Memory Policy Analysis for Semantics Specifications in Maude

Adrián Riesco[1], Irina Măriuca Asăvoae[2], and Mihail Asăvoae[3](✉)

[1] Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es
[2] Swansea University, Swansea, UK
I.M.Asavoae@swansea.ac.uk
[3] Inria Paris-Rocquencourt, Paris, France
mihail.asavoae@inria.fr

Abstract. In this paper we propose an approach to the analysis of formal language semantics. In our analysis we target memory policies, namely, whether the formal specification under consideration follows a particular standard when defining how the language constructs work with the memory. More specifically, we consider Maude specifications of formal programming language semantics and we investigate these specifications at the meta-level in order to identify the memory elements (e.g., variables and values) and how the language syntactic constructs employ the memory and its elements. The current work is motivated by previous work on generic slicing in Maude, in the pursuit of making our generic slicing as general as possible. In this way, we integrate the current technique into an existing implementation of a generic semantics-based program slicer.

Keywords: Formal semantics · Maude · Slicing · Analysis · Memory policies

1 Introduction

Static program analysis provides functional and non-functional guarantees with respect to the program behavior. These guarantees, e.g., invariants, are automatically computed from predefined approximations of the concrete program executions. Examples on standard invariants include pointer behavior in sequential code, data races in concurrent code, or bounds of execution time/memory usage.

Rewriting logic provides support to define formal and executable language semantics. A key aspect in a language definition is the memory model—the set of all semantic entities that are required to describe the storage component of a program execution. Let us consider how memory is organized for two

This research has been partially supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04) and by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

M. Falaschi (Ed.): LOPSTR 2015, LNCS 9527, pp. 293–310, 2015.
DOI: 10.1007/978-3-319-27436-2_18

languages defined in rewriting logic—an imperative language with functions and input/output support and an assembly language generated from it. For example, the formal definition of the imperative language would require a global memory for (global) program variables, local environment (for locals), call stack for functions, and input/output buffers. A program execution is a sequence of rewrite steps that access one or more of these storages. In a similar fashion, the formal definition of an assembly language relies on a main memory represented as an array of memory cells, each cell stores one value, and a set of (general purpose or specialized) registers for everything else.

Our goal is to design generic program analysis tools based on a meta-level analysis of the programming language semantic definition. This would allow a certain degree of parameterization of the program analysis such that changes in the formal language semantics should not result in the need of adapting the corresponding analyzer, since the analyzer automatically incorporates the modifications. This approach builds on the formal executable language semantics given as a rewriting logic theory [7,19] and on the program to be analyzed. The generic design for program analysis tools based on language semantics comes in two steps. The first step is a meta-analysis of the formal language semantics. The second step is a data dependency analysis of the program. The meta-level analysis is a fixpoint computation of the set of basic language constructs of interest, e.g., side-effect constructs, which is then used to extract safe program slices based on a required criterion. This methodology is instantiated in [3,26] on the classical WHILE language augmented with a side-effect assignment and read/write statements and, respectively, on the WhileF language—an extension of WHILE to allow functions and scope declaration for variables.

An example program in WhileF is in Fig. 1 (left). Note however that both intra- and inter-procedural program slicing methods are based on a less generic assumption: the general memory update operation—the assignment statements—has a fixed destination: its left-hand side. This is not necessarily generic as, for example, the family of the assembly languages uses explicit memory operations (load/store) and arithmetic/logic operations (which update registers), with flexible destination placement in the language syntax. For example:

- in MIPS assembly language, in Fig. 1 (middle), the load instruction `lw` has a direction right (source) to left (destination), while the store instruction `sw` has a reverse direction. Moreover, `mult` multiplies the values in the two registers and writes the result in a special multiplication register.
- in `x86` assembly language generated by `gas` (Fig. 1, right top), which is the GNU assembler and the default back-end of the standard `gcc` compiler, an instruction like `movl 16(%esp), %eax` copies into the register `%eax` the value found at the address referred by the register `%esp` shifted left by `16`, as in Fig. 1 (right top). The update is from left (source) to right (destination).
- in `x86` assembly language, in Fig. 1 (right bottom), an instruction like `mov eax, DWORD PTR [esp+28]` copies into the register `eax` a word-length from the address found in the register `esp` shifted to the right with the offset 28. The update is from right (source) to left (destination).

```
read i ; read j ;
s := 0 ; p := 1 ;
while not (i == 0) do
    write (i - j) ;
    s := s + i ;
    p := p * i ;
    read i ;
```

```
   ...
lw $2, 24($fp)
lw $3, 16($fp)
addu $2, $2, $3
sw $2, 24($fp)
lw $2, 28($fp)
lw $3, 16($fp)
mult $2, $3
mflo $2
sw $2, 28($fp)
   ...
```

```
    ...
movl 16(%esp), %eax
addl %eax, 24(%esp)
movl 28(%esp), %eax
imull 16(%esp), %eax
    ...
----------------------------
    ...
mov eax, DWORD PTR [esp+16]
add DWORD PTR [esp+24], eax
mov eax, DWORD PTR [esp+28]
imul eax, DWORD PTR [esp+16]
    ...
```

Fig. 1. WhileF program (left) with snapshots of assembly code MIPS IV (middle), x86 - AT&T (right - top) and x86 - Intel (right - bottom)

The direction of the memory update is an example of what we call a memory policy, meaning the way the language constructs make use of the semantic entities that define the memory model in the formal semantics. Moreover, when we infer the direction of the memory update operation we actually address (in a uniform way) a wide range of low-level languages.

