Coinductive Big-Step Semantics for Concurrency [Extended Abstract]

Tarmo Uustalu
Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia
tarmo@cs.ioc.ee

In a paper presented at SOS 2010 [13], we developed a framework for big-step semantics for interactive input-output in combination with divergence, based on coinductive and mixed inductive-coinductive notions of resumptions, evaluation and termination-sensitive weak bisimilarity. In contrast to standard inductively defined big-step semantics, this framework handles divergence properly; in particular, runs that produce some observable effects and then diverge, are not "lost". Here we scale this approach for shared-variable concurrency on a simple example language. We develop the metatheory of this semantics in a constructive logic.

1 Introduction

The purpose of this paper is to advocate two ideas. First, big-step operational semantics can handle divergence as well as small-step semantics, so that both terminating and diverging behaviors can be reasoned about uniformly. Big-step semantics that account for divergence properly are achieved by working with coinductive semantic entities (transcripts of possible infinite computation paths or nonwellfounded computation trees) and coinductive evaluation. Second, contrary to what is so often stated, concurrency is not inherently small-step, or at least not more inherently than any kind of effect produced incrementally during a program's run (e.g., interactive output). Big-step semantics for concurrency can be built by borrowing the suitable denotational machinery, except that we do not want to use domains and fixpoints to deal with partiality, but coinductively defined sets and corecursion. In this paper, we use a version of resumptions, more specifically of coinductive resumptions. This datatype is a monad that accommodates both concurrency and divergence.

We build on our previous work [13] and develop a resumption-based big-step semantics with several variations for a simple imperative language with shared-variable concurrency. The metatheory of this semantics (e.g., the equivalence of evaluation in the big-step semantics to maximal multi-step reduction in a reference small-step semantics) is entirely constructive (which means that we can compute big-step evaluations from maximal multi-step reductions and vice versa); moreover, deterministic versions of evaluation are also computable functions.

The idea that divergence can be properly accounted for by switching to coinductively defined semantic entities such as possibly infinitely delayed states or possibly infinite traces is due to Capretta [3]. The deeper underlying theory is based on completely iterative monads and has been treated in detail by Goncharov and Schröder [7].

Leroy [9] attempted to use coinductive big-step semantics to reason about both terminating and diverging program runs in the Compcert project on a formally certified compiler, but ran into certain semantic anomalies (proving the big-step and small-step semantics equivalent required the use of excluded middle, which should not be needed; infinite loops were not specifically arranged to be productive, with the effect that infinite loops with no observable effects led to finite traces, to which other traces could be

Preliminary Report. Final version to appear in:
PLACES 2013

appended) (cf. also the simultaneous work by Cousot and Cousot [5]). Nakata and Uustalu [11] fixed the anomalies and arrived at a systematic account of trace-based big-step semantics for divergence in a purely sequential, side-effect-free setting (in relational and also functional styles). Further [12, 13], they also developed a matching Hoare logic and scaled the approach to a combination of interactive input/output with divergence. Danielsson [6] has strongly promoted especially functional-style coinductive big-step semantics. Ancona [2] used a coinductive big-step semantics of Java to show it type-sound in a sense that covers also divergence: if a program is type-sound, it produces a trace.

The tool of resumptions was originated by Plotkin [14] and has since been developed and used by several authors [4, 8]. An inductive trace-based big-step semantics for a concurrent language (not handling divergence) has appeared in the work of Mitchell [10].

2 An example language and semantics

2.1 Syntax

We look at a minimal language with shared-variable concurrency (cf. Amadio [1]) whose statements are given inductively by the grammar

$$s ::= x := e \mid \mathsf{skip} \mid s_0; s_1 \mid \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \mathsf{while}\ e\ \mathsf{do}\ s_t \mid s_0 \parallel s_1 \mid \mathsf{atomic}\ s \mid \mathsf{await}\ e\ \mathsf{do}\ s$$

The intention is that $s_0 \parallel s_1$ is parallel composition of s_0 and s_1 (in particular, it terminates when both branches have terminated). The statement $\mathsf{atomic}\ s$ is executed by running s atomically; the statement $\mathsf{await}\ e\ \mathsf{do}\ s$ is executed by waiting until e is true (other computations can have their chance in the meantime) and then running s atomically. Scheduling is preemptive, only assignments and boolean guards are atomic implicitly.

