Submitted to the International Conference on Rewriting Techniques and Applications
http://rewriting.loria.fr/rta/

A REWRITING LOGIC SEMANTICS APPROACH TO MODULAR PROGRAM ANALYSIS

MARK HILLS [1] AND GRIGORE ROŞU [2]

[1] Centrum Wiskunde & Informatica
Science Park 123, 1098 XG Amsterdam, The Netherlands
E-mail address: Mark.Hills@cwi.nl
URL: http://www.cwi.nl

[2] Department of Computer Science, University of Illinois at Urbana-Champaign
201 N. Goodwin Av., Urbana, IL 61801, USA
E-mail address: grosu@cs.uiuc.edu
URL: http://fsl.cs.uiuc.edu

Abstract. The K framework, based on rewriting logic semantics, provides a powerful logic for defining the semantics of programming languages. While most work in this area has focused on defining an evaluation semantics for a language, it is also possible to define an abstract semantics that can be used for program analysis. Using the SILF language (Hills, Serbanuta and Rosu, 2007), this paper describes one technique for defining such a semantics: policy frameworks. In policy frameworks, an analysis-generic, modular framework is first defined for a language. Individual analyses, called policies, are then defined as extensions of this framework, with each policy defining analysis-specific semantic rules and an annotation language which, in combination with support in the language front-end, allows users to annotate program types and functions with information used during program analysis. Standard term rewriting techniques are used to analyze programs by evaluating them in the policy semantics.

1. Introduction

Programs compute by manipulating different kinds of *explicit* data made available by the language, like integers, objects, lists, functions, or strings. Some of this data may also have *implicit* properties, important to the correctness of the program but impossible to represent directly in the language. For example, many languages have no way to indicate that a variable has been (or must already be) explicitly initialized, or that a reference or pointer never contains null. Some languages also leave the types of values and variables implicit, providing no syntax to indicate that certain types are expected at given points in the

1998 ACM Subject Classification: F.3.2 [Semantics of Programming Languages]: Program Analysis.
Key words and phrases: K, rewriting logic semantics, program analysis.
Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

program. Domain-specific examples of implicit properties are also common. One compelling example, commonly used in scientific computing applications, is units of measurement [nis], where program values and variables are assumed to have specific units (meters, kilograms, seconds, etc) at specific points in the program or along specific execution paths.

These implicit properties of program data give rise to implicit *policies*, or rules about how this information can be manipulated. For instance, one may require that variables be initialized on all paths before being read, or that only non-null pointers can be assigned to other non-null pointers. Languages with implicit types generally still place type restrictions on operations such as arithmetic, where only values representing numbers can be used. Programs that use units of measurement must adhere to a number of rules, such as requiring two operands in an addition or comparison operation to have the same unit, or treating the result of a multiplication operation as having a unit equal to the product of the units of the operands (e.g., given `meter` and `second`, the resulting unit would be `meter second`).

Because these properties are hard to check by hand, a number of techniques have been developed to allow implicit properties to be either inferred or stated explicitly in a program. In this paper we focus on the use of *annotations*, either given by "decorating" program constructs with type-like information (type annotations) or by including additional information in special language constructs or inside comments (code annotations). Many systems that use annotations are designed with specific analysis domains in mind; those that are more general often support either type or code annotations, but not both, or provide limited capabilities to adapt to new domains. In this paper we present a solution designed to overcome these limitations: *policy frameworks*. Policy frameworks support the use of type and code annotations through an augmented language front-end, with each analysis *policy* defining its own annotation language, specialized to the domain under analysis. Program analysis is then based on program evaluation in an abstract rewriting logic semantics of the programming and annotation languages. Principles developed during work on the rewriting logic semantics project are used to ensure that the semantics is modular, allowing a large core of the framework to be reused across policies.