In this paper we propose a refinement of a previously introduced technique in [3,26], where we described a generic intra- and inter-procedural slicing method, respectively. In [26] we focused on inferring the language constructs that produce side-effects from the semantics specification, i.e., language constructs inducing memory updates. In the current work, we infer memory policies, i.e., formal semantics properties about how the language constructs use the memory model defined by the semantics. We particularise the memory policy to *detecting the direction of the data flow in the memory updates*. Namely, given a side-effect construct c in the considered language, we infer which are the sources and which is the destination of the data flow detected in c. For example, in an assignment `x := y + z` our memory policy detects that `y` and `z` are the sources while `x` is the destination. For inferring this memory policy the meta-analysis tracks down how each element in the construct c is used at the memory level (either read or write) and then we trickle up this information back in the components of c.

Paper Outline. This paper is organized as follows: Sect. 2 covers related work; Sect. 3 introduces rewriting logic and Maude as well as our view on memory polices from the rewriting-logic perspective; Sect. 4 details the algorithm of inferring memory policies; Sect. 5 describes the prototype tool. We conclude in Sect. 6.

2 Related Work

Our goal is to design and implement generic formal semantics-based tools for program analysis in a rewriting logic environment, with focus on memory models. Hence, we relate our approach to static program analysis and rewriting logic.

Static program analysis is a compile-time process for automatically extracting run-time semantic information (i.e., invariants) from programs. Abstract interpretation [8], which systematically derives sound approximations of the concrete semantics, and type systems [22], which define correct programs with respect to typing information, are two of the most used techniques for program analysis.

Program analysis based on abstract interpretation uses abstract domains and abstract semantics. The latter is an abstract re-implementation of (some of) the language operations as well as an abstract memory. From the point of view of the abstract representation of the program memory, the abstract semantics can capture a wide range of properties: functional properties, e.g., pointer and alias analyses [15,23], data race detection [12] on shared memory programs, stack safety [24], automated checks for coding standards [29], or non-functional properties, e.g., computation of safe upper bounds for heap size [1] and stack size [4]. In comparison with these approaches, we propose to infer, via meta-analysis of the formal language semantics, certain information (which we call policies) about the abstract memory system.

Having a formal executable semantics with precise memory models allows verification of both sequential and concurrent code. For example, the encoding of the x86 assembly language semantics in HOL proof assistant [28] allows reasoning about memory consistency in threaded applications while the encoding of the memory model of C language in Coq [18] is suitable for pointer arithmetic reasoning. In general, theorem proving either interactive or automated provides the necessary infrastructure to allow meta-level reasoning for programming language semantics, in a similar fashion with our proposal. These approaches are complemented by the rewriting logic semantics project [21], which focuses on how to define formal semantics of programming languages in rewriting logic and how to construct program analysis tools directly over these semantics. The memory component of a language definition in rewriting logic and its applicability in program analysis is presented in [11,14]. The memory model of [14] is exemplified on a simple imperative language with functions. Also, they define pluggable program analyses by reusing parts of the concrete language semantics. For example, the rewriting logic specification of the Java Memory Model [11] is used for model checking Java programs. Our approach accommodates the concept of pluggable program analysis via meta-level manipulation of the program semantics, as given for program slicing in [3].

The term-slicing aspect of our proposed program slicing technique is rooted into the notions of descendant/ancestor and origin tracking [5,16,17]. Origin tracking, introduced in [17] is a refinement of the descendant/ancestor relationship as it follows the symbols of an expression to their causes in an earlier expression in a rewrite sequence. The origin tracking in first-order term rewriting systems [16] is intrinsic to slicing due to its strategy of reasoning on every reduction from a term to its normal form. The term-slicing uses an extended concept of origin tracking, w.r.t. the aforementioned approaches, because it tracks changes in conditional rewriting rules, as defined in Maude, with a particular emphasis on how to slice through rule conditions. In fact, the proposed notion of

term-slicing determines variable dependencies in rules and equations in Maude specifications where the variables are subterms of a certain sort.

