A Contextual Equivalence Checker for IMJ*

Andrzej S. Murawski[1], Steven J. Ramsay[1], and Nikos Tzevelekos[2]

[1] University of Warwick, Coventry, UK
[2] Queen Mary University of London, London, UK

Abstract. We present CONEQCT: a contextual equivalence checking tool for terms of IMJ*, a fragment of Interface Middleweight Java for which the problem is decidable. Given two, possibly open (containing free identifiers), terms of the language, the contextual equivalence problem asks if the terms can be distinguished by any possible IMJ context. Although there has been a lot of prior work describing methods for constructing proofs of equivalence by hand, ours is the first tool to decide equivalences for a non-trivial, object-oriented language, completely automatically. This is achieved by reducing the equivalence problem to the emptiness problem for fresh-register pushdown automata. An evaluation demonstrates that our tool works well on examples taken from the literature.

A dedicated webpage for the tool is: http://bitbucket.org/sjr/coneqct.

1 Introduction

Two phrases of a programming language are *contextually equivalent* if occurrences of the first phrase in any program can be replaced by the second phrase without affecting the result of the program. The notion plays a fundamental role in a variety of verification tasks: it can be used to support proofs of correctness for program transformations, code optimisations, refactoring and updates. However, due to quantification over all possible contexts in which a phrase can be inserted ("in any program"), contextual equivalences are notoriously difficult to establish directly. Over the years, both semantic (e.g. domain theory) and operational (e.g. traces, bisimulations, logical relations) methods have been used to provide techniques to overcome the problem, yet decidability results have been scarce. Our tool, which targets a fragment of Middleweight Java augmented with interfaces, capitalises on recent progress in game semantics as well as automata theory over infinite alphabets.

In recent years game semantics has led to the construction of fully abstract models for a whole range of programming languages. Although originating from the tradition of denotational semantics, they have a concrete flavour, which makes them suitable for representations as formal languages. The particular model for Java [10], that we take advantage of in this paper, is built over an infinite set of names (used to model object references), suggesting the use of automata over infinite alphabets as a framework for representing the denotations. Indeed, in the companion research paper [8], we have identified a fragment of Java, called IMJ*, whose game semantics can be captured using (visibly) pushdown register automata over infinite alphabets with fresh-symbol generation [9]. The automata are equipped with a *finite* set of registers for storing elements of

the *infinite* alphabet as well as a pushdown store. In the present tool paper we present the implementation of the associated decision procedure.

Our input language IMJ* is a fragment of Interface Middleweight Java (IMJ) [10]. It is a stripped down version of Middleweight Java (MJ) [4], designed to expose the interactions of MJ-style objects with the environment through the addition of interfaces.

Example 1 ([12]). Consider the interfaces $\mathsf{IntRef} = \{\mathsf{val} : \mathsf{int}\}$, $I = \{m : \mathsf{IntRef} \to \mathsf{IntRef}\}$ and terms $\{\mathsf{IntRef}, I\} \mid \emptyset \vdash M_1, M_2 : I$ given below. Note that a and b are declared locally, i.e. they play the role of private fields. As both are unknown to the environment, the first call to m in any computational scenario will make the if-condition fail and a will be exposed to the environment as a result. Consequently, it can be recorded by the context and used in subsequent interactions with the terms. For M_1, this makes it possible to satisfy the branching condition, which will reveal the second private name b. In contrast, M_2 will never reveal b, however many times the object is used. Hence, the terms are not contextually equivalent. This example appears as inp11.imj in Table 1.

$M_1 \equiv$
```
let a = new {_:IntRef;} in
let b = new {_:IntRef;} in
new {_:I; m: λc. if c = a then b else a}
```

$M_2 \equiv$
```
let a = new {_: IntRef;} in
let b = new {_:IntRef;} in
new {_:I; m: λc. if c = b then b else a}
```

2 Tool Architecture

The tool decides contextual equivalence by compiling the pair of input terms to their game semantics and checking that they are assigned the same meaning. This is a complete method for deciding equivalence due to the full abstraction result in [10], which states that two terms are observationally equivalent iff they are assigned the same sets of (complete) plays by the game model.

The plays are sequences of moves that trace out the possible interactions of a term with its environment. For example, the program may play a move call $a.m(v)^{\Sigma}$, which represents a call with argument v to method m of an object named a in the environment, at a point in the execution where the externally observable part of the heap is described by Σ. A play can be viewed as a word over an alphabet, but the alphabet is infinite since there is no bound on the number of objects that can be created (and hence the set of objects names a). In IMJ, objects names can only be tested for equality, and in [8], it is shown that a special kind of visibly pushdown register automaton, called an IMJ Automaton (IMJA) suffices to exactly characterise the set of plays assigned to an IMJ* term by its game semantics. In this representation, the move above will have shape call $r.m(w)^S$, where r is the index of a register of the automaton which contains the name of the object a, w is a symbolic representation of v and S of Σ. Even though Σ may be unbounded, IMJ* terms modify only bounded fragments of the heap, represented by S. Checking that two sets of plays are equal is then reduced to checking IMJA language equivalence (strictly speaking, up to saturation of stores S to their full size corresponding to Σ).