2.2 A resumption-based big-step semantics

The central semantic entities in our semantics are *resumptions* (computation trees). Resumptions are defined coinductively by the following rules (in this text, coinductive definitions are indicated by double rule-lines).

$$\frac{\sigma : \mathit{state}}{\mathit{ret}\ \sigma : \mathit{res}} \qquad \frac{r : \mathit{res}}{\delta\ r : \mathit{res}} \qquad \frac{r_0 : \mathit{res} \quad r_1 : \mathit{res}}{r_0 + r_1 : \mathit{res}} \qquad \frac{s : \mathit{stmt} \quad \sigma : \mathit{state}}{\mathit{yield}\ s\ \sigma : \mathit{res}}$$

The resumption $\mathit{ret}\ \sigma$ denotes a computation that terminated in a state σ. The resumption $\delta\ r$ is a computation that first produces an unit delay (makes an internal small step) and continues then as r. The resumption $r_0 + r_1$ is a choice between two resumptions r_0 and r_1. The resumption $\mathit{yield}\ s\ \sigma$ is a computation that has released control in a state σ and will further execute a statement s when (and if) it regains control. The definition being coinductive has the effect that resumptions can be non-wellfounded, i.e., computations can go on forever.

Evaluation of a statement s relates a (pre-)state to a (post-)resumption and is defined coinductively by the rules

$$\frac{}{x := e, \sigma \Rightarrow \delta\ (\mathit{ret}\ \sigma[x \mapsto [\![e]\!]\ \sigma])} \qquad \frac{}{\mathsf{skip}, \sigma \Rightarrow \mathit{ret}\ \sigma} \qquad \frac{s_0, \sigma \Rightarrow r \quad s_1, r \Rightarrow^{\mathrm{seq}} r'}{s_0; s_1, \sigma \Rightarrow r'}$$

$$\frac{\sigma \models e}{\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma \Rightarrow \delta\ (\mathit{yield}\ s_t\ \sigma)} \qquad \frac{\sigma \not\models e}{\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma \Rightarrow \delta\ (\mathit{yield}\ s_f\ \sigma)}$$

$$\frac{\sigma \models e}{\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma \Rightarrow \delta\ (\mathit{yield}\ (s_t; \mathsf{while}\ e\ \mathsf{do}\ s_t)\ \sigma)} \qquad \frac{\sigma \not\models e}{\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma \Rightarrow \delta\ (\mathit{ret}\ \sigma)}$$

$$\frac{s_0,\sigma \Rightarrow r_0 \quad s_1,r_0 \Rightarrow^{\mathrm{par}} r_0' \quad s_1,\sigma \Rightarrow r_1 \quad s_0,r_1 \Rightarrow^{\mathrm{par}} r_1'}{s_0 \parallel s_1,\sigma \Rightarrow r_0'+r_1'} \qquad \frac{s,\sigma \Rightarrow r \quad r \rightsquigarrow r'}{\mathsf{atomic}\ s,\sigma \Rightarrow r'}$$

$$\frac{\sigma \models e \quad s,\sigma \Rightarrow r \quad r \rightsquigarrow r'}{\mathsf{await}\ e\ \mathsf{do}\ s,\sigma \Rightarrow \delta\ r'} \qquad \frac{\sigma \not\models e}{\mathsf{await}\ e\ \mathsf{do}\ s,\sigma \Rightarrow \delta\ (\mathit{yield}\ (\mathsf{await}\ e\ \mathsf{do}\ s)\ \sigma)}$$

We have made sure that internal small steps take their time by inserting unit delays at all places where assignments or boolean guards are evaluated. The *yield*s in the rules for if and while signify control release points. Control release also occurs at the "midpoint" of evaluation of any sequential composition. This is handled by the base rule for sequential extension of evaluation (see below).

Sequential extension of evaluation relates a (pre-)resumption (the resumption present before some statement is evaluated) to a (post-)resumption (the total resumption after). It is defined coinductively by the rules

$$\frac{}{s,\mathit{ret}\ \sigma \Rightarrow^{\mathrm{seq}} \mathit{yield}\ s\ \sigma} \qquad \frac{s,r \Rightarrow^{\mathrm{seq}} r'}{s,\delta\ r \Rightarrow^{\mathrm{seq}} \delta\ r'} \qquad \frac{s,r_0 \Rightarrow^{\mathrm{seq}} r_0' \quad s,r_1 \Rightarrow^{\mathrm{seq}} r_1'}{s,r_0+r_1 \Rightarrow^{\mathrm{seq}} r_0'+r_1'} \qquad \frac{}{s,\mathit{yield}\ s_0\ \sigma \Rightarrow^{\mathrm{seq}} \mathit{yield}\ (s_0;s)\ \sigma}$$

Essentially, sequential extension of evaluation is a form of coinductive prefix closure of evaluation. But, in addition, the base case inserts a control release between the termination of the first statement and the start of the second statement of a sequential composition.