In rewriting logic there are several approaches for analysis tools, not necessarily for programs. For example, debugging [2], testing [25], and slicing [2,3,13,26]. The program slicing technique in [13] executes the term representation of a program with the formal semantics and extracts dynamic slices. In comparison, our approach does not execute the formal semantics; the term slicing is based on a meta-level analysis of the semantics. In terms of genericity [13], requires translation steps from a given language semantics into an intermediate language (which is the base for program slicing), whereas our approach works directly on the semantics, as it is defined. The slicing technique in [2] works on generic Maude execution traces. In comparison, we propose a static approach built around a formal semantics and with an emphasis on computing slices for programs and not for execution traces. The work in [25] presents an approach to generate test cases similar to the one presented here in the sense that both use the semantics of programming languages formally specified to extract specific information. However, in [25] the narrowing technique is used on the semantic rules to instantiate the state of the variables in the given program. Matching logic [27] is a program verification technique based on executing a program with a rewriting-based formal semantics, by proving the necessary program invariants. In comparison, our approach is complementary to matching logic as it attempts to compute invariants from the semantics and afterwards, to apply them in program reasoning (e.g., program slicing). Moreover, our approach uses the meta-level capabilities of rewriting logic, which to the best of our knowledge are not available in the matching logic framework.

The technique in the current paper follows our previous work on language-independent program slicing in rewriting logic environment [3]. Actually, the implementation of the current work improves the genericity aspect of the slicing tool developed in [3], since we infer policies about memory updates applied to imperative and assembly languages. The program slicing over the formal semantics S of the language L follows the same two steps as in [3]: (1) an initial meta-analysis of S followed by (2) a program analysis conducted over the programs in L using term slicing.

3 Preliminaries

We present in this section the basic ideas about Maude and memory policies.

3.1 Memory Policies

A formal language semantics consists of the set of all semantic entities that are required to fully specify all possible behaviors of any correct program, i.e., with respect to the semantics definition. Part of the language semantic entities describe the memory system. Examples of such semantic entities are heaps, stacks (e.g., call stack, loop stack), environments, register file, etc. Then,

the language constructs interact, directly or indirectly, with the memory system. Our aim is to infer information about this interaction in an automated way.

We achieve our declared goal of designing generic program analysis tools by employing a meta-level analysis of the formal language semantics. Such a meta-analysis extracts semantics level properties, e.g., the sets of language constructs that may induce side-effects or may result into context-updates. From a memory system point of view, these properties are inferred from the semantics specification by following how the language constructs operate on the memory system. We call this kind of properties *memory policies*. For example, in the case of an imperative language semantics as WhileF, i.e., with functions and input-output capabilities, one memory policy could be named as "direction property". This would involve inferring that in the assignment statements the right-hand side is the source and the left-hand side is the destination.

A more formal view on inferring memory policies would require reasoning at the level of sorting relationships of the semantic entities present in the language semantics specification (starting with a given set of memory-related sorts). For assembly languages in Fig. 1, MIPS considers left to right direction for store and right to left for load instructions while the two x86 styles use the same style for both direction, although it is from left to right for one architecture and from right to left in the other. Consequently, if we are to extend our tool for dealing with a larger class of programming languages, we need to incorporate this particular memory policy inference, which automatically deduces from the semantics specification, for the side-effect constructs, what is the direction of the data flow in each such construct. Note that this direction is crucial for the accuracy of the slicing result, as we need to incorporate in the slicing set only the changing points of certain variables, i.e., where those variables are destination.

3.2 Semantics in Maude

Maude modules are executable rewriting logic specifications. Rewriting logic [20] is a logic of change very suitable for the specification of concurrent systems and it is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [6], which, in addition to equations, allows one to state membership axioms characterizing the elements of a sort. Rewriting logic extends *MEL* by adding rewrite rules.

Maude functional modules [7, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (introduced by the keyword `cmb` and `ceq`, respectively). Maude system modules [7, Chap. 6], introduced with syntax `mod`

`... endm`, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

Maude has been widely used for specifying the semantics of several languages, such as Java [10] or C [9]. The key idea for specifying semantics is, first, to define the signature by means of declaring sorts and their respective constructors (operators). We illustrate this methodology by presenting a simple assembly language that we will use throughout the rest of the paper. This language uses registers to keep intermediate values and defines standard functions, such as addition and subtraction, over them, and also has a memory where values are stored for later sessions. The specification of this language requires a sort identifying a register (`RegId`), for the value stored in a register (`Register`), and for the set of such values (`Registers`). Note the use of the keyword `subsort` indicating that `Register` is a particular case of `Registers`:

```
sorts RegId Register Registers .          subsort Register < Registers .
```

We define now values for these sorts as follows: `RegId` are built with the constructor `reg`, which receives a natural number; a `Register` is just a pair of a `RegId` and an integer (underscores are just placeholders); finally, we can have either the empty `Registers` (`mtReg`) or the juxtaposition of elements, which is commutative and associative and has `mtReg` as identity:

```
op reg : Nat -> RegId [ctor] .
op <_,_> : RegId Int -> Register [ctor] .

op mtReg : -> Registers [ctor] .
op __ : Registers Registers -> Registers [ctor assoc comm id: mtReg] .
```

