Operational game semantics for generative algebraic effects and handlers

Hamza Jaafar, Guilhem Jaber - Nantes Université, LS2N, Inria Gallinette

Algebraic effects originally appeared in the works of Power and Plotkin [12, 13], where languages with computational effects such as global state, exceptions and non-determinism are seen as algebraic theories with carriers being the domain of computations and effects being the operations described by their arity and the equations they satisfy. Algebraic operations and handlers [14] is a more recent approach to programming with effects, it is a generalization of exception handlers in the sense that when a performed effect is caught by the enclosing handler, the latter also gains access to the delimited continuation. It offers a modular approach in dealing with effects, a seamless way of combining them and allows user-defined effects. One of the challenges this abstraction poses is that of handling several occurrences of the same effect while maintaining its modularity. The solutions vary from the *generative approach* [3, 7], whereby fresh instances (*names*) of an effect are dynamically introduced, *lexically scoped handlers* [6], to introducing new language constructs [5]. Our aim is to build a fully-abstract model for such a language with typed effects as in [2, 8], focusing on the *generative approach*, where instances of an effect are taken as first-class values.

Programming language In the present work, we consider a fine-grained call-by-value programming language with typed algebraic effects and handlers. We stress that effect instances are first-class values that can be exchanged in the same way exceptions or ML-style references, in say OCaml, could be shared. The *generative approach* coupled with the ability to exchange names introduce several challenges to the design of a strong static typing system that rules out unhandled performed effects [7]. We do not address these difficulties in this work.

The operational semantics is standard and is given by an *reduction relation* over pairs $(\mathtt{M};\mathcal{V})$ formed by a term $\mathtt{M}$ and a set $\mathcal{V}$ of all previously generated effect instance names. This component $\mathcal{V}$ is crucial to provide a semantics to the syntactic construction $\mathtt{new\ E}$ representing the generation of a *fresh* instance of an effect of type $\mathtt{E}$.

We consider in this work the standard notion of *contextual equivalence* $\equiv_{ctx}$ for this language. Consider the two following terms, a variation of an example from [5] with dynamic instance generation.

$$\mathtt{M}_1 \triangleq f\ (\lambda x.5) \qquad \mathtt{M}_2 \triangleq \mathsf{let}\, y = \mathtt{new\ E}\, \mathsf{in}\, \mathtt{handle}\ f\ (\lambda x.y\#\mathtt{op}\ ())\ \mathtt{with}\ \{y\#\mathtt{op}\ x\ \kappa \mapsto \kappa\ 5\}$$

These two terms are contextually equivalent because the effect instance bound to y is generated dynamically so that the function f cannot have any prior knowledge of it. And given it has not been disclosed to the environment, it remains private and thus cannot be handled by f.

Consider now a variation of the previous example:

$$\mathtt{N}_1 = \mathsf{let}\, y = \mathtt{new\ E}\, \mathsf{in}\, g\ y; f\ (\lambda x.5) \qquad \mathtt{N}_2 = \mathsf{let}\, y = \mathtt{new\ E}\, \mathsf{in}\, \mathtt{handle}\ g\ y; f\ (\lambda x.y\#\mathtt{op}\ ())\ \mathtt{with}\ \{y\#\mathtt{op}\ x\ \kappa \mapsto \kappa\ 5\}$$

The terms are no longer contextually equivalent because the secrecy of the instance bound to y is not maintained in this example; it is disclosed to g. So a suitably defined context where the fresh instance is stored by g, and reused subsequently by f to handle the effect would be able to distinguish them.

Normal form bisimulations [10] have been designed for a language with algebraic effects and handlers [4], and shown to be fully abstract, that is to coincide with contextual equivalence. However, it does not seem possible to extend normal form bisimulation for a language with generativity and local instances of effects.

Operational Game Semantics Operational game semantics (OGS) is a trace semantics originally introduced in [9], described by a bipartite labelled transition system (LTS) with passive states (environment

configurations) and active states (program configurations). An environment configuration represents the knowledge state at a given stage of the interaction as it contains the functional values and evaluation contexts that a program has disclosed to the environment. They are accessed by the environment respectively through function and continuation names, that are exchanged through actions of the LTS. A program configuration is a term in addition to an environment configuration. While an environment state restricts the set of possible environment moves, a program move is deterministically given by its normal form. Transitions are either questions with a label of the form $\overline{f}(\mathtt{A}, c)$ (requesting the result of f applied to $\mathtt{A}$ as an answer through the continuation name c) or answers with a label of the form $\overline{c}(\mathtt{A})$ (answering with $\mathtt{A}$ through the continuation named c). In OGS, a term is thus interpreted by a set of traces representing all of its possible interactions with any compatible environment. For example, given the open term $f\!:\!\mathtt{Unit} \to \mathtt{Bool} \vdash f(\lambda x.5)$, an interaction with an environment in which f does not call its argument and returns $\mathtt{true}$ would result in a trace such as $t_0 = \overline{f}(g, c)\ c(\mathtt{true})$, while an environment in which f calls its argument twice before returning $\mathtt{true}$ is given by $t_2 = \overline{f}(g, c)\ g((), d)\ \overline{d}(5)\ g((), d')\ \overline{d'}(5)\ c(\mathtt{true})$.

