skip} \\
 & & & \mid \quad x := e \qquad (x \in \text{Vars}) \\
 & & & \mid \quad C; D \\
 & & & \mid \quad \texttt{if } b \texttt{ then } C \texttt{ else } D \\
 & & & \mid \quad \texttt{while } b \texttt{ do } C \\
 & & & \mid \quad \texttt{async } C \\
 & & & \mid \quad \texttt{yield} \\
 & & & \mid \quad \texttt{block}
\end{array}
$$

$\texttt{async } x := 0;$
$x := 1;$
$\texttt{yield};$
$\texttt{if } x = 0 \texttt{ then skip else block};$
$x := 2$

Reduction semantics [Abadi & Plotkin]

$$
\begin{array}{rcrcl}
\Gamma & \in & \text{State} & = & \text{Store} \times \text{ComSeq} \times \text{Com} \\
\sigma & \in & \text{Store} & = & \text{Vars} \to \text{Value} \\
n & \in & \text{Value} & = & \mathbb{N} \\
T & \in & \text{ComSeq} & = & \text{Com}^*
\end{array}
$$

$$
\mathcal{E} \;=\; [\,] \mid \mathcal{E}; C
$$

$$
\begin{array}{lcll}
\langle \sigma, T, \mathcal{E}[x := e] \rangle & \longrightarrow & \langle \sigma[x \mapsto n], T, \mathcal{E}[\texttt{skip}] \rangle & (\text{if } \; \sigma(e) = n) \\
\langle \sigma, T, \mathcal{E}[\texttt{skip}; C] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[C] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[C] \rangle & (\text{if } \; \sigma(b) = \texttt{true}) \\
\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[D] \rangle & (\text{if } \; \sigma(b) = \texttt{false}) \\
\langle \sigma, T, \mathcal{E}[\texttt{while } b \texttt{ do } C] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } (C; \texttt{while } b \texttt{ do } C) \texttt{ else skip}] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{async } C] \rangle & \longrightarrow & \langle \sigma, T.C, \mathcal{E}[\texttt{skip}] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{yield}] \rangle & \longrightarrow & \langle \sigma, T.\mathcal{E}[\texttt{skip}], \texttt{skip} \rangle & \\
\langle \sigma, T.C.T', \texttt{skip} \rangle & \longrightarrow & \langle \sigma, T.T', C \rangle &
\end{array}
$$

Reduction semantics [Abadi & Plotkin]

"Despite some subtleties, this semantics is not meant to be challenging."

Implicit:

- initial state: $\langle\ \sigma, (\ ), C\ \rangle$
- stuck states: $\langle\ \sigma, (), \texttt{skip}\ \rangle$ $\langle\ \sigma, T, \mathcal{E}[\texttt{block}]\ \rangle$
  - implications of "normal" and "abnormal" termination!
- no scheduling: arbitrary choice of thread on yield

Semantics of Transactional Memory and Automatic Mutual Exclusion

MARTÍN ABADI
Microsoft Research, University of California, Santa Cruz, and Collège de France
and
ANDREW BIRRELL, TIM HARRIS, and MICHAEL ISARD
Microsoft Research

Software Transactional Memory (STM) is an attractive basis for the development of language features for concurrent programming. However, the semantics of these features can be delicate and problematic. In this article we explore the trade-offs semantic simplicity, the viability of efficient implementation strategies, and the flexibility of language constructs. Specifically, we develop semantics and type systems for the constructs of the Automatic Mutual Exclusion (AME) programming model; our results apply also to other constructs, such as atomic blocks. With this semantics as a point of reference, we study several implementation strategies. We model STM systems that use in-place update, optimistic concurrency, lazy conflict detection, and rollback. These strategies are correct only under nontrivial assumptions that we identify and analyze. One important source of errors is that some efficient implementations create dangerous "zombie" computations where a transaction keeps running after experiencing a conflict; the assumptions confine the effects of these computations.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Languages, Theory

Additional Key Words and Phrases: Atomicity, correctness

ACM Reference Format:

Abadi, M., Birrell, A., Harris, T., and Isard, M. 2010. Semantics of transactional memory and automatic mutual exclusion. ACM Trans. Program. Lang. Syst. 33, 1, Article 2 (January 2011), 50 pages. DOI = 10.1145/1889997.1889999 http://doi.acm.org/10.1145/1889997.1889999

Syntax for IMP+threads in CBS

```
Syntax
C, D : com
   ::= 'skip'
     | var ':=' nexp
     | com ';' com
     | 'if' bexp 'then' com 'else' com
     | 'while' bexp 'do' com
     | 'async' com
     | 'yield'
     | 'block'

B : bexp
   ::= 'true' | 'false' | nexp '=' nexp

E : nexp
   ::= nat | var
```

Semantics for IMP in CBS

```
Semantics
  exec[[ _:com ]] : =>null-type
Rule
  exec[[ 'skip' ]] = null-value
Rule
  exec[[ X ':=' E ]] = assign(bound \"X\", eval[[ E ]])
Rule
  exec[[ C ';' D ]] = sequential(exec[[ C ]], exec[[ D ]])
Rule
  exec[[ 'if' B 'then' C 'else' D ]] =
  if-true-else(bval[[ B ]], exec[[ C ]], exec[[ D ]])
Rule
  exec[[ 'while' B 'do' C ]] = while-true(bval[[ B ]],
exec[[C ]])
```

Fundamental constructs for threads

Aims

- abstract from POSIX (Pthreads) details
  - ~~*efficiency*~~, ~~*scheduling*~~, *real time*, ~~*resource limits*~~, …
- exhibit required behaviour
  - *forks, shared data, atomicity, synchronisation, …*
- allow encoding of OpenMP constructs