Parallel extension of evaluation, which also relates a resumption to a resumption, is for evaluating a given statement in parallel with a given resumption. The idea is to create an opportunity for the given statement to start when (and if) the resumption terminates or releases control. Also this relation is defined coinductively.

$$\frac{}{s,\mathit{ret}\ \sigma \Rightarrow^{\mathrm{par}} \mathit{yield}\ s\ \sigma} \qquad \frac{s,r \Rightarrow^{\mathrm{par}} r'}{s,\delta\ r \Rightarrow^{\mathrm{par}} \delta\ r'} \qquad \frac{s,r_0 \Rightarrow^{\mathrm{par}} r_0' \quad s,r_1 \Rightarrow^{\mathrm{par}} r_1'}{s,r_0+r_1 \Rightarrow^{\mathrm{par}} r_0'+r_1'} \qquad \frac{}{s,\mathit{yield}\ s_0\ \sigma \Rightarrow^{\mathrm{par}} \mathit{yield}\ (s_0 \parallel s)\ \sigma}$$

Finally, *closing a resumption* makes sure it does not release control. This is done by (repeatedly) "stitching" a resumption at every control release point by evaluating the residual statement from the state at this point. The corresponding relation between two resumptions is defined coinductively by

$$\frac{}{\mathit{ret}\ \sigma \rightsquigarrow \mathit{ret}\ \sigma} \qquad \frac{r \rightsquigarrow r'}{\delta\ r \rightsquigarrow \delta\ r'} \qquad \frac{r_0 \rightsquigarrow r_0' \quad r_1 \rightsquigarrow r_1'}{r_0+r_1 \rightsquigarrow r_0'+r_1'} \qquad \frac{s,\sigma \Rightarrow r \quad r \rightsquigarrow r'}{\mathit{yield}\ s\ \sigma \rightsquigarrow \delta\ r'}$$

To give only two smallest examples, for $s = x := 1 \parallel (x := x+2; x := x+2)$, $\sigma = [x \mapsto 0]$, we have $s,\sigma \Rightarrow \delta(\mathit{yield}\ (x := x+2; x := x+2)\ [x \mapsto 1]) + \delta(\mathit{yield}\ (x := 1 \parallel x := x+2)\ [x \mapsto 2])$ while $\mathsf{atomic}\ s,\sigma \Rightarrow \delta^5(\mathit{ret}\ [x \mapsto 5]) + \delta^2(\delta^3(\mathit{ret}\ [x \mapsto 3]) + \delta^3(\mathit{ret}\ [x \mapsto 1]))$. For $s = (\mathsf{await}\ x = 0\ \mathsf{do}\ x := 1) \parallel x := 2$, $\sigma = [x \mapsto 0]$, we have $s,\sigma \Rightarrow \delta^2(\mathit{yield}\ x := 2\ [x \mapsto 1]) + \delta^1(\mathit{yield}\ (\mathsf{await}\ x = 0\ \mathsf{do}\ x := 1)\ [x \mapsto 2])$ whereas $\mathsf{atomic}\ s,\sigma \Rightarrow \delta^4(\mathit{ret}\ [x \mapsto 2]) + \delta^\infty$.

To validate this semantics, we can relate it to a small-step semantics. We give a suitable definition in Appendix A.1. Evaluation in our big-step semantics agrees with the maximal multi-step reduction in the small-step semantics.

Since we collect the possible executions of a statement into a single computation tree and divergence is represented by infinite delays, evaluation is deterministic and total. This means that it can be turned into a function (so from the constructive point of view, evaluations can not only be checked, but also computed—which is only good of course). We show the corresponding definition in Appendix A.2.

It is also possible to work with a different (purely semantic) version of resumptions, requiring that the residual programs at control release points are evaluated. This leads to a form of giant-step semantics. We show the definition of the functional version of this semantics in Appendix A.3.[1]

[1] One might, of course argue, that what I have called the "big-step" semantics here should be called "medium-step", and the

2.3 Equivalences of resumptions

When are two resumptions to be considered equivalent? This depends on the purpose at hand. The finest sensible notion is *"very strong" bisimilarity* defined coinductively by the rules

$$\frac{}{\mathit{ret}\ \sigma \sim \mathit{ret}\ \sigma} \quad \frac{r \sim r_*}{\delta\ r \sim \delta\ r_*} \quad \frac{r_0 \sim r_{0*} \quad r_1 \sim r_{1*}}{r_0 + r_1 \sim r_{0*} + r_{1*}} \quad \frac{}{\mathit{yield}\ s\ \sigma \sim \mathit{yield}\ s\ \sigma}$$

(Classically, it is just equality of resumptions; but equality in intensional type theory is yet stronger.) Our big-step semantics is deterministic and agrees with the functional version in exactly this strongest meaningful sense.