The remainder of this paper is organized as follows. As background, Section 2 provides an introduction to rewriting logic semantics and K, a rewrite-based formalism for language semantics. Section 3 then presents a policy framework for the SILF programming language along with two policies. SILF has been chosen because it is complex enough to show many common features of programming languages, but simple enough to allow policy frameworks to be understood in isolation from the language, something that is more difficult for languages such as C. Finally, Section 4 discusses related work, while Section 5 concludes.

2. Rewriting Logic Semantics and K

Equational logic has long been seen as a viable formalism for defining the semantics of sequential programming languages [Gog77, Ber89, Gog96]. Rewriting logic extends this by providing a formalism for defining the semantics of nondeterministic and concurrent languages, leading to an area of research known as rewriting logic semantics [Mes04, Mes07]. One specific style of rewriting logic semantics is computation-based rewriting logic semantics [Mes07], hereafter referred to as RLS.

RLS defines the semantics of a programming language as a rewrite theory. Terms formed over the signature of the theory are used to represent the program *configuration*, made up of the current program and auxiliary entities such as environments, stores, etc.

Rules and equations are then used to transition between configurations, with equations used to define sequential language features and rules used to define nondeterministic and concurrent language features. The configuration is defined as a nested multiset: individual parts of the configuration (*configuration items*) can be repeated and can also be nested inside other items, allowing the flexibility to represent language features such as multiple threads (repetition), each with local state (nesting). Multiset matching is used so that configuration items do not need to be named in a specific order in equations and rules; matching also allows unused parts of the configuration around the matched subterm to be elided, allowing equations and rules to remain unchanged even when the surrounding configuration changes.

One often-used configuration item, `k`, holds the current computation. Computations in `k` are lists; each item in the list is referred to as a computation item, each of which represents an individual task or piece of information. The head of the list can be seen as the "next" task, with the tail containing tasks that will be computed later. Instead of using "`,`" as the list separator, an arrow, written in text as `->` and mathematically as $\curvearrowright$, is used, hopefully providing some added intuition: do this (ci_1), then that (ci_2), then that (ci_3), etc, until finished (no items are left):

$$ci_1 \curvearrowright ci_2 \curvearrowright ci_3 \curvearrowright ... \curvearrowright ci_n$$

The equations and rules used to define the semantics often break up computations into smaller pieces, which are then put at the head of the computation to indicate that they need to be computed first before the overall computation can continue. Since computations in RLS are first-order terms, they can also be manipulated as a whole, such as by saving the current computation for evaluation later (such as for call/cc or coroutines).

Figure 1 shows several examples of equations. The first provides the semantics for the statement `E ;`, saying that this is defined as the result of evaluating the expression `E` and then discarding the result. Note that `exp(E)` is placed "on top of" `discard` in the computation, meaning that it will be evaluated first, with the expectation that it will produce a value. The second equation provides a semantics for names used as expressions: `X` is looked up to retrieve its current value. The third equation provides the semantics for assignment: `E` is evaluated, and the resulting value is assigned to `X` using `assignTo`. Finally, the fourth equation defines the semantics of `assignTo`. `Env` and `Mem` are both multisets of pairs, defined equationally as maps – `Env` from names to locations, `Mem` from locations to values. The equation states that the result of assigning value `V` to name `X` is a configuration where the value `V'` held at location `L` in `Mem` – location `L` is assigned to `X` by `Env` – is replaced with the new value `V`.

```
eq stmt(E ;) = exp(E) -> discard .
eq exp(X) = lookup(X) .
eq stmt(X <- E ;) = exp(E) -> assignTo(X) .
eq k(val(V) -> assignTo(X) -> K) env([X,L] Env) mem([L,V'] Mem) =
   k(                         K) env([X,L] Env) mem([L,V ] Mem) .
```

Figure 1: RLS Semantics with Equations

K. K [Ros07], based on RLS, provides additional notation for defining the semantics of a language. K configurations are defined identically to those in RLS, with each configuration item called a K *cell*. The cells are given in K rules using an XML-like notation, with an opening cell "tag", like $\langle k \rangle$ and a closing tag like $\langle / k \rangle$. Rules in K are defined similarly to rules and equations in RLS, but with a number of notational conveniences. Figure 2 shows the step-by-step results of a number of individual transformation steps to convert the last RLS equation in Figure 1 (augmented with support for threads) into a K rule.