Fundamental constructs for threads

Means

- labels on steps (using Modular SOS)
  - *indicate yielding, waiting, …*
- atomic synchronisation operations on variables
  - *locking mutexes, signalling conditions, barriers, …*
- data-race-freedom implies sequential consistency

Semantics for threads in CBS

```
Rule
  exec[[ 'async' C ]] =
    effect(thread-fork(
      thread({cooperative|->true}, closure exec[[ C ]])))
Rule
  exec[[ 'yield' ]] = thread-yield
Rule
  exec[[ 'block' ]] = fail // ???
```

Semantics for threads in CBS

```
Syntax
  START: start ::= com
Semantics
  start[[_:start]] : =>null-type
Rule
  start[[ C ]] =
    initialise-binding
    initialise-storing
    initialise-threading
    finalise-failing
    scope(declare-vars,
      thread-schedule(thread-fork(
        thread({cooperative|->true}, closure exec[[ C ]]))))

Funcon
  declare-vars : =>environments
   ~> bind("x", allocate-variable(natural-numbers))
```

Funcons for processes

Aims

- abstract from MPI details
  - ~~*efficiency*~~, ~~*scheduling*~~, *real time*, ~~*resource limits*~~, …
- exhibit required behaviour
  - *spawning, messaging, asynchrony, blocking, …*
- allow encoding of channels, rendezvous, etc.

Funcons for processes

Means

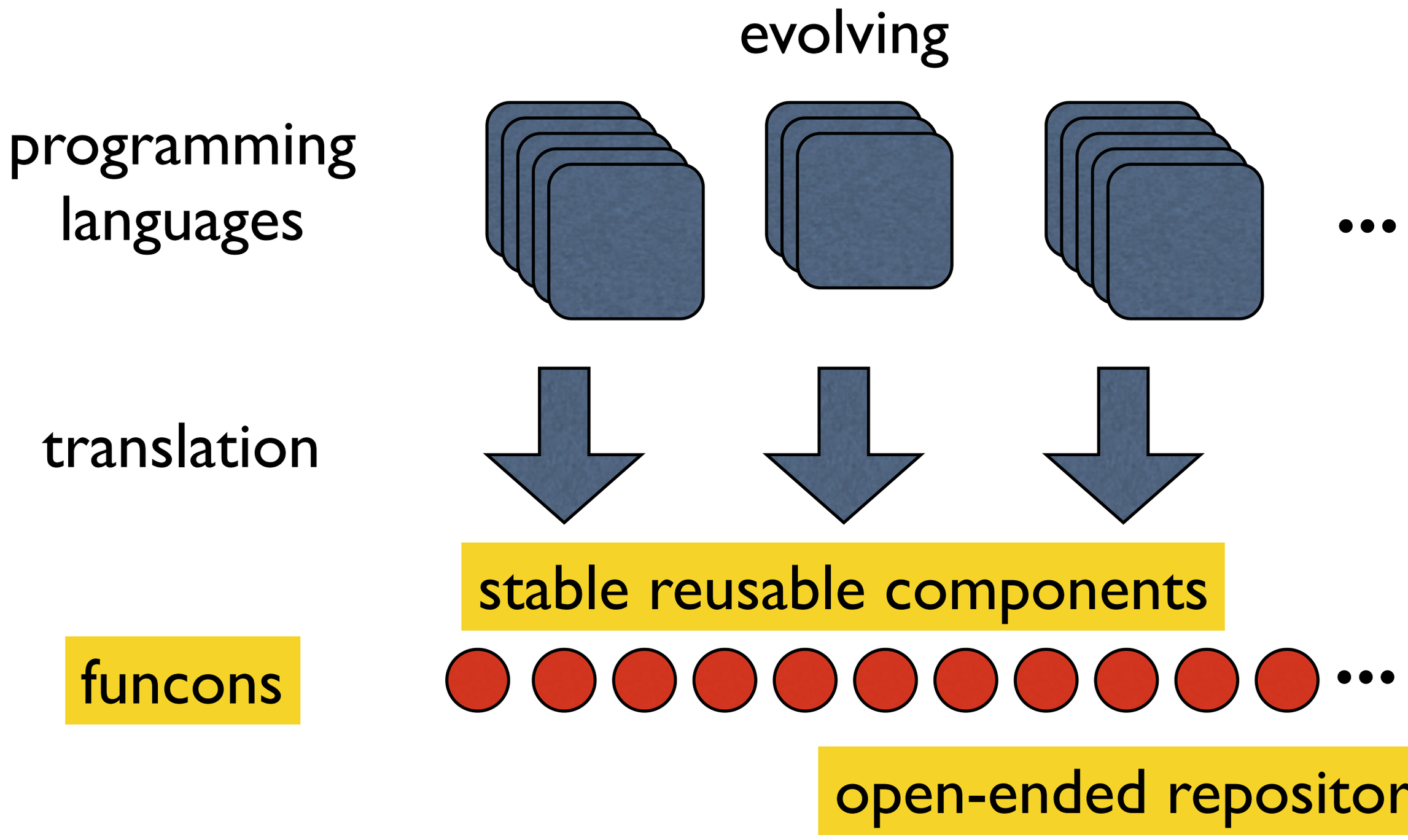
- Erlang-like process model (similar to Action Semantics)

  - *non-blocking message send*

  - *received message buffer*

  - *interleaving*

Component-based semantics



Conjecture

Using ***component-based*** semantics can significantly reduce the effort of language specification

… leading to improved programming languages

CBS beta-release

plancomps.github.io/CBS-beta

Funcons-beta currently under **review**

- *those funcons will then be **fixed** (more can be added)*

Languages-beta illustrates CBS and use of Funcons-beta

- *simple languages: IMP, SIMPLE, SL*
- *sub-languages: MiniJava, OCaml Light*
- *language specifications **may evolve***