The useful coarser notions evaluate residual statements of suspended resumptions and compare the results for bisimilarity; ignore order and multiplicity of choices (ordinary strong bisimilarity); or hide finite delays and/or choices altogether (termination-sensitive weak bisimilarity). The definition of termination-sensitive weak bisimilarity requires combining or mixing induction and coinduction, with several caveats to avoid. (First, it is easy to misdefine weak bisimilarity so that it equates any resumption with the divergent resumption and therefore all resumptions. Second, a fairly attractive definition fails to give reflexivity without the use of excluded middle, which is a warning that the definition is not the "right one" from the constructive point of view.) We refrain from giving the details here, but the key ideas are those from our SOS '10 paper.

2.4 A trace-based big-step semantics

A trace-based semantics is obtained from the resumption-based semantics by simply removing the + constructor of resumptions, splitting the evaluation rule for $\|$ into two rules (thereby turning evaluation nondeterministic) and removing the rules for + in the definitions of extended evaluations and closing. Because of the nondeterminism, trace-based evaluation cannot be turned into a function. But notice that it is still total. Any scheduling leads to a valid trace (differently from standard inductive big-step semantics, divergence from endless work or waiting does not lead to a "lost trace").

Differently from the case of trace-based big-step evaluation, trace-based giant-step evaluation must, in one way or another, use "guessing". A valid giant-step trace of a statement consists in a trace from a given pre-state to a control release state, followed by a trace of the residual statement from an arbitrary ("guessed") control grab state, etc. (unless the run terminates or diverges). When a statement is atomized, most of the traces constructed in this speculative way are thrown away.

3 Conclusion

We have shown that, with coinductive denotations and coinductive evaluation, it is possible to give simple and meaningful big-step descriptions of semantics of languages with concurrency. The key ideas remain the same as in the purely sequential case. Most importantly, due care must be taken of the possibilities of divergence. In particular, even diverging loops or await statements must be productive (by growing resumptions or traces by unit delays). Finite delays can then be equated by a suitable notion of weak bisimilarity.

"giant-step" semantics should be called "big-step". I would not disagree. My choice of terminology here was motivated by the intuition that "big-step" evaluation should run a statement to its completion. When a statement's run has reached a control release point, it is complete in the sense that it cannot run further on its own and it cannot be told what the scheduler will do with it (it might even be unfair and not return control to it at all). Note, however, that big-step and giant-step evaluation agree fully for statements of the form atomic *s*.

Although we could not delve into this topic here, our definitions and proofs benefit heavily from the fact that the datatype of resumptions is a monad, in fact a completely iterative monad, and moreover a free one (as long as we equate only very strongly bisimilar resumptions).

With Wolfgang Ahrendt and Keiko Nakata, we have developed a coinductive big-step semantics for ABS, an exploratory object-oriented language with an intricate concurrency model, developed in the FP7 ICT project HATS. ABS has cooperative scheduling of tasks (method invocations) communicating via shared memory (fields) within every object and preemptive scheduling of objects communicating via asynchronous method calls and futures. This work will be reported elsewhere.

Acknowledgements This research was supported by the EU FP7 ICT project HATS.

References

[1] R. Amadio (2012): *Operational methods for concurrency*. Draft lecture notes.

[2] D. Ancona (2012): *Soundness of object-oriented languages with coinductive big-step semantics*. In: J. Noble (ed.) *Proc. of 26th Europ. Conf. on Object-Oriented Programming, ECOOP 2012 (Beijing, June 2012). Lect. Notes in Comput. Sci.* 7313. Springer, Berlin, pp. 459–483.

[3] V. Capretta (2005): *General recursion via coinductive types*. *Log. Methods in Comput. Sci.* 1(2), article 1.

[4] P. Cenciarelli & E. Moggi (1993): *A syntactic approach to modularity in denotational semantics*. In: *Proc. of 5th Biennial Meeting on Category Theory and Computer Science, CTCS '93 (Amsterdam, Sept. 1993)*. Tech. report, CWI, Amsterdam.