$$\langle t\rangle\ \langle k\rangle\ X\ \leftarrow\ V \curvearrowright K\ \langle/k\rangle\langle env\rangle\ (X,L)\ Env\ \langle/env\rangle\ TS\ \langle/t\rangle \quad \langle mem\rangle\ (L,V')\ Mem\ \langle/mem\rangle \rightarrow$$
$$\langle t\rangle\ \langle k\rangle\ K\ \langle/k\rangle\langle env\rangle\ (X,L)\ Env\ \langle/env\rangle\ TS\ \langle/t\rangle \quad \langle mem\rangle\ (L,V)\ Mem\ \langle/mem\rangle \tag{2.1}$$

$$\langle k\rangle\ X\ \leftarrow\ V \curvearrowright K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,V')\ Mem\ \langle/mem\rangle \rightarrow$$
$$\langle k\rangle\ K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,V)\ Mem\ \langle/mem\rangle \tag{2.2}$$

$$\langle k\rangle\ X\ \leftarrow\ V \curvearrowright K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,\_)\ Mem\ \langle/mem\rangle \rightarrow$$
$$\langle k\rangle\ K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,V)\ Mem\ \langle/mem\rangle \tag{2.3}$$

$$\langle k\rangle\ X\ \leftarrow\ V\ \ ...\langle/k\rangle \quad \langle env\rangle...\ \ (X,L)\ \ ...\langle/env\rangle \quad \langle mem\rangle...\ \ (L,\_)\ \ ...\langle/mem\rangle \rightarrow$$
$$\langle k\rangle\ \cdot\ \ ...\langle/k\rangle \quad \langle env\rangle...\ \ (X,L)\ \ ...\langle/env\rangle \quad \langle mem\rangle...\ \ (L,V)\ \ ...\langle/mem\rangle \tag{2.4}$$

$$\langle k\rangle\ \underset{\cdot}{X\ \leftarrow\ V}\ ...\langle/k\rangle\ \langle env\rangle...\ \ (X,L)\ \ ...\langle/env\rangle\ \langle mem\rangle...\ \ (L,\underset{V}{\_})\ \ ...\langle/mem\rangle \tag{2.5}$$

Figure 2: Converting RLS to K

The first step of the transformation is shown in Rule 2.1. The main change in this first step is the use of the XML-like cell notation and the replacement of `val(V) -> assignTo(X)` with $X \leftarrow V$. The second step, shown in Rule 2.2, is the removal of the context needed only for matching the configuration structure. The removal of this context allows the rule to be more modular, since it is not then tied to a specific configuration, only to the configuration pieces used directly by the rule. The third transformation step, shown in Rule 2.3, is the replacement of variables given on the left-hand side of a rule by underscores if they are not otherwise used (in conditions or on the right-hand side). This is used to replace V' with an underscore, since it is not used elsewhere in the rule.

Since matching against lists and sets is used quite often, it is also helpful to have special notation for both lists and sets. In K, this is indicated by using "...", with a "..." at the start or end of a cell indicating a list match ("..." at the start indicates that one is matching the list tail, while "..." at the end indicates that one is matching the head), and "..." at both ends of the cell indicating a set or multiset match. In the fourth transformation step, Rule 2.4 shows the transformation of Rule 2.3 to use this notation. Finally, since some of the information in Rule 2.4 is redundant it can be removed by switching to a special rule format, where the parts of a term that are changed are underlined, with the changes to the term then placed below the underlines. The result of applying this fifth transformation to Rule 2.4 is shown in Rule 2.5. In the `k` cell, the $\cdot$ is now under $X\ \leftarrow V$, while V is now under $\_$. Since both the environment and the first element (L) of the pair in *mem* do not change, each only needs to be written once.