Whereas in previous game semantics works on computational effects, explicit strategies had to be given for each effectful construct in the language [1, 11], in presence of algebraic effect and handlers one has to account for observable effects in a generic fashion. Indeed, in this setting one can distinguish between two kinds of normal forms; *open-stuck* terms (open normal forms) and *control-stuck* terms (due to unhandled effects) [4]. An *open-stuck* term, e.g $f\ (\lambda x.5)$, calls for a question to the environment on the unknown function name f represented by a transition labelled by $\overline{f}(g, c)$. For *control-stuck* terms, we extend the OGS LTS with a transition labelled by $\overline{c}[\iota\#\mathtt{op}\ \mathtt{A}\ r]$. Intuitively such terms $\mathcal{E}[\iota\#\mathtt{op}\,\mathtt{V}]$ call for a request to the environment (on the enclosing continuation name, say c) to handle the performed effect $\iota\#\mathtt{op}\,\mathtt{A}$ (if possible) by giving it access to the delimited continuation $\mathcal{E}$ through the abstract name r.

To compute normal forms of terms found in active states, all the states of the LTS have to carry a component $\mathcal{V}$ of previously generated name instances. To track the instances that have been exchanged by the program and the environment, we embed an extra component $\mathcal{N}$ of *disclosed* instance names. When the effect instance is private, that is it has not been previously disclosed, the environment cannot possibly handle it but only propagate it to the enclosing evaluation context. We thus introduce additional transitions that enforce the propagation and access to the fragment of the evaluation context that belongs to the environment. The transition $\overline{\mathbf{fwd}}(\overline{c}, d, \kappa_c)$ corresponds roughly to the two actions $\overline{c}[\iota\#\mathtt{op}\ \mathtt{A}\ r]\ d[\iota\#\mathtt{op}\ \mathtt{A}\ \kappa_c]$ where the program first performs an effect, then the environment propagates it with $d[\iota\#\mathtt{op}\ \mathtt{A}\ \kappa_c]$ giving the program access to its delimited continuation c through the name κ_c.

Full-abstraction Building a fully-abstract OGS model means that we need a suitable notion of equivalence of traces so that the set of traces generated by two terms are equivalent *iff* they are contextually equivalent.

Let us consider again the terms $\mathtt{M}_1$ and $\mathtt{M}_2$ from the previous example, even though no context would be able to distinguish them, the sets of traces they generate are not identical. The interaction with a context in which f does not call its argument results in identical traces for $\mathtt{M}_1$ and $\mathtt{M}_2$, but it is not the case for a context that does. Indeed, the trace $t_{\mathtt{M}_1} = \overline{f}(g, c)\ g((), d)\ \overline{d}(5)\ c(\mathtt{A})$ belongs to the interpretation of $\mathtt{M}_1$ but not that of $\mathtt{M}_2$. Similarily, $t_{\mathtt{M}_2} = \overline{f}(g, c)\ g((), d)\ \overline{\mathbf{fwd}}(\overline{d}, c, \kappa_d)\ \overline{\kappa_d}(5, c')\ c'(\mathtt{A})$ belongs to the interpretation of $\mathtt{M}_2$ but not that of $\mathtt{M}_1$. Both traces represent interactions of $\mathtt{M}_1$ (resp. $\mathtt{M}_2$) with a same environment (in which f is a function that calls its argument once before returning $\mathtt{A}$) that cannot tell them apart. Trace equality then appears to be too strict and traces need to be quotiented up-to forward moves. We do this by considering a forward-free canonical form of traces in the definition of trace equivalence.

Completeness of game models famously relies on the property of definability; that is for every object from the semantic domain (a trace, here) there exists a term that corresponds to it. When contexts have access to high-order references, the proofs are generally based on constructions that will store in private references all the functions and continuations shared with the environment so that they could be dereferenced and used when needed. Moreover, an integer reference is also used to implement a 'clock', thus tailoring the behavior expected from the trace (strategy) at any given clock tick. In our setting, this is achieved by a suitable encoding of (high-order) references by private store effect instances and a resort to the quotient previously mentioned.

References

[1] Abramsky, S., and McCusker, G. Game semantics. In *Computational Logic: Proceedings of the NATO Advanced Study Institute on Computational Logic, held in Marktoberdorf, Germany, July 29–August 10, 1997* (1999), Springer, pp. 1–55.

[2] Bauer, A., and Pretnar, M. An effect system for algebraic effects and handlers. In *Algebra and Coalgebra in Computer Science: 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings 5* (2013), Springer, pp. 1–16.

[3] Bauer, A., and Pretnar, M. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming 84*, 1 (2015), 108–123. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

[4] Biernacki, D., Lenglet, S., and Polesiuk, P. A complete normal-form bisimilarity for algebraic effects and handlers. In *Formal Structures for Computation and Deduction* (2020).

[5] Biernacki, D., Piróg, M., Polesiuk, P., and Sieczkowski, F. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 1–30.

[6] Biernacki, D., Piróg, M., Polesiuk, P., and Sieczkowski, F. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages 4*, POPL (2019), 1–29.

[7] de Vilhena, P. E., and Pottier, F. A type system for effect handlers and dynamic labels. In *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings* (2023), Springer Nature Switzerland Cham, pp. 225–252.

[8] Hillerström, D., and Lindley, S. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (2016), pp. 15–27.

[9] Laird, J. A fully abstract trace semantics for general references. In *Proceedings of the 34th International Conference on Automata, Languages and Programming* (Berlin, Heidelberg, 2007), ICALP'07, Springer-Verlag, p. 667–679.

[10] Lassen, S. B. Eager normal form bisimulation. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings* (2005), IEEE Computer Society, pp. 345–354.

[11] Murawski, A. S., and Tzevelekos, N. Game semantics for nominal exceptions. In *Foundations of Software Science and Computation Structures: 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 17* (2014), Springer, pp. 164–179.

[12] Plotkin, G., and Power, J. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science 45* (2001), 332–345.

[13] Plotkin, G., and Power, J. Computational effects and operations: An overview.

[14] Plotkin, G. D., and Pretnar, M. Handling algebraic effects. *Logical methods in computer science 9* (2013).