We can also define functions on these sorts. We specify the function `_[_]` for looking-up a value in the registers (note that it returns `0` if it is not initialized) and `update` for updating the memory:

```
op _[_] : Registers RegId -> Int .
eq [lu1] : (< R, I > RS) [R] = I .
eq [lu2] : RS [R] = 0 [owise] .

op update : RegId Int Registers -> Registers .
eq [upd_int1] : update(R, I, < R, I' > RS) = < R, I > RS .
eq [upd_int2] : update(R, I, RS) = < R, I > RS [owise] .
```

The sort for the long-term memory, `Memory`, is defined in a similar way. It is also worth presenting the syntax for instructions and the whole system that will be used when defining the semantics. Instructions have sort `Ins` and their syntax depends on the specific instructions. For example, the instruction for adding two registers and storing the result in a third one is defined below. We will infer later the direction of this instruction, that is, we identify which one is "the third register." Finally, the complete system has sort `System` and puts

together a list of instructions (`Instructions`), the state of the registers (sort `Registers`), the state of the memory (`Memory`), and the program counter (of a predefined sort `Nat`):

```
op add_,_,_ : RegId RegId RegId -> Ins [ctor] .
op [_|_|_|_] : Instructions Registers Memory Nat -> System [ctor] .
```

Once the signature is established, the semantics are defined by means of rewrite rules. Rewrite rules mimic the behavior specified by the inference rules in the formal semantics by executing the premises in the conditions and the conclusion in the body of the rule. The rule labeled `[add]` below defines the expected behavior of the add instruction: retrieves the values stored in the second and the third register parametrizing the instruction, adds them, and stores the thus obtained value in the first register:

```
crl [add] : [IIL | RS | M | PC] => [IIL | RS' | M | PC + 1]
 if (add RI, RI', RI'') := getIns(IIL, PC) /\
    I := RS [RI'] /\
    I' := RS [RI''] /\
    RS' := update(RI, I + I', RS) .
```

Note that we use matching conditions (`:=`) to indicate that the pattern in the lefthand side matches the term in the righthand side, once it has been reduced by means of equations. This condition binds the free variables (that is, the variables that did not appear in the lefthand side of the rule or in previous matching conditions) to the appropriate values.

4 Inferring Memory Policies

We describe next the refinement that extends our previous work on discovering side-effect constructs in a programming language starting with the semantics specification of the considered language [26]. There, we show a generic intraprocedural slicing process where the generic aspect is given by the inference of what we call side-effect language constructs, i.e., the instructions that determine memory changes. To achieve this, we construct a so called *hyper-tree*, whose nodes are sets of rewrite rules and edges are dependencies between these rules. As such, we are able to infer which constructs are going to possibly produce memory updates by following the paths in the hyper-tree from the root to the leaves. We can see our current work as a trickle-up in this hyper-tree. Namely, at the leaves level we extract information regarding the source-destination relation of memory updates and we propagate this relation up in the hyper-tree at the level of the language constructs. Note that the method in [26] produces an over-approximation of the side-effect constructs, which we now refine not in terms of cutting out elements from the resulting set, but by enriching the information contained in this set with data-flow direction.

Hence, we present in this section the ideas underlying our framework, illustrating them on the Maude semantics of an assembly language. The results are

equally applicable to other rewriting-based semantics, like the one for WhileF language that we describe in [3]. Next, we elaborate on the semantics of the assembly language and how to infer memory policies like the "direction of a memory update" for programming language constructs.

We assume that the sorts for the memory (`Registers` and `Memory` in the example in Sect. 3) are provided by the user, while the rest of information is inferred by the system. It is important to state that these inferences work under a natural assumption: memory sorts are composed by tuples mapping program variables into values, possibly via addresses.

4.1 Maude Slicing

In order to narrow down the source of the changes, we first apply term-slicing to the equations and rules in the semantics. Slicing for Maude specifications (named in this paper *term-slicing* to differentiate it from the standard program slicing component present in our work) is already used for improving the results from Maude model checker [2]. We use here a simpler approximation of term-slicing that traces back the source of a given set of variables by adding to this set the variables involved in their generation. This approximation is a syntactic procedure for computing dependencies in a single rule/equation by taking into account that variables can be bound in the lefthand side of matching conditions (`:=`) and in the righthand side of rewrite conditions (`=>`). Hence, starting from an initial set of variables of interest $\mathcal{V}$, we traverse the conditions following a bottom-up strategy and, when a variable $v \in \mathcal{V}$ is bound by these conditions we add all the variables in the "opposite" side (hence in the righthand side of matching conditions and the lefthand side of rewrite conditions) to $\mathcal{V}$. For example, let us assume we have a rule as follows

```
crl f(X, Y, Z) => g(h(B, A3), Z)
 if X >= 3 /\
    A1 := aux1(X, Y) /\
    B := other_fun(X, Y, Z) /\
    aux2(Y) => A2 /\
    A3 := aux3(A1, A2) .
```

and we want to trace back `A3`, since it modifies the memory. In the rule above, the condition `A3 := aux3(A1, A2)` indicates that the value in `A3`, the variable in the term-slicing set, depends on the value of both `A1` and `A2` used in function `aux3`, so they are both included in the term-slicing set. The previous condition, `aux2(Y) => A2`, indicates that `A2` depends on `Y`, and hence it is included in the slicing set. Note however that the condition `B := other_fun(X, Y, Z)` does not produce any change in the term-slicing set because `Y` is only used and not changed by this condition, hence `B` is not included in the term-slicing set. The condition `A1 := aux1(X, Y)` adds `X` (and `Y`) into the term-slicing set. Finally, the first condition, `X >= 3`, has no effect because it is not a matching or rewriting condition. From this analysis we find that `A3` depends on `Y` and `X` from the lefthand side of the rule; similarly, we can decide the dependencies of any term.

