A Denotational Approach to Release/Acquire Concurrency

We want to report on recent and ongoing work into the denotational semantics of shared-state concurrent programming languages with Brookes-style trace semantics. Most of the talk would cover our trace semantics for the Release/Acquire (RA) memory model, a fragment of the C/C++ standard. Developing this semantics required us to re-think the interpretation of Brookes trace-sets, moving away from interrupted executions and towards a game-like/rely-guarantee-like intuition about the interaction of the program with its environment. We would like to present to the GALOP community these results and, time permitting, ongoing work about more general trace semantics for shared state, to facilitate discussion and further directions.

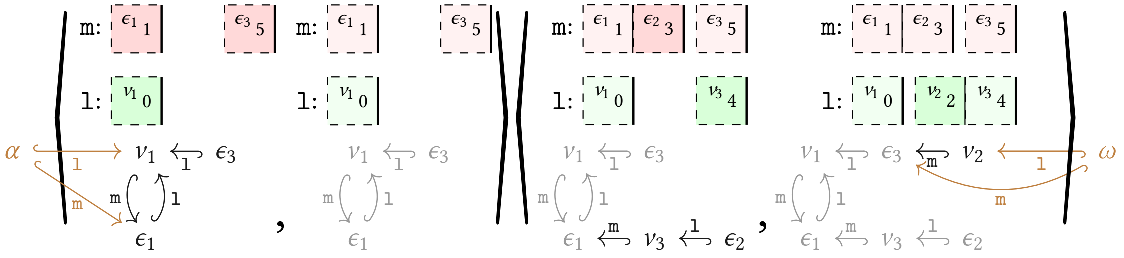
For shared-memory concurrent programming, the seminal work of Brookes [1996] defined a denotational semantics, where the denotation $[\![M]\!]$ is a set of totally ordered traces that consist of sequences of pairs of memory snapshots $\langle \mu_0, \varrho_0 \rangle \dots \langle \mu_n, \varrho_n \rangle$. Each sequence represents a behavior that the program fragment M may exhibit. In every pair $\langle \mu, \varrho \rangle$ in a trace, μ is the snapshot that M relies on the environment to provide; and ϱ is the snapshot that M guarantees to provide in return. The gaps between pairs represent possible interference by the environment. Working under the assumption of preemptive scheduling—imposing no restrictions on the interleaving of steps of execution between parallel threads—denotations are closed under the following two trace-rewriting operations which maintain the representation of possible behavior. *Stutter* adds a transition of the form $\langle \mu, \mu \rangle$ anywhere in the trace; a program fragment can always guarantee no changes between its actions. *Mumble* combines a couple of subsequent transitions of the form $\langle \mu, \varrho \rangle \langle \varrho, \theta \rangle$ into a single transition $\langle \mu, \theta \rangle$ anywhere in the trace; a program fragment can always rely on its own guarantees in the absence of observable interference from the environment.

A *memory model* describes how memory access by concurrently running threads is handled through a program's routine. Brookes established the adequacy of the trace-based denotational semantics w.r.t. the strongest operational semantics of shared-memory concurrent programs, known as *sequential consistency* (SC), where every memory access happens instantaneously and immediately affects all concurrent threads. Jagadeesan et al. [2012] closely followed Brookes to define denotational semantics for x86-TSO [Owens et al. 2009; Pulte et al. 2018]. Other weak memory models, in particular, models of *programming languages*, and *non-multi-copy-atomic* models, where writes can be observed by different threads in different orders, have so far been out of reach of Brookes's totally ordered traces, and were only captured by much more sophisticated models based on *partial orders* [Castellan 2016; Dodds et al. 2018; Jagadeesan et al. 2020; Jeffrey et al. 2022; Kavanagh and Brookes 2018; Paviotti et al. 2020]. In this work we target the Release/Acquire memory model. This model, obtained by restricting the C/C++11 memory model [Batty et al. 2011] to release/acquire atomics, is a well-studied fundamental memory model weaker than x86-TSO, which, roughly speaking, ensures "causal consistency" together with "per-location-SC" and "RMW (read-modify-write) atomicity" [Lahav 2019; Lahav et al. 2016]. These guarantees make RA sufficiently strong for implementing common synchronization idioms.