[5] P. Cousot & R. Cousot (2009): *Bi-inductive operational semantics*. *Inf. and Comput.* 207(2), pp. 258–283.

[6] N. A. Danielsson (2012): *Operational semantics using the partiality monad*. In: *Proc. of 17th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '12 (Copenhagen, Sept. 2012)*. ACM Press, New York, pp. 127–138.

[7] S. Goncharov & L. Schröder (2011): *A coinductive calculus for asynchronous side-effecting processes*. In: O. Owe, M. Steffen & J. A. Telle (eds.) *Proc. of 18th Int. Symp. on Fundamentals of Computation Theory, FCT 2011 (Oslo, Aug. 2011). Lect. Notes in Comput. Sci.* 6914. Springer, Berlin, pp. 276–287.

[8] W. L. Harrison (2006): *The essence of multitasking*. In: M. Johnson & V. Vene (eds.) *Proc. of 11th Int. Conf. on Algebraic Methdology and Software Technology, AMAST 2006 (Kuressaare, July 2006). Lect. Notes in Comput. Sci.* 4019. Springer, Berlin, pp. 158–172.

[9] X. Leroy & H. Grall (2009): *Coinductive big-step operational semantics*. *Inf. and Comput.* 207(2), pp. 285–305.

[10] K. Mitchell (1994): *Concurrency in a natural semantics*. Report ECS-LFCS-94-311. Univ. of Edinburgh.

[11] K. Nakata & T. Uustalu (2009): *Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles*. In: S. Berghofer, T. Nipkow, C. Urban & M. Wenzel (eds.) *Proc. of 22nd Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2009 (Munich, Aug. 2009). Lect. Notes in Comput. Sci.* 5674. Springer, Berlin, pp. 375–390.

[12] K. Nakata & T. Uustalu (2010): *A Hoare logic for the coinductive trace-based big-step semantics of While*. In: A. D. Gordon (ed.) *Proc. of 19th Europ. Symp. on Programming, ESOP 2010 (Paphos, March 2010). Lect. Notes in Comput. Sci.* 6012. Springer, Berlin, pp. 488–506.

[13] K. Nakata & T. Uustalu (2010). *Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: an exercise in mixed induction-coinduction*. In: L. Aceto & P. Sobocinski (eds.) *Proc. of 7th Wksh. on Structural Operational Semantics, SOS 2010 (Paris, Aug. 2010). Electron. Proc. in Theor. Comput. Sci.* 32. Open Publishing Assoc., Sydney, pp. 57–75.

[14] G. D. Plotkin (1976): *A powerdomain construction*. *SIAM J. of Comput.* 5(3), pp. 452–487.

A Variations

A.1 A small-step semantics

To validate the big-step semantics of Sec. 2.2, we can compare it to a small-step semantics.

Here we give a small-step semantics that makes it possible to keep track of all runs of a statement at once.

Single-step reduction associates to a statement and state an extended configuration in a deterministic and total way (if the step amounts to making a choice, both alternatives are recorded). With maximal multi-step reduction, a statement's all runs are developed into a resumption.
Extended configurations (we use the notation of an inductive definition, but in fact this datatype is a simple disjoint union):

$$\frac{\sigma : state}{ret\ \sigma : xcfg} \quad \frac{s : stmt \quad \sigma : state}{\delta\ (s,\sigma) : xcfg} \quad \frac{s_0 : stmt \quad \sigma_0 : state \quad s_1 : stmt \quad \sigma_1 : state}{(s_0,\sigma_0) + (s_1,\sigma_1) : xcfg} \quad \frac{s : stmt \quad \sigma : state}{yield\ s\ \sigma : xcfg}$$

Resumptions:

$$\frac{\sigma : state}{\overline{ret\ \sigma : res}} \quad \frac{r : res}{\overline{\delta\ r : res}} \quad \frac{r_0 : res \quad r_1 : res}{\overline{r_0 + r_1 : res}} \quad \frac{s : stmt \quad \sigma : state}{\overline{yield\ s\ \sigma : res}}$$