Data: A specification and the sorts for the memory $\mathcal{M}$.
Result: Set of sorts for values $\mathcal{V}$.
$\mathcal{V} = \emptyset$;
foreach *constructor* $c(s_0, \ldots, s_n, v)$ *of sort* $S, n \geq 0, S \in \mathcal{M}$ **do**
  foreach *function* $f : ar \rightarrow v$ **do** // Explicit inference
    if $S \in ar$ **then** $\mathcal{V} = \mathcal{V} \cup \{v\}$;
  end
  foreach *rule* $l \rightarrow r$ *if cond* **do** // Implicit inference
    $vm = varsOfSortMemory(r, \mathcal{M})$;
    $vs = slicing(l, cond, vm)$;
    $\mathcal{V} = \mathcal{V} \cup getVarsInConstructor(vs, c)$;
  end
end

Algorithm 1. Algorithm for inferring the sort for the values

4.2 Inferring the Sorts for the Values in the Memory

We now emphasize on the settings characterizing the memory part in the class of language semantics specifications that we consider. As previously mentioned, we assume that the memory component of the specification is connecting the program variables to their current values, either directly as in a simplified memory model, or via a chain of "addresses" as in a more accurate representation of the machine. Note that by "values" we understand those terms building the memory that are used by the semantics to modify the state, while by "addresses" we understand those terms used to access the values. We now show how to obtain the sorts for the values stored in the memory given by the, e.g., `Registers` sort in the considered language specification.

We present the algorithm for inferring these sorts in Algorithm 1. We traverse the constructors for the sorts specifying the memory and check all the possible outcomes for them. The first inner loop deals with *explicit* access to the memory: functions that receive the memory and return one of the sorts used in the constructor.[1] This case is illustrated by the function look-up (`_[_]`) in Sect. 3. The look-up function is defined by the equations `[lu1]` and `[lu2]` and it extracts a term of sort `Int`, which is used to build a `Register`, which is, in turn, a subsort of the sort of a specific part of the memory, i.e., `Registers`. Since this function is used in the semantics of the language, we infer that `Int` is the sort of a possible value.

We can also find *implicit* access to the memory: patterns in the lefthand side of rules or in matching/rewrite conditions can be used to retrieve values from the memory, as illustrated in the second inner loop of Algorithm 1. In this case, we trace back the variables modifying the memory and keep only those obtained

[1] We have placed the sort v as the last sort in the arity to ease the presentation, but it is not required.

Data: A specification, the sorts for the memory $\mathcal{M}$, and the sorts standing for values $\mathcal{V}$.
Result: Set of functions $\mathcal{F}$ modifying the memory annotated with the variables responsible for the modifications.

$\mathcal{F} = \emptyset$;
foreach *function* $f : s_0, \ldots, s_n \rightarrow s, s \in \mathcal{M}, \exists i.s_i \in \mathcal{V}$ **do** // Explicit inference
  $\mathcal{F} = \mathcal{F} \cup \{f_{s_i}\}$
end
foreach *rule* $l \rightarrow r$ *if cond* **do** // Implicit inference
  $vm = varsOfSortMemory(r, \mathcal{M})$;
  $vs = slicing(l, cond, vm)$;
  $vv = varsOfSortValue(vm, \mathcal{V})$;
  if $vv \neq \emptyset$ **then** $\mathcal{F} = \mathcal{F} \cup \{f_{vv}\}$;
end

Algorithm 2. Algorithm for inferring the functions modifying the memory

from the memory. For example, assume we modify the rule `[add]` from Sect. 3 to avoid the look-up function, obtaining `[addv2]`.[2]

```
crl [addv2] : [IIL | RS | M | PC] => [IIL | RS'' | M | PC + 1]
 if (add RI, RI', RI'') := getIns(IIL, PC) /\
    < RI', I > < RI'', I' > RS' := RS /\
    RS'' := update(RI, I + I', RS) .
```

In this case, we know that it is possible for the memory to be modified (we have a new variable `RS''` for a memory sort), so we consider the term-slicing set to initially contain only this variable. We then trace back its related variables using the technique described in Sect. 4.1, obtaining `RI`, `I`, `I'`, `RS`, `IIL`, and `PC`. We can now filter the obtained term-slicing set and retain only the values in the memory (in this case both `I` and `I'`), which have the sort `Int` that we previously inferred. Note that it is possible to use a matching with unrequited information to make the method above to include some sorts that are not proper values. However, this is not a threat for soundness, because our technique computes over-approximations, so adding a sort that is not memory related will just worsen the granularity of the slice computed later.