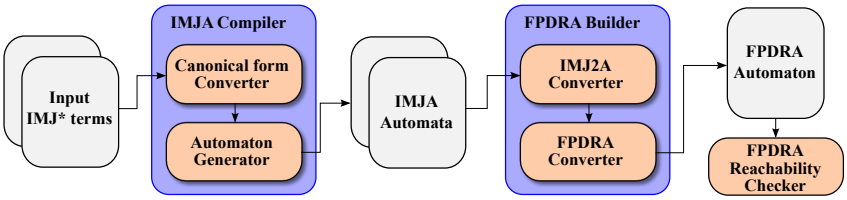

Fig. 1. Overview of tool architecture.

Checking language equivalence of IMJA proceeds through a series of intermediate constructions, ultimately concluding with a fresh-register pushdown automaton (FPDRA). Due to the properties of the translation, the two IMJA are language equivalent iff the FPDRA is (language) empty. An overview is shown in Figure 1. The tool reports the main characteristics of each of the intermediate constructions (e.g. number of states, number of registers) and the time taken to construct them.

From IMJ* *Terms to IMJA.* The first transformation is from the pair input terms to a pair of IMJA. This translation is extensively documented in [8] and our implementation is faithful to that description, so we shall not discuss it further here. As soon as the IMJA are produced, we remove states that are not graph-reachable from the initial state or backwards-graph-unreachable from an accepting state (by graph-reachable we mean reachable in the finite transition graph of the IMJA).

From IMJA to IMJ2A. The strategy for checking language equivalence of IMJA is to construct a kind of symmetric difference automaton, which accepts exactly those words that are in one of the two IMJA but not in the other, which is called an IMJ2 Automaton (IMJ2A) in [8]. This is possible, because IMJA operate over a visibly pushdown alphabet [2] and, hence, their stacks can be synchronised. Overall, the translation in *ibid.* ensures that the pair of IMJA represent the same interactions (plays) iff their IMJ2A translate is empty.

From IMJ2A to FPDRA. IMJ2A are defined in terms of the underlying transitions of the two constituent IMJA. Because they refer to two sets of registers, emptiness checking is not straightforward: in order for the automata to synchronise, matching names have to be used as labels. The following is an example of an IMJA2 transition:

$$(q_1, q_2) \xrightarrow{\mathsf{call}\ r_1.m()^{S_1}, \mathsf{call}\ r_2.m()^{S_2}} (q_1', q_2')$$

This transition describes how if the two underlying IMJA have reached states q_1 and q_2 respectively and, moreover, the same object name is contained in register r_1 of the first IMJA and register r_2 of the second IMJA and there is a correspondence between the names contained in the registers of the two IMJA which makes S_1 and S_2 correspond, then they will both consume this call move and step into states q_1' and q_2' respectively.

The tool compiles away such special transitions by tracking register correspondences. These are pairs of maps which describe how registers from the two constituent

IMJA are represented by registers from the FPDRA under construction. This is achieved by a least fixed point computation: the two initial states are extended with the identity correspondence (representing the fact that initially both IMJA have the same register assignment) and transitions are simulated to obtain a register correspondence at the end of the transition given one at the start. A transition such as the one above can then be compiled into a set of simpler transitions that do not have a semantics that is specialized to representing plays of IMJ^* terms. For example, if a correspondence has r_1 from the first IMJA and r_2 from the second being represented by the same register of the new FPDRA and, lifted to stores, makes S_1 and S_2 correspond, then the above transition degenerates to $(q_1, \sigma, q_2) \xrightarrow{\epsilon} (q_1', \sigma, q_2')$. Since we are only interested in emptiness of the IMJ2A, i.e. the impossibility of reaching a final state, the particular letter that is read is irrelevant, which explains why the degenerate transition is an epsilon transition. The result of this fixed point computation is a fresh pushdown-register automaton (FPDRA) [9], that is, a register automaton [11] (RA) with global-fresh transitions and a pushdown stack.

FPDRA Reachability Check via Saturation. Finally we decide the emptiness of the FPDRA by using an extension (to handle global freshness and empty registers) of the saturation algorithm described in [7]. This procedure constructs an RA that represents all the possible configurations of the FPRDA that can reach accepting states. In relation to [7], the main addition is the use of a tagging technique for specifying elements of the registers and the stack that are required to be globally fresh, which allows us to simulate global freshness via local freshness for reachability purposes (cf. [9]). Consequently, equivalence of the two terms is reduced to checking whether an initial configuration of the FPDRA is accepted by the RA, which is solved by graph reachability.