3. The SILF Policy Framework

SILF, the Simple Imperative Language with Functions, was introduced in earlier work by the authors [Hil07, Hil09]. Here, we extend the language presented in this earlier work with a policy framework and with two policies: one for type checking programs in SILF, using type annotations; and one for checking the unit safety of programs, using type and

code annotations to indicate the units associated with program variables and values. To give an idea of the size of the specification, the policy framework is made up of 281 operators and 284 equations across 49 modules; the type checking policy of 100 equations and 17 operators across 7 modules; and the unit checking policy of 273 equations and 52 operators across 38 modules [1]. The SILF Policy Framework is available for download or online use at the SILF Policy Framework homepage [Hil].

3.1. Adding a Policy Framework to SILF

Adding a policy framework to SILF requires adding a policy-aware front end, a core abstract language semantics, generic analysis support, and the individual analysis policies. Since compatibility with existing SILF code is not an issue, to add a policy framework to SILF annotation support has been added directly to the language, versus (as was done in the C Policy Framework [Hil08] to maintain compatibility with existing C compilers) adding support through source comments. In the extended SILF syntax, type annotations are identifiers with a leading `$`, like `$int` or `$meter`. Type variables are given with similar syntax: `$$`, instead of just `$`, like `$$X`. Type identifiers and variables are used in standard type positions, like on variables and formal parameters. Code annotations are given in syntax extensions for invariants on loops (for both `while` and `for` loops), assume and assert statements, and, in function declarations, function contracts with preconditions, postconditions, and modifies (which identifies the globals changed by a function). The code annotation, an arbitrary string, is actually parsed by the policy. Each code annotation includes a *policy tag*, identifying the policy associated with the annotation language used in the annotation. This policy tag is just an identifier, and is given before the annotation, like pre(*UNITS*). This allows annotations for multiple policies to be present in the program source at once.

The core semantics includes the original SILF abstract syntax, extended with the new type and code annotation constructs mentioned above; and the configuration (i.e., K cells). The original dynamic semantics can be viewed as a special policy which ignores the additional constructs, with evaluation over a concrete, versus abstract, domain. Extensions to the semantics to support analysis include modules providing: basic logical connectives for the annotation language; pretty printing capabilities over the abstract syntax for generating error messages; support for type annotation variables with limited forms of polymorphism; additional K cells for analysis information; and operators for working with these extensions. K's modularity allows new cells to be added without requiring changes to existing rules, making it easy to extend the state with new analysis information, such as line numbers (cell *currLn*), a copy of the environment current at function entry (cell *old*), and error messages generated by the analysis (cell *log*). Many SILF language features are also given a default generic semantics, with special computation items used to indicate "hooks" whose behavior is defined by individual policies. For instance, the result of an addition expression is left up to the individual policy, since a type checking policy would have different correctness requirements than a units of measurement checking policy.

Figure 3 provides several examples of policy-generic semantics rules. Rule 3.1 is the generic rule for assignments. To check an assignment, the semantics first evaluates `X` and `E`. The computation item *checkAssign* then determines, in a policy-specific manner, whether the value of `E` can be assigned to `X` (based, for instance, on the current assigned value

[1] This actually defines three progressively more complex policies, with no single policy using all the operators, equations, or modules defined.

$$\langle k \rangle \frac{X := E}{(X, E) \curvearrowright checkAssign} \ldots \langle / k \rangle \tag{3.1}$$

$$\langle k \rangle \frac{X[E] := E'}{(X, E, E') \curvearrowright checkArrayAssign} \ldots \langle / k \rangle \tag{3.2}$$

$$\langle k \rangle \frac{\mathsf{if}\ E\ \mathsf{then}\ Dt\ St\ \mathsf{else}\ Df\ Sf\ \mathsf{fi}}{E\ \curvearrowright checkIfGuard\ \curvearrowright if(Dt \curvearrowright St \curvearrowright Env, Df \curvearrowright Sf \curvearrowright Env)} \ldots \langle / k \rangle\ \langle env \rangle Env \langle / env \rangle \tag{3.3}$$