Our first contribution is a Brookes-style denotational semantics for RA. As Brookes's traces are totally ordered, this result may seem counterintuitive. The standard semantics for RA is a declarative (a.k.a. axiomatic) memory model, in the form of acyclicity consistency constraints over partially ordered candidate execution graphs. Since these partial orders are not totally ordered, one might expect that Brookes's traces are insufficient. Nevertheless, our first key observation is that an *operational* presentation of RA as an interleaving semantics of a weak memory system lends itself to Brookes-style semantics. We develop a notion of traces compatible with Kang et al.'s "view-based" machine [2017], an operational semantics that is equivalent to RA's declarative formulation. There, a threads writes by adding a message to memory. Once added, a message is never removed or modified. The memory associates a timeline to each location, on which messages are placed, each occupying a segment. Messages cannot overlap, but they can be placed adjacently. If the message originated in an RMW, the written message must be placed adjacently to the read message. This blocks another RMW from doing the same, thus enforcing atomicity. Each thread maintains a view that determines, for each location, what is the latest messages of which it was made aware on the timeline. A thread cannot read nor write messages prepending its last known message. To enforce causal consistency, each message records the view of the thread that wrote it. When a thread reads a message, it inherits its view, possibly making it aware of later messages.

Authors' addresses: Yotam Dvir, yotamdvir@mail.tau.ac.il, Tel Aviv University, Israel; Ohad Kammar, ohad.kammar@ed.ac.uk, University of Edinburgh, Scotland; Ori Lahav, orilahav@tau.ac.il, Tel Aviv University, Israel.

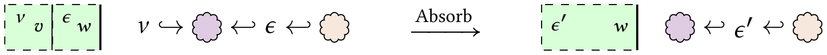
Below we illustrate an example trace, in the setting of two memory locations `l` and `m`. The trace has two pairs of memory snapshots, illustrated using two diagrams. Top: shows the messages per location as (possibly adjacent) segments on the location's timeline. Bottom: shows the graph structure induced by the views within messages. The only message added within a transition is ν_2, and thus it is the only *local* message: one that the program fragment guarantees to provide, rather than relies on receiving. The trace also has an initial view α and a final view ω, illustrated by showing the messages to which they point.



In the future, we hope to bring in nominal techniques to account for timestamps in a more principled manner. In particular, we would like to investigate the possibility of using a Fraïssé limit [Bojanczyk et al. 2012; Fraïssé 1986] for comparing names for (i) equality, (ii) order, and (iii) adjacency.

We prove several results about our denotational semantics. Soundness: for every interrupted execution there is a corresponding single-transition trace in the denotation. Fundamental Lemma: for every trace in the denotation there exists an interrupted execution of the program fragment exhibiting a related behavior. Adequacy: denotational approximation implies contextual refinement. An immediate practical application of adequacy is the ability to provide *local* formal justifications of program transformations, such as those performed by optimizing compilers. Formally justifying these transformations without the local analysis that denotational semantics provides is non-trivial [Dodds et al. 2018; Vafeiadis et al. 2015].

An important aspect of denotational semantics is its abstraction. As an external measure, we verify that our adequate semantics validates various transformations/optimizations: standard and structural transformations; algebraic laws of parallel programming; and all known thread-local RA-valid compiler transformations involving atomic RA memory accesses. This level of abstraction is achieved thanks to our denotations being closed not only under analogs to Brookes's stutter and mumble, but also several RA-specific operations. This allows us to relate programs which would naively correspond to rather different sets of traces. For example, we have the *Absorb* rewrite rule, which combines two adjacent local messages added within the same transition. Below we sketch how it modifies memory snapshots:



Figuratively, the preceding message ν is "absorbed" by the successor ϵ, thus becoming ϵ'. Nothing must point to the preceding message ν, so as to not leave dangling names. We use this rule when validating transformations which eliminate a write that is followed by another, such as `l` := v ; `l` := w $\twoheadrightarrow$ `l` := w.