3 Evaluation

We evaluated CONEQCT on examples drawn from the literature around contextual equivalence for high-level languages [1,3,5,6,8,12,13], adapted to IMJ^* syntax. (The website of the tool contains a more detailed listing.) For equivalences, the papers contain manually constructed proofs based on logical relations or environmental bisimulations. Their full automation would be challenging, because they require witness relations that have to be guessed. In contrast, CONEQCT is automated and also detects inequivalences.

The tool is written in F# and we evaluated it on Microsoft's .NET Framework 4.5.2, running on a machine with an Intel Core i7 1.8GHz processor and 8GB of RAM. The results of the evaluation are shown in Table 1. The first column gives the file name of the inputand the second states whether the input is an equivalence (Y) or an inequivalence (N). The next four columns give the number of states in the intermediate constructions. The number of states in the IMJ2A is omitted since it is always a simple function of the number of states in the two constituent IMJA automata. Column seven gives the number of registers in the FPDRA and hence RA, and column eight gives the total time for processing the input in milliseconds (mean $\pm$ s.e.; $n = 10$).

A couple of interesting observations can be made regarding the results. First, in many instances of equivalence, the number of states in the RA is 0. This indicates that

Table 1. Results of the evaluation

Input	Eq?	IMJA 1	IMJA 2	FPDRS	RA	Regs	Time (ms)
inp1.imj	Y	26	82	1688	0	7	1833 ± 11
inp2.imj	Y	1	1	0	0	0	253 ± 10
inp3.imj	Y	32	32	428	0	6	355 ± 11
inp4.imj	Y	21	22	61	0	2	292 ± 10
inp4b.imj	N	21	12	76	41	2	301 ± 10
inp5.imj	Y	18	3	5	0	4	293 ± 9
inp6.imj	Y	26	26	292	0	4	322 ± 12
inp7.imj	Y	76	16	1078	972	4	445 ± 13
inp7b.imj	N	76	11	418	404	4	373 ± 11
inp8.imj	Y	17	11	124	0	4	289 ± 1
inp9.imj	Y	33	11	287	141	4	326 ± 1
inp9b.imj	N	17	11	138	134	4	320 ± 2
inp10.imj	Y	96	96	1528	0	4	1344 ± 9
inp10b.imj	N	42	96	2476	2468	4	1256 ± 30
inp11.imj	N	92	32	796	796	7	584 ± 4
inp12.imj	Y	11	9	26	0	2	266 ± 1
inp13.imj	Y	13	13	111	0	4	299 ± 1
inp14.imj	Y	13	13	37	0	2	21908 ± 291
inp15.imj	Y	34	242	8714	0	7	3647 ± 49
inp15b.imj	N	34	242	10725	10725	7	4095 ± 44
inp16.imj	Y	272	137	17888	0	8	16499 ± 155
inp16b.imj	N	56	137	11833	11833	8	4731 ± 123

the corresponding FPDRA has no accepting states. This happens if, as a result of the fixed point computation, the exploration of register correspondences reveals that it is not possible for the IMJ2A to accept a word. For example, this will be the case when the compilation of the two terms happens to yield identical IMJA.

A second observation regards the time taken. One of the most time expensive examples is input 14, yet the various intermediate constructions are all relatively small. Further investigation reveals that the vast majority of the time is spent constructing the two IMJA. Although the two IMJA that are constructed are ultimately small, as part of their construction far larger automata are built but then later restricted and have many unreachable states pruned away. It will be interesting future work to understand how to ensure that time is not wasted producing such expensive yet transient constructions.

Further Directions. Whilst we have shown that the tool performs well on the kinds of examples seen in the literature, we see further directions which would make this line of work more generally applicable. We aim to optimise the handling of time-intensive automata constructions, and to empower CONEQCT with predicate abstraction, allowing it to reason symbolically not only about object names but also basic data values.

Acknowledgements. Research supported by the Engineering and Physical Sciences Research Council (EP/J019577/1) and the Royal Academy of Engineering (RF: Tzevelekos).

References

1. A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proceedings of POPL*, pages 340–353. ACM, 2009.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of STOC'04*, pages 202–211, 2004.
3. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proceedings of TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2005.
4. G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Computer Laboratory, University of Cambridge, 2002.
5. D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proceedings of ICFP*, pages 143–156. ACM, 2010.
6. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of POPL*, pages 141–152. ACM, 2006.
7. A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. Reachability in pushdown register automata. In *Proceedings of MFCS*, LNCS, pages 464–473. Springer, 2014.
8. A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. Game semantic analysis of equivalence in IMJ. To appear at ATVA'15, available at http://bitbucket.org/sjr/coneqct/src, 2015.
9. A. S. Murawski and N. Tzevelekos. Algorithmic games for full ground references. In *ICALP*, volume 7392 of *LNCS*, pages 312–324. Springer, 2012.
10. A. S. Murawski and N. Tzevelekos. Game semantics for Interface Middleweight Java. In *POPL*, pages 517–528, 2014.
11. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.
12. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher-Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.
13. Y. Welsch and A. Poetzsch-Heffter. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Science of Computer Programming*, 92, Part B(0):129 – 161, 2014.