4.3 Inferring the Functions Modifying the Memory

At this step we look for the functions that introduce new values into the memory. As presented in Algorithm 2, in this case we can also find both *explicit* and *implicit* access to the memory. Note that the algorithm returns the set of functions annotated with the variables responsible for the effects. The explicit case, shown in the first loop, is easy to detect: we just traverse the operators

[2] We would need extra rules to take care of non-initialized registers, but this is not relevant for the technique.

looking for those creating/modifying the memory (i.e., the memory appears in the coarity). We then trace the source of this modifications by using the slicing technique in Sect. 4.1 to annotate those arguments responsible for the changes and having one of the sorts annotated in the previous step. For instance, the function `update` from Sect. 3 modifies the memory by introducing the element of sort `Int` received as the second parameter.

Note that since `update` is found to modify the memory, then also the `[addv2]` rule modifies the memory since it uses `update` to match the memory variable `RS''`. This connection between the rules and functions that modify the memory is already presented in our previous work [26]. There we describe the construction of a hyper-tree containing in its nodes rules from the language semantics specification while its arcs are given by relations as the one mentioned above. For instance, `[addv2]` is a parent of `[upd_int1]` and `[upd_int2]` in the hyper-tree because it uses the `update` function which is described by the two `[upd_]` rules.

The implicit modifications to the memory, shown in the second loop of Algorithm 2, occur when a rewrite rule modifies the memory directly, i.e., without using any auxiliary function. In this case, we must slice again the rule using the updated memory criteria and keep those variables that have the sort obtained in the previous step. We illustrate this with a third version of the `[add]` rule from Sect. 3, called `[addv3]`.[3]

```
crl [addv3] : [IIL | RS | M | PC] => [IIL | RS'' | M | PC + 1]
 if (add RI, RI', RI'') := getIns(IIL, PC) /\
    < RI, I > < RI', I' > < RI'', I'' > RS' := RS /\
    RS'' := < RI, I' + I'' > < RI', I' > < RI'', I'' > RS' .
```

In this case, the last matching condition updates the memory onsite by using the values `I'` and `I''`. Consequently, `I'` and `I''` are annotated as side-effect sources, i.e., sources of changes in the memory.

4.4 Inferring the Data-Flow Information

By using the results obtained in the previous steps, we have enough information to infer the data-flow relation that is of interest here, i.e., the source-destination relation in the language constructs producing side-effects. As shown in Algorithm 3, we take for each rewrite rule the variables modifying the memory (obtained from either explicit or implicit change) and apply *enriched* slicing to them. This enriched slicing takes into account the assumption stated at the beginning of the section: the memory is composed of cells (tuples) connecting the program variables (or registers in the case of assembly languages) with their values. Hence, when facing a matching condition involving the memory we extend the slicing set to all the elements in the tuple in order to make sure we consider all the "addresses" connecting the program variables with their values. Finally, we need to recognize the instruction being executed. This term is the one that

[3] Note that we would need another rule to deal with the case where `RI` is not initialized, but this does not change the inference.

Data: A specification and the functions modifying the memory $\mathcal{F}$.
Result: Data-flow information $\mathcal{D}$.
$\mathcal{D} = \emptyset$;
foreach *rule* $rl \equiv l \ \rightarrow \ r \ if \ cond$ **do**
  $vs = getVarsFromAnnotations(rl, \mathcal{F})$;
  $vss = slicing(rl, vs)$;
  $ins = getInstruction(rl, vss)$;
  $active = getVars(ins) \ \cap \ vss$;
  $passive = getVars(ins) \setminus vss$;
  $\mathcal{D} = \mathcal{D} \ \cup \ \{ins : active \mapsto passive\}$
end

Algorithm 3. Algorithm for inferring the data-flow information

fulfills the following properties: (i) must be (or depend on) a subterm that contains the complete state, including the memory and any other sort required by the semantics and (ii) contains all the variables from the slicing set not related with the memory. The variables appearing in this term and in the slicing set are responsible for the modifications we are tracing in the memory. Note that many rules can specify the behavior of the same instruction. In this case, we put together all the possible sources of change.

For example, this method infers for the rule `[add]` from Sect. 3, that:

1. The term being executed is `add RI, RI', RI''`, since it is not related to the memory and contains `RI'` and `RI''`, which in turn generate `I` and `I'` from the slicing set.
2. The variables `RI'` and `RI''`, which appear in both the term and the slicing set, modify the rest of the variables (`RI`). Hence, this instruction works from right to left.

The same result is easily obtained for `addv2` and `addv3`. Moreover, note that the same approach can be easily followed to analyze the direction of a standard assignment instruction in any imperative language.

5 Prototype

The ideas presented in the previous sections have been used to extend the slicing tool in [3]. It allows us to apply our generic slicing framework to semantics of imperative languages, like the WhileF language in [3], to languages with mixed data-flow policies, like the assembly language presented in this paper, or to "eccentric" semantics, such as, languages with a left to right assignment statement. The source code of the tool, examples, and more explanations are available at http://maude.sip.ucm.es/slicing/.