Our second contribution is to connect the core semantics of parallel programming languages exhibiting weak behaviors to the more standard semantic account for sequential programming languages. Brookes presented his semantics for a simple imperative WHILE language, but Benton et al. [2016]; Dvir et al. [2022] later extended it to higher-order languages using Moggi's monad-based approach [1991].

A denotational semantics given in this monadic style comes ready-made with a rich semantic toolkit for program denotation [Benton et al. 2000], transformations [Benton et al. 2014, 2007, 2009; Benton and Leperchey 2005; Hofmann 2008], reasoning [Aguirre et al. 2022; Maillard et al. 2019], etc. We want to challenge, compare, and reuse this diverse toolkit in the concurrent setting. As a yardstick to the applicability of the monadic toolkit, we develop our semantics for a *higher-order functional language* with a general, first-class parallel composition operator. This is in contrast to most of the weak memory models research which employs imperative languages and assumes a single top-level parallel composition, but more in line with game models for concurrency [e.g. Castellan et al. 2017]. This puts weak memory models, which often require bespoke and highly specialized presentations, on a similar footing to many other programming effects.

REFERENCES

Alejandro Aguirre, Shin-ya Katsumata, and Satoshi Kura. 2022. Weakest preconditions in fibrations. *Mathematical Structures in Computer Science* 32, 4 (2022), 472–510. https://doi.org/10.1017/S0960129522000330

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *Proc. POPL*. ACM, 619–632.

Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent transformations for concurrent programs. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, James Cheney and Germán Vidal (Eds.). ACM, 188–201. https://doi.org/10.1145/2967973.2968602

Nick Benton, John Hughes, and Eugenio Moggi. 2000. Monads and Effects. In *APPSEM*. 42–122.

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *Proc. PPDP*. ACM, 87–96.

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based program transformations: higher-order store. In *Proc. PPDP*. ACM, 301–312.

Nick Benton and Benjamin Leperchey. 2005. Relational Reasoning in a Nominal Semantics for Storage. In *TLCA*. Springer, 86–101.

Mikolaj Bojanczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. 2012. Towards nominal computation. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 401–412. https://doi.org/10.1145/2103656.2103704

Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996), 145–163. https://doi.org/10.1006/inco.1996.0056

Simon Castellan. 2016. Weak memory models using event structures. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, Julien Signoles (Ed.). Saint-Malo, France. https://hal.inria.fr/hal-01333582

Simon Castellan, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. 2017. Games and Strategies as Event Structures. *Log. Methods Comput. Sci.* 13, 3 (2017). https://doi.org/10.23638/LMCS-13(3:35)2017

Mike Dodds, Mark Batty, and Alexey Gotsman. 2018. Compositional Verification of Compiler Optimisations on Relaxed Memory. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 1027–1055. https://doi.org/10.1007/978-3-319-89884-1_36

Yotam Dvir, Ohad Kammar, and Ori Lahav. 2022. An Algebraic Theory for Shared-State Concurrency. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 3–24. https://doi.org/10.1007/978-3-031-21037-2_1

R. Fraïssé. 1986. *Theory of Relations*. North-Holland. https://books.google.co.il/books?id=oFV1P6riLZ0C

Martin Hofmann. 2008. Correctness of effect-based program transformations. In *Formal Logical Methods for System Security and Correctness*, Orna Grumberg, Tobias Nipkow, and Christian Pfaller (Eds.). IOS Press, 149–173.

Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. https://doi.org/10.1145/3428262

Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes Is Relaxed, Almost!. In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7213)*, Lars Birkedal (Ed.). Springer, 180–194. https://doi.org/10.1007/978-3-642-28729-9_12

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498716

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. https://doi.org/10.1145/3009837.3009850

Ryan Kavanagh and Stephen Brookes. 2018. A Denotational Semantics for SPARC TSO. In *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018 (Electronic Notes in Theoretical Computer Science, Vol. 341)*, Sam Staton (Ed.). Elsevier, 223–239. https://doi.org/10.1016/j.entcs.2018.03.025

Ori Lahav. 2019. Verification under Causally Consistent Shared Memory. *ACM SIGLOG News* 6, 2 (apr 2019), 43–56. https://doi.org/10.1145/3326938.3326942

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 649–662. https://doi.org/10.1145/2837614.2837643

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2019. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (dec 2019), 33 pages. https://doi.org/10.1145/3371072

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 391–407. https:

//doi.org/10.1007/978-3-642-03359-9_27

Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 209–220. https://doi.org/10.1145/2676726.2676995

APPENDIX

The table below summarizes the transformations that we have validated using our denotational semantics. Some are given first using the general **rmw** construct, then specialized to loads (?) and well-known RMWs (CAS, FAA, XCHG). When a non-trivial closure rule (Ab, Ti, Di) is used for the denotational justification it appears above the symbol ↠.

Generalized Sequencing

$$(\mathbf{let}\,\mathrm{x} = M_1\,\mathbf{in}\,M_2) \parallel (\mathbf{let}\,\mathrm{y} = N_1\,\mathbf{in}\,N_2) \twoheadrightarrow \mathbf{match}\,M_1 \parallel N_1\ \mathbf{with}\,\langle \mathrm{x}, \mathrm{y}\rangle.\,M_2 \parallel N_2$$

Sequencing $M \parallel N \twoheadrightarrow \langle M, N\rangle$

Irrelevant Read Introduction $\langle\rangle \twoheadrightarrow \ell?\,;\langle\rangle$

Irrelevant Read Elimination $\ell?\,;\langle\rangle \twoheadrightarrow \langle\rangle$

Write-Write Elimination

$$\ell := w\,;\ell := v \ \overset{\mathsf{Ab}}{\twoheadrightarrow}\ \ell := v$$

Write-Read Deorder $(\ell \neq \ell')$

$$\left\langle \ell := v, \ell'?\right\rangle \ \overset{\mathsf{Ti}}{\twoheadrightarrow}\ \ell := v \parallel \ell'?$$

RMW Expansion $(\varphi_{\vec{v}} \leq \psi_{\vec{w}})$

$$\mathbf{rmw}_{\varphi}\,(\ell;\vec{v}) \ \overset{\mathsf{Di}}{\twoheadrightarrow}\ \mathbf{rmw}_{\psi}\,(\ell;\vec{w})$$

$$\ell? \ \overset{\mathsf{Di}}{\twoheadrightarrow}\ \mathrm{CAS}\,(\ell, v, v)$$

$$\mathrm{CAS}\,(\ell, v, v) \ \overset{\mathsf{Di}}{\twoheadrightarrow}\ \mathrm{FAA}\,(\ell, 0)$$

Atomic Store

$$\ell := v \ \twoheadrightarrow\ \mathrm{XCHG}\,(\ell, v)\,;\langle\rangle$$

Symmetric-Monoidal Laws, e.g.

$$M \parallel N \ \twoheadrightarrow\ \mathbf{match}\,N \parallel M\ \mathbf{with}\,\langle \mathrm{x}, \mathrm{y}\rangle.\ \langle \mathrm{y}, \mathrm{x}\rangle$$

Write-RMW Elimination

$$\ell := v\,;\mathbf{rmw}_{\varphi}\,(\ell;\vec{w}) \ \overset{\mathsf{Ab}}{\twoheadrightarrow}\ \ell := \varphi^{\mathrm{id}}_{\vec{w}}v\,;v$$

$$\ell := v\,;\ell? \ \twoheadrightarrow\ \ell := v\,;v$$

$$\ell := v\,;\mathrm{CAS}\,(\ell, v, u) \ \overset{\mathsf{Ab}}{\twoheadrightarrow}\ \ell := u\,;v$$

$$\ell := v\,;\mathrm{CAS}\,(\ell, w, u) \ \twoheadrightarrow\ \ell := v\,;v \qquad (v \neq w)$$

$$\ell := v\,;\mathrm{FAA}\,(\ell, w) \ \overset{\mathsf{Ab}}{\twoheadrightarrow}\ \ell := v + w\,;v$$

$$\ell := v\,;\mathrm{XCHG}\,(\ell, w) \ \overset{\mathsf{Ab}}{\twoheadrightarrow}\ \ell := w\,;v$$