Single-step reduction relates a state to an extended configuration and is defined inductively(!):

$$\frac{}{x := e, \sigma \to \delta\ (\mathsf{skip}, \sigma[x \mapsto [\![e]\!]\ \sigma])}$$

$$\frac{}{\mathsf{skip}, \sigma \to ret\ \sigma}$$

$$\frac{s_0, \sigma \to ret\ \sigma'}{s_0; s_1, \sigma \to yield\ s_1 \sigma'} \quad \frac{s_0, \sigma \to \delta\ (s_0', \sigma')}{s_0; s_1, \sigma \to \delta\ (s_0'; s_1, \sigma')}$$

$$\frac{s_0, \sigma \to (s_{00}, \sigma_0) + (s_{01}, \sigma_1)}{s_0; s_1, \sigma \to (s_{00}; s_1, \sigma_0) + (s_{01}; s_1, \sigma_1)} \quad \frac{s_0, \sigma \to yield\ s_0'\ \sigma'}{s_0; s_1, \sigma \to yield\ (s_0'; s_1)\ \sigma'}$$

$$\frac{\sigma \models e}{\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma \to \delta\ (\mathsf{skip}; s_t, \sigma)} \quad \frac{\sigma \not\models e}{\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, \sigma \to \delta\ (\mathsf{skip}; s_f, \sigma)}$$

$$\frac{\sigma \models e}{\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma \to \delta\ (\mathsf{skip}; (s_t; \mathsf{while}\ e\ \mathsf{do}\ s_t), \sigma)} \quad \frac{\sigma \not\models e}{\mathsf{while}\ e\ \mathsf{do}\ s_t, \sigma \to \delta\ (\mathsf{skip}, \sigma)}$$

$$\frac{}{s_0 \parallel s_1, \sigma \to (s_0 \mathbin{[\![} s_1, \sigma) + (s_1 \mathbin{[\![} s_0, \sigma)}$$

$$\frac{s_0, \sigma \to ret\ \sigma'}{s_0 \mathbin{[\![} s_1, \sigma \to yield\ s_1 \sigma'} \quad \frac{s_0, \sigma \to \delta\ (s_0', \sigma')}{s_0 \mathbin{[\![} s_1, \sigma \to \delta\ (s_0' \mathbin{[\![} s_1, \sigma')}$$

$$\frac{s_0, \sigma \to (s_{00}, \sigma_0) + (s_{01}, \sigma_1)}{s_0 \mathbin{[\![} s_1, \sigma \to (s_{00} \mathbin{[\![} s_1, \sigma_0) + (s_{01} \mathbin{[\![} s_1, \sigma_1)} \quad \frac{s_0, \sigma \to yield\ s_0'\ \sigma'}{s_0 \mathbin{[\![} s_1, \sigma \to yield\ (s_0' \parallel s_1)\ \sigma'}$$

$$\frac{s, \sigma \to ret\ \sigma'}{\mathsf{atomic}\ s, \sigma \to ret\ \sigma'} \quad \frac{s, \sigma \to \delta\ (s', \sigma')}{\mathsf{atomic}\ s, \sigma \to \delta\ (\mathsf{atomic}\ s', \sigma')}$$

$$\frac{s, \sigma \to (s_0, \sigma_0) + (s_1, \sigma_1)}{\mathsf{atomic}\ s, \sigma \to (\mathsf{atomic}\ s_0, \sigma_0) + (\mathsf{atomic}\ s_1, \sigma_1)} \quad \frac{s, \sigma \to yield\ s'\ \sigma'}{\mathsf{atomic}\ s, \sigma \to \delta\ (\mathsf{atomic}\ s', \sigma')}$$

$$\frac{\sigma \models e}{\mathsf{await}\ e\ \mathsf{do}\ s, \sigma \to \delta\ (\mathsf{atomic}\ s, \sigma)} \quad \frac{\sigma \not\models e}{\mathsf{await}\ e\ \mathsf{do}\ s, \sigma \to \delta\ (\mathsf{skip}; \mathsf{await}\ e\ \mathsf{do}\ s, \sigma)}$$

Notice that $\mathsf{skip}; s$ differs from s by allowing a control release before s is started. We have also used an auxiliary statement form $s_0 \mathbin{[\![} s_1$ (parallel composition, but s_0 makes the first small step).

Maximal multi-step reduction relates a state to a resumption and is defined coinductively:

$$\frac{s,\sigma \to \mathit{ret}\ s'}{s,\sigma \to^{\mathrm{m}} \mathit{ret}\ s'} \qquad \frac{s,\sigma \to \delta\ (s',\sigma') \quad s',\sigma' \to^{\mathrm{m}} r}{s,\sigma \to^{\mathrm{m}} \delta\ r}$$

$$\frac{s,\sigma \to (s_0,\sigma_0)+(s_1,\sigma_1) \quad s_0,\sigma_0 \to^{\mathrm{m}} r_0 \quad s_1,\sigma_1 \to^{\mathrm{m}} r_1}{s,\sigma \to^{\mathrm{m}} r_0+r_1} \qquad \frac{s,\sigma \to \mathit{yield}\ s'\ \sigma'}{s,\sigma \to^{\mathrm{m}} \mathit{yield}\ s'\ \sigma'}$$

Evaluation agrees with maximal multi-step reduction: $s,\sigma \Rightarrow r$ iff $s,\sigma \to^{\mathrm{m}} r$.