The tool is started by loading in a Maude session the `slicing.maude` file available at the webpage. This starts an input/output loop where other Maude modules can be introduced and analyzed. We introduce the semantics for the language, e.g., the assembly language partially presented throughout the paper.

One of the tool's features is to infer the data flow information for the basic language constructs. For example, `add` presented in more detail in Sect. 4 has a right to left direction, i.e., `add R1, R2, R3` stores `R2` + `R3` in `R1`; the direction of the load instruction is from left to right, i.e., `load R1, R2` loads in `R2` the data stored in the memory cell indicated by `R1`; while for the store instruction the tool infers a data flow direction from right to left, i.e., `str R1, R2` stores in the register indicated by `R1` the value in the cell indicated by `R2`.

Because we have at hand an executable semantics, we can use the assembly language semantics to execute the program `pow`, which computes x^y (assuming `x` and `y` are stored in the memory cells `0` and `1`, respectively) and stores the result in the cell `2`:

```
op pow : -> InsList .
eq pow = load R1, R1           *** Load M[0] in R1 (left to right)
         addi R2, R2, 1        *** Add 1 and save it in R2
         load R2, R2           *** Load M[1] in R2
         addi R4, R4, 1        *** Add 1 and save it in R4
'loop    beq  R2, R3, 'out     *** Jump to out when R2 and R3 are equal
         mul  R4, R4, R1       *** Store in R4 the result of R4 * R1
                               *** (function from right to left)
         subi R2, R2, 1        *** Update the counter
         jmp  'loop            *** Jump to loop
'out     addi R5, R5, 2        *** Add 2 and save it in R5
         str  R5, R4           *** Store the value of R4 in M[R5]
                               *** (function from right to left)
         break .               *** end
```

The execution of the program needs the user's input of initial state, e.g., the function `testPow` introduces `3` and `5` in the memory cells `0` and `1`, respectively:

```
op testPow : -> System .
eq testPow = [ pow | mtReg | [0, 3] [1, 5] | 0 ] .
```

Furthermore, in order to obtain the slicing results, the user introduces the sorts corresponding to the memory with the command:

```
Maude> (set side-effect sorts Memory Registers .)
Memory Registers selected as side effect sorts.
```

Once these sorts are set, we can start the slicing process by indicating the program to slice, e.g., `testPow`, and the initial slicing set, e.g., the singleton set containing `R5` the variable storing the final result of `pow`:

```
Maude> (slice testPow wrt R5 .)
```

Note that the initial state of the program is not used by the slicer, which performs static analysis in the true sense, i.e., without using any information from the current state of the program. The program's state is there just to exemplify the executing capabilities of the programming language semantics used in our tool. Now, for slicing, the tool analyzes the list of instructions of the program, given in `pow`, and returns:

– The rules producing side effects, obtained by using the sorts for values and checking which rules modify them:

```
The rules causing side effects are: add addi and load mul muli
                                    str sub subi
```

– The data-flow information for each rule producing side-effects. It is interesting to see the difference between `load` and `store`, as discussed above:

```
The inferred data-flow information is:
- For function add RI:RegId,RI':RegId,RI'':RegId :
Variable(s) RI:RegId are modified by RI':RegId RI'':RegId
- For function load RI:RegId,RI':RegId :
Variable(s) RI':RegId are modified by RI:RegId
- For function str RI:RegId,RI':RegId :
Variable(s) RI:RegId are modified by RI':RegId ...
```

– The final slicing set. In this case, the value stored in the position `R5` is updated with the contents of `R4`, which was in turn updated with the contents in `R1`. Hence, these registers compose the final slicing set:

```
The variables obtained by the slicing process are: R5 R4 R1
```

It is important to remember that the tool works for any programming language whose semantics has been defined in Maude. Hence, we can use the WhileF language from [3] to further test the semantics. Briefly, WhileF is an imperative language with functions and input-output capabilities. Henceforth, the algorithm that infers memory policy information on the WhileF semantics works with the sorts of the underlying memory model: a state sort `ST` mapping variables to values (the global memory), a sort `ESt` for the program environment (the local memory), and a sort for the read/write buffer `RWBUF`. Hence, we can introduce the code from Fig. 1(left) in WhileF as follows:

```
op whileExample : -> Com .
eq whileExample = Read i ; Read j ; s := 0 ; p := 1 ;
                  While Not Equal(i, 0) Do
                   Write (i -. j) ; s := s +. i ; p := p *. i ; Read i .
```

Our tool will traverse the semantics, find the sort of values, and show that the assignment works from left to right. Moreover, it also indicates that the variable related to `p` is just `i`, used in the multiplication:

```
Maude> (slice whileExample wrt p .)
The inferred data-flow information is:
- For function X:Var := e:Exp :
Variable(s) X:Var are modified by e:Exp
The variables obtained by the slicing process are: p i
```

6 Concluding Remarks and Future Work

In this paper we used formal language semantics to infer a certain type of memory policy, i.e., data-flow information for language constructs which produce memory updates. This inference has allowed us to improve on the genericity of our slicing technique [3] and to make another step towards a complete design of a automatized semantics-based slicing tool. Moreover, this addition to the slicing tool allowed testing the tool on other class of programming language specifications such as the assembly languages.

