Slicing from Formal Semantics: Chisel

Adrián Riesco[1(✉)], Irina Măriuca Asăvoae[2], and Mihail Asăvoae[2]

[1] Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es
[2] Inria Paris, Paris, France
{irina-mariuca.asavoae,mihail.asavoae}@inria.fr

Abstract. We describe Chisel—a tool that synthesizes a program slicer directly from a given algebraic specification of a programming language operational semantics. This semantics is assumed to be a rewriting logic specification, given in Maude, while the program is a ground term of this specification. We present the tool on two types of language paradigms: high-level, imperative and low-level assembly languages. We conduct experiments with standard benchmarking used in avionics.

1 Introduction

Lately we observe an increased interest in defining programming languages semantics as rewriting systems. This desideratum is stated in *the rewriting logic semantics project* [5], where the languages semantics are defined as rewriting systems using Maude [3], and it is followed by the $\mathbb{K}$ framework [8]. Our work complements the rewriting logic semantics project by developing static analysis methods, e.g., slicing, for programs written in languages with semantics already defined in Maude. Here we present Chisel—a Maude tool for *generic* slicing.

Slicing is an analysis method employed for program debugging, testing, code parallelization, compiler tuning, etc. In essence, a slicing method evaluates data flow equations over the control flow graph of the program. Tip gives in [9] a comprehensive survey on the standard program slicing techniques applied over different programming language concepts. All these techniques are built using different models that represent augmentations of the control flow graph. Hence, the translation of the programs into these models has to be automatized and this has to be produced at the level of the programming language under consideration.

Chisel aims to advance the generic synthesis of program models from any programming language, provided the algebraic semantics of the language is given as a rewriting system. Namely, from a programming language semantics, given as a Maude specification, Chisel extracts pieces of interest for slicing, and uses these pieces to augment the program term and to produce the model, which is then sliced. We use for experiments two semantics: one for an imperative

Research partially supported by MINECO Spanish projects *StrongSoft* (TIN2012-39391-C04-04) and *TRACES* (TIN2015-67522-C3-3-R), and Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

M. Huisman and J. Rubin (Eds.): FASE 2017, LNCS 10202, pp. 374–378, 2017.
DOI: 10.1007/978-3-662-54494-5_21

programming language with functions, WhileFun, and another for the MIPS assembly language. Chisel analyzes these semantics and extracts key information for slicing (e.g. side-effect constructs and their data flow direction) that are used when traversing the program term in order to obtain the program slice.

With Chisel we target sequential imperative code generated from synchronous designs—a class of applications used in real-time systems, e.g., avionics. Note that Chisel is not yet able to handle pointers, but the synchronous programs do not contain pointers either. For the evaluation of Chisel on industrial benchmarks, the pointers are transformed into function calls.

Related Work: An early work on generic slicing is presented in [4] where the tool compiles a program into a self slicer. Generic slicing is also the focus of the ORBS [2] tool which proposes observation-based slicing, a statement deletion technique for dynamic slicing. In rewriting logic [1] implements dynamic slicing for execution traces of the Maude model checker. In comparison with these tools, Chisel proposes a static approach to generating slicers for programming languages starting from their formal semantics. Given its static base, Chisel computes slices for programs and not for (model checker) runs. In [7] we refer to more technical details of the approach. Nevertheless, the developing platform of Chisel—Maude allows us to approach other types of slicing, e.g., dynamic or amorphous. Note that while dynamic slicing methods (either the classic or deletion-based) let us preserve genericity, determining generic statements equivalence for amorphous slicing might prove difficult. For the later case, we envision heuristics that start with our data dependence inference and compute some form of transitive closure.

The rest of the paper is organized as follows: Sect. 2 gives an overview of the tool, Sect. 3 describes the experimental results obtained, while Sect. 4 concludes and outlines some lines of future work. The complete code of the tool and examples are available at https://github.com/ariesco/chisel.

2 Chisel Design

In this section we describe the ideas underlying the tool. The main observation we use for Chisel is the fact that side-effects induce an update in the memory afferent to the program. Hence, Chisel first detects the operators used by the semantics to produce memory updates. Then, the usage of the memory update operators is traced through semantics up to the language constructs. Any language construct that may produce a memory update is classified as producing side-effects. Moreover, following the direction of the memory updates, we infer also the data flow details for each side-effect language construct. Finally, the information gathered by Chisel about language constructs is used to traverse the term representing the program and to extract the subterms representing the slice.

In Fig. 1 we depict the structure of Chisel by components and their input-output relation. We present next details about each of Chisel's components that work with $\mathcal{S}$—the Maude specification of the programming language semantics.

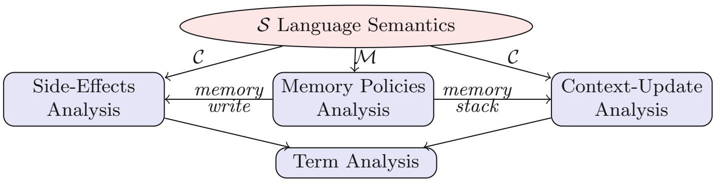

Fig. 1. Chisel components: the formal language semantics and the analyses.

Memory Policies Semantics Analysis: Let us define $\mathcal{M}$ as the part of $\mathcal{S}$ that defines some (abstract) form of the memory used during program execution. Our assumption about the structure of the memory is that it connects the variables in the program with their values possibly via a chain of intermediate addresses. We define a *memory policy* as a particular type of operators specified using $\mathcal{M}$. For example, a *memory-read* is the set of operators in $\mathcal{M}$ that contain in their arity the sort for variables and for memory, and in their co-arity the sort for values. A *memory-write* operator contains in the arity the memory, the variable, and a value, in the co-arity the memory, and the rules defining this operator change the memory variable by updating the value.