A.2 Functional version of the big-step semantics

That the evaluation relation of Sec. 2.2 is deterministic and total is proved by showing that it is a function. Here is an equational specification of this function that can be massaged into an honest definition by structural corecursion.

Resumptions:

$$\frac{\sigma : \mathit{state}}{\mathit{ret}\ \sigma : \mathit{res}} \qquad \frac{r : \mathit{res}}{\delta\ r : \mathit{res}} \qquad \frac{r_0 : \mathit{res} \quad r_1 : \mathit{res}}{r_0+r_1 : \mathit{res}} \qquad \frac{s : \mathit{stmt} \quad \sigma : \mathit{state}}{\mathit{yield}\ s\ \sigma : \mathit{res}}$$

Evaluation:

$$\begin{array}{llcl}
\mathit{eval} & (x := e) & \sigma & = \delta\ (\mathit{ret}\ \sigma[x \mapsto [\![e]\!]\ \sigma]) \\
\mathit{eval} & \mathsf{skip} & \sigma & = \mathit{ret}\ \sigma \\
\mathit{eval} & (s_0; s_1) & \sigma & = \mathit{evalseq}\ s_1\ (\mathit{eval}\ s_0\ \sigma) \\
\mathit{eval} & (\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f) & \sigma & = \mathit{if}\ [\![e]\!]\ \sigma\ \mathit{then}\ \delta\ (\mathit{yield}\ s_t\ \sigma)\ \mathit{else}\ \delta\ (\mathit{yield}\ s_t\ \sigma) \\
\mathit{eval} & (\mathsf{while}\ e\ \mathsf{do}\ s_t) & \sigma & = \mathit{if}\ [\![e]\!]\ \sigma\ \mathit{then}\ \delta\ (\mathit{yield}\ (s_t; \mathsf{while}\ e\ \mathsf{do}\ s_t)\ \sigma)\ \mathit{else}\ \delta\ (\mathit{ret}\ \sigma) \\
\mathit{eval} & (s_0 \parallel s_1) & \sigma & = \mathit{evalpar}\ s_1\ (\mathit{eval}\ s_0\ \sigma) + \mathit{evalpar}\ s_0\ (\mathit{eval}\ s_1\ \sigma) \\
\mathit{eval} & (\mathsf{atomic}\ s) & \sigma & = \mathit{close}\ (\mathit{eval}\ s\ \sigma) \\
\mathit{eval} & (\mathsf{await}\ e\ \mathsf{do}\ s) & \sigma & = \mathit{if}\ [\![e]\!]\ \sigma\ \mathit{then}\ \delta\ (\mathit{close}\ (\mathit{eval}\ s\ \sigma))\ \mathit{else}\ \delta\ (\mathit{yield}\ (\mathsf{await}\ e\ \mathsf{do}\ s)\ \sigma)
\end{array}$$

Sequential extension of evaluation:

$$\begin{array}{llcl}
\mathit{evalseq} & s & (\mathit{ret}\ \sigma) & = \mathit{yield}\ s\ \sigma \\
\mathit{evalseq} & s & (\delta\ r) & = \delta\ (\mathit{evalseq}\ s\ r) \\
\mathit{evalseq} & s & (r_0+r_1) & = \mathit{evalseq}\ s\ r_0 + \mathit{evalseq}\ s\ r_1 \\
\mathit{evalseq} & s & (\mathit{yield}\ s_0\ \sigma) & = \mathit{yield}\ (s_0; s)\ \sigma
\end{array}$$

Parallel extension of evaluation:

$$\begin{array}{llcl}
\mathit{evalpar} & s & (\mathit{ret}\ \sigma) & = \mathit{yield}\ s\ \sigma \\
\mathit{evalpar} & s & (\delta\ r) & = \delta\ (\mathit{evalpar}\ s\ r) \\
\mathit{evalpar} & s & (r_0+r_1) & = \mathit{evalpar}\ s\ r_0 + \mathit{evalpar}\ s\ r_1 \\
\mathit{evalpar} & s & (\mathit{yield}\ s_0\ \sigma) & = \mathit{yield}\ (s_0 \parallel s)\ \sigma
\end{array}$$