We are currently investigating the automatic inference of other slicing prerequisites for interprocedural methods such as the automatic deduction of function call/returns and the inference of their parameter passing patterns. These improvements would further automatize our generic slicing tool as the language designer would roughly need only to define the semantics of the programming language, to give the input program, and the slicing criterion, then our generic slicer will generate all the necessary information for slicing. From a language perspective, we aim to extend the language, for example with pointers and hence, to be able to accommodate more complex memory policies, based on a more refined memory model. Note that the addition of pointers to our framework will allow us to use arrays as well. Finally, our aim is to introduce concurrency in the framework, so we can cover and test out proposed methodology on a larger and significant class of programming languages.

Acknowledgments. We thank the anonymous reviewers for their valuable comments and suggestions, which greatly improved the quality of the paper.

References

1. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Live heap space analysis for languages with garbage collection. In: Proceedings of the International Symposium on Memory Management, ISMM 2009, pp. 129–138. ACM (2009)
2. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Using conditional trace slicing for improving Maude programs. Sci. Comput. Program. **80**, 385–415 (2014)
3. Asăvoae, I.M., Asăvoae, M., Riesco, A.: Towards a formal semantics-based technique for interprocedural slicing. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 291–306. Springer, Heidelberg (2014)
4. Baufreton, P., Heckmann, R.: Reliable and precise WCET and stack size determination for a real-life embedded application. In: ISoLA 2007, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Revue des Nouvelles Technologies de l'Information, pp. 41–48. (2007)
5. Bethke, I., Klop, J.W., de Vrijer, R.C.: Descendants and origins in term rewriting. Inf. Comput. **159**(1–2), 59–124 (2000)
6. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theor. Comput. Sci. **236**(1–2), 35–132 (2000)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)

8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Symposium on Principles of Programming Languages, POPL 1977, pp. 238–252. ACM (1977)
9. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Proceedings of the Symposium on Principles of Programming Languages, POPL 2012, pp. 533–544. ACM (2012)
10. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
11. Farzan, A., Meseguer, J., Roşu, G.: Formal JVM code analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 132–147. Springer, Heidelberg (2004)
12. Ferrara, P.: A generic static analyzer for multithreaded java programs. Softw., Pract. Exper. **43**(6), 663–684 (2013)
13. Field, J., Tip, F.: Dynamic dependence in term rewriting systems and its application to program slicing. Inf. Softw. Technol. **40**(11–12), 609–636 (1998)
14. Hills, M., Rosu, G.: A rewriting logic semantics approach to modular program analysis. In: Proceedings of the International Conference on Rewriting Techniques and Applications, RTA 2010, LIPIcs, vol. 6, pp. 151–160. (2010)
15. Hind, M., Pioli, A.: Evaluating the effectiveness of pointer alias analyses. Sci. Comput. Program. **39**(1), 31–55 (2001)
16. Huet, G.P., Lévy, J.: Computations in orthogonal rewriting systems, I. In: Computational Logic - Essays in Honor of Alan Robinson, pp. 395–414. (1991)
17. Klop, J.W.: Term rewriting systems from Church-Rosser to Knuth-Bendix and beyond. In: Paterson, M.S. (ed.) Automata, Languages and Programming. LNCS, vol. 443, pp. 350–369. Springer, Heidelberg (1990)
18. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Autom. Reason. **41**(1), 1–31 (2008)
19. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theor. Comput. Sci. **285**(2), 121–154 (2002)
20. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
21. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
22. Pierce, B.C.: Types and Programming Languages. MIT Press, London (2002)
23. Ramalingam, G.: The undecidability of aliasing. ACM Trans. Program. Lang. Syst. **16**(5), 1467–1471 (1994)
24. Regehr, J., Reid, A., Webb, K.: Eliminating stack overflow by abstract interpretation. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 306–322. Springer, Heidelberg (2003)
25. Riesco, A.: Using semantics specified in maude to generate test cases. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 90–104. Springer, Heidelberg (2012)
26. Riesco, A., Asăvoae, I.M., Asăvoae, M.: A generic program slicing technique based on language definitions. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 248–264. Springer, Heidelberg (2013)
27. Rosu, G., Stefanescu, A.: Matching logic: a new program verification approach. In: Proceedings of the International Conference on Software Engineering, ICSE 2011, pp. 868–871. ACM (2011)

28. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-cc multiprocessor machine code. In: Proceedings of the Symposium on Principles of Programming Languages, POPL 2009, pp. 379–391. ACM (2009)
29. Venkitaraman, R., Gupta, G.: Static program analysis of embedded executable assembly code. In: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2004, pp. 157–166. ACM (2004)