Side-Effect Semantics Analysis: Let us denote as $\mathcal{C}$ the part of $\mathcal{S}$ that defines the operators representing the programming language constructs, i.e., language instructions. We name *side-effect constructs* those operators in $\mathcal{C}$ that may produce a memory-write over some of its variable component. The side-effect analysis starts with the rules with $\mathcal{C}$ operators in the left-hand side and constructs a *hyper-tree* $\mathcal{T}$ whose nodes are sets of rewrite rules and edges are unification-based dependencies between these rules. The paths $\mathcal{P}$ in $\mathcal{T}$ with leaves that contain rules classified by the memory policy phase as memory-writes are signalling the side-effect constructs. Next, by trickling-up the paths in $\mathcal{P}$, Chisel determines the data flow (source-destination) produced by the side-effect constructs.

Context-Update Semantics Analysis: We see the program p as a term $t \in \mathcal{S}$ that can be flattened into a list $\mathcal{L}$ of elements from $\mathcal{C}$ by a preorder traversal of the tree associated to t. We define as *context-update constructs* those operators in $\mathcal{C}$ that, during p execution using $\mathcal{S}$, produce changes to $\mathcal{L}$. For example, function calls and gotos are context-updates. The inference of a set of constructs that may produce context-updates filters the paths in $\mathcal{T}$ by using a *stack memory policy* at the leaves level. This analysis is work in progress, which we only brief here. Currently, we provide these constructs for each $\mathcal{S}$.

Term Augmentation and Traversal: The algorithm for slicing a program takes as input a *slicing criterion* S consisting in a set of program variables. In this step, Chisel takes the list $\mathcal{L}$ of $\mathcal{C}$ subterms obtained from the program term and traverses it repeatedly until the set S stabilizes. While traversing the list $\mathcal{L}$, whenever a side-effect construct is encountered, if the destination of this construct is from S then all the source variables are added to S. Moreover, whenever a context-update construct is encountered, the traversal of $\mathcal{L}$ is redirected towards the element of $\mathcal{L}$ matching a particular subterm in the context-update construct.

3 Chisel Experiments

We run Chisel on a standard benchmark for real-time systems called PapaBench [6], a code snapshot extracted from an actual real-time system designed for Unmanned Aerial Vehicle. We report the results of Chisel for the core functionalities (rows 1, 2), and the complete PapaBench benchmark (rows 3, 4). For both WhileFun and MIPS variants of the benchmarks, we quantify the number of functions and function calls (columns `#Funs` and respectively `#Calls`), the code size (`LOC`), and the slicing reduction factor, `red(%)`. The reduction factor captures the slicing performance w.r.t. the original code on both WhileFun and MIPS variants (Fig. 2).

No	Name	# Funs	# Calls	LOC (WhileFun)	red (%) (WhileFun)	LOC (MIPS)	red (%) (MIPS)
1	`scheduler_fbw`	14	18	103	72.8 %	396	44.4 %
2	`periodic_auto`	21	80	225	73.3 %	779	36.3 %
3	`fly_by_wire`	41	110	638	91.1 %	1913	41 %
4	`autopilot`	95	214	1384	92 %	5639	41.5 %

Fig. 2. Chisel performance on PapaBench benchmark

The lower percentages obtained for the MIPS code appear because of the current limitation of Chisel in handling memory addresses. Moreover, any function call in a small sized function involves setting the function stack with registers global and stack pointer, which dominate the code size yielding longer slices.

4 Conclusions

In this paper we have presented Chisel, a Maude tool that, given the semantics of a programming language written as a rewriting specification in Maude, can (both intra- and interprocedural) slice programs written in that language. We tested Chisel with different semantics: WhileFun (imperative) and MIPS (assembly), both with different variations (e.g. different memory models and data flow styles). In future work, we plan to extend the language with pointers, hence supporting more complex memory policies, based on more refined memory models.

References

1. Alpuente, M., Ballis, D., Frechina, F., Sapiña, J.: Combining runtime checking and slicing to improve Maude error diagnosis. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C. (eds.) Logic, Rewriting, and Concurrency. LNCS, vol. 9200, pp. 72–96. Springer, Cham (2015). doi:10.1007/978-3-319-23165-5_3

2. Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., Yoo, S.: ORBS: language-independent program slicing. In: SIGSOFT FSE 2014, pp. 109–120. ACM (2014)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71999-1
4. Danicic, S., Harman, M.: Espresso: a slicer generator. In: SAC 2000, pp. 831–839. ACM (2000)
5. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoret. Comput. Sci. **373**(3), 213–237 (2007)
6. Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.P., Michiel, M.D.: PapaBench: a free real-time benchmark. In: WCET 2006. IBFI, Schloss Dagstuhl (2006)
7. Riesco, A., Asavoae, I.M., Asavoae, M.: Memory policy analysis for semantics specifications in Maude. In: Falaschi, M. (ed.) LOPSTR 2015. LNCS, vol. 9527, pp. 293–310. Springer, Cham (2015). doi:10.1007/978-3-319-27436-2_18
8. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010)
9. Tip, F.: A survey of program slicing techniques. JPL **3**(3), 121–189 (1995)