Closing a resumption:

$$\begin{array}{llcl}
\mathit{close} & (\mathit{ret}\ \sigma) & = \mathit{ret}\ \sigma \\
\mathit{close} & (\delta\ r) & = \delta\ (\mathit{close}\ r) \\
\mathit{close} & (r_0+r_1) & = \mathit{close}\ r_0 + \mathit{close}\ r_1 \\
\mathit{close} & (\mathit{yield}\ s\ \sigma) & = \delta\ (\mathit{close}\ (\mathit{eval}\ s\ \sigma))
\end{array}$$

Functional and relational evaluation agree: $\mathit{eval}\, s\, \sigma \sim r$ iff $s,\sigma \Rightarrow r$.

A.3 A giant-step semantics (functional version)

The giant-step semantics uses a different notion of resumptions requiring residual statements to be evaluated.

Resumptions:

$$\frac{\sigma : state}{ret\ \sigma : Res} \quad \frac{r : Res}{\delta\ r : Res} \quad \frac{r_0 : Res \quad r_1 : Res}{r_0 + r_1 : Res} \quad \frac{k : state \to Res \quad \sigma : state}{yield\ k\ \sigma : Res}$$

Evaluation:

$$\begin{array}{llcl}
Eval & (x := e) & \sigma & = & \delta\ (ret\ \sigma[x \mapsto [\![e]\!]\ \sigma]) \\
Eval & \mathsf{skip} & \sigma & = & ret\ \sigma \\
Eval & (s_0; s_1) & \sigma & = & Evalseq\ s_1\ (Eval\ s_0\ \sigma) \\
Eval & (\mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f) & \sigma & = & if\ [\![e]\!]\ \sigma\ then\ \delta\ (yield\ (Eval\ s_t)\ \sigma)\ else\ \delta\ (yield\ (Eval\ s_t)\ \sigma) \\
Eval & (\mathsf{while}\ e\ \mathsf{do}\ s_t) & \sigma & = & if\ [\![e]\!]\ \sigma\ then\ \delta\ (yield\ (Evalseq\ (\mathsf{while}\ e\ \mathsf{do}\ s_t) \circ Eval\ s_t)\ \sigma)\ else\ \delta\ (ret\ \sigma) \\
Eval & (s_0 \parallel s_1) & \sigma & = & Merge\ (Eval\ s_1)\ (Eval\ s_0\ \sigma) + Merge\ (Eval\ s_0)\ (Eval\ s_1\ \sigma) \\
Eval & (\mathsf{atomic}\ s) & \sigma & = & Close\ (Eval\ s\ \sigma) \\
Eval & (\mathsf{await}\ e\ \mathsf{do}\ s) & \sigma & = & if\ [\![e]\!]\ \sigma\ then\ \delta\ (Close\ (Eval\ s\ \sigma))\ else\ \delta\ (yield\ (Eval\ (\mathsf{await}\ e\ \mathsf{do}\ s))\ \sigma)
\end{array}$$

Sequential extension of evaluation:

$$\begin{array}{llcl}
Evalseq & s\ (ret\ \sigma) & = & yield\ (Eval\ s)\ \sigma \\
Evalseq & s\ (\delta\ r) & = & \delta\ (Evalseq\ s\ r) \\
Evalseq & s\ (r_0 + r_1) & = & Evalseq\ s\ r_0 + Evalseq\ s\ r_1 \\
Evalseq & s\ (yield\ k\ \sigma) & = & yield\ (Evalseq\ s \circ k)\ \sigma
\end{array}$$

Merge of a continuation into a resumption:

$$\begin{array}{llcl}
Merge & k\ (ret\ \sigma) & = & yield\ k\ \sigma \\
Merge & k\ (\delta\ r) & = & \delta\ (Merge\ k\ r) \\
Merge & k\ (r_0 + r_1) & = & Merge\ k\ r_0 + Merge\ k\ r_1 \\
Merge & k\ (yield\ k_0\ \sigma) & = & yield\ (\lambda\sigma'.\ Merge\ k\ (k_0\ \sigma') + Merge\ k_0\ (k\ \sigma'))\ \sigma
\end{array}$$

Closing a resumption:

$$\begin{array}{llcl}
Close & (ret\ \sigma) & = & ret\ \sigma \\
Close & (\delta\ r) & = & \delta\ (Close\ r) \\
Close & (r_0 + r_1) & = & Close\ r_0 + Close\ r_1 \\
Close & (yield\ k\ \sigma) & = & \delta\ (Close(k\ \sigma))
\end{array}$$