RMW-Write Elimination $(\mathrm{dom}\,\psi_{\vec{w}} \supseteq \mathrm{dom}\,\varphi_{\vec{u}})$

$$\begin{aligned}&\mathbf{let}\,\mathrm{x} = \mathbf{rmw}_{\varphi}\,(\ell;\vec{u})\ \mathbf{in}\\ &\quad \mathbf{match}\,(\psi_{\vec{w}})\,\mathrm{x}\,\mathbf{with}\\ &\quad \{\iota_{\perp}\,\_.\mathrm{x} \mid \iota_{\top}\,v.\ell := v\,;\mathrm{x}\} \ \overset{\mathsf{Ab}}{\twoheadrightarrow}\ \mathbf{rmw}_{\psi}\,(\ell;\vec{w})\end{aligned}$$

$$\begin{aligned}&\mathbf{let}\,\mathrm{x} = \ell?\,\mathbf{in}\,(\mathbf{if}\,\mathrm{x} = v\\ &\quad \mathbf{then}\,\ell := w\,\mathbf{else}\,\langle\rangle\,)\,;\mathrm{x} \ \twoheadrightarrow\ \mathrm{CAS}\,(\ell, v, w)\end{aligned}$$

$$\mathbf{let}\,\mathrm{x} = \ell?\,\mathbf{in}\,\ell := \mathrm{x} + v\,;\mathrm{x} \ \twoheadrightarrow\ \mathrm{FAA}\,(\ell, v)$$

$$\mathbf{let}\,\mathrm{x} = \ell?\,\mathbf{in}\,\ell := v\,;\mathrm{x} \ \twoheadrightarrow\ \mathrm{XCHG}\,(\ell, v)$$

RMW-RMW Elimination $\left\langle \mathbf{rmw}_{\varphi}\,(\ell;\vec{v}), \mathbf{rmw}_{\psi}\,(\ell;\vec{w})\right\rangle \overset{\mathsf{Ab}}{\twoheadrightarrow} \mathbf{let}\,\mathrm{x} = \mathbf{rmw}_{\zeta}\,(\ell;\vec{u})\ \mathbf{in}\ \left\langle \mathrm{x}, \varphi^{\mathrm{id}}_{\vec{v}}\mathrm{x}\right\rangle \quad (\zeta_{\vec{u}} = \psi_{\vec{w}} \circ^{\mathrm{id}} \varphi_{\vec{v}})$

$$\langle \ell?, \ell?\rangle \twoheadrightarrow \mathbf{let}\,\mathrm{x} = \ell?\,\mathbf{in}\,\langle \mathrm{x}, \mathrm{x}\rangle \qquad \langle \mathrm{FAA}\,(\ell, v), \mathrm{FAA}\,(\ell, w)\rangle \twoheadrightarrow \mathbf{let}\,\mathrm{x} = \mathrm{FAA}\,(\ell, v + w)\ \mathbf{in}\ \langle \mathrm{x}, \mathrm{x} + v\rangle$$

$$\langle \ell?, \mathrm{CAS}\,(\ell, v, w)\rangle \twoheadrightarrow \mathbf{let}\,\mathrm{x} = \mathrm{CAS}\,(\ell, v, w)\ \mathbf{in}\ \langle \mathrm{x}, \mathrm{x}\rangle \qquad \langle \mathrm{XCHG}\,(\ell, w), \ell?\rangle \twoheadrightarrow \mathbf{let}\,\mathrm{x} = \mathrm{XCHG}\,(\ell, w)\ \mathbf{in}\ \langle \mathrm{x}, w\rangle$$