Figure 3: SILF Abstract Statement Semantics

or any annotations given on the declaration). Rule 3.2 is similar, but also evaluates the array index expression, and then uses computation item *checkArrayAssign* to ensure the assignment meets all requirements for the policy. The final rule shown, Rule 3.3, shows the generic semantics for a conditional. The conditional guard, `E`, is evaluated first; the computation item *checkIfGuard* will then check the value in a policy-specific manner. The computations stored as part of the *if* computation item will then be used to check both the then and else branches, with each computation handling the declarations and statements along that branch and restoring the environment to that active at the start of the conditional, maintaining block scoping.

The most challenging part of the generic semantics deals with handling function calls and function call sites. During analysis, each function is executed using the policy semantics. Any preconditions are first assumed correct, with postconditions verified at each function return. Since checking is static, call sites are modeled using a computation representing a summary of the called function's behavior. Any preconditions of the called function are first checked, using the actual parameter values in place of the formal parameters in the preconditions; the modifies clauses of this called function are then evaluated, generally setting any modified globals to policy-specific unknown or random values. Finally any postconditions on the called function are evaluated, with the results assumed to hold. Full details of this process can be found with the definition of the framework [Hil].

3.2. Defining A Type Checking Policy for SILF

To define a type checker for SILF as a policy, the first step is to define the analysis domain for types. The values in this domain are shown in Figure 4. Since this policy only uses type annotations, no separate code annotation language needs to be given.

```
sort BaseType .
subsort BaseType < Type .
ops $int $bool : -> BaseType .
op $array : BaseType -> Type .
op $notype : -> Type .
```

Figure 4: SILF Types Domain

The second part of defining the policy is defining the analysis-specific semantics for type checking. These rules generally follow the dynamic semantics rules closely, with the addition that error checking logic has been added to catch errors that, dynamically, led to stuck states. For instance, Figure 5 shows the original integer addition rule for the SILF dynamic semantics, Rule 3.4, as well as two typing rules for addition. The first typing rule, Rule 3.5, indicates the expected scenario: each operand is of type `$int`, with the entire operation also of type `$int`. The second, Rule 3.6, is an error case; at least one of the types is not `$int`. To handle this, the policy code generates an error message (abstracted here as *msg*) of severity 1 (an error) using `issueWarning`. It

$$i_1 + i_2 \to i, \text{ if } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{3.4}$$

$$(\$int, \$int) \curvearrowright plus \to \$int \tag{3.5}$$

$$\langle k \rangle \frac{(t, t') \curvearrowright plus}{issueWarning(1, msg) \curvearrowright \$int} ... \langle /k \rangle, \quad \text{if } t =/= \$int \text{ or } t' =/= \$int \tag{3.6}$$

$$\texttt{if } true \texttt{ then } Kt \texttt{ else } Kf \to Kt \tag{3.7}$$

$$\$bool \curvearrowright \texttt{if}(Kt, Kf) \to Kt \curvearrowright Kf \tag{3.8}$$

Figure 5: SILF Type Checking Policy Rules

also returns `$int` as the result type to prevent a cascade of additional errors being triggered by this one type error.

Figure 5 also provides a comparison between the rules for conditionals in the dynamic and the type checking policy semantics. Rule 3.7, part of the dynamic semantics, selects the `then` (*Kt*) branch in the case of a true condition (the similar else branch rule is not shown). Rule 3.8, part of the policy semantics, makes sure the condition evaluates to a boolean; after this check, the then branch and the else branch are both analyzed. Another rule, not shown, handles the case where the condition does not evaluate to `$bool`, using `issueWarning` like in Rule 3.6 to issue an error message and then, like in Rule 3.7, checking both branches of the conditional. Similar rules are used to define most of the features of the language.

```
function $int f($int x)                1
begin                                  2
  return x + 1;                        3
end                                    4
function $int main(void)               5
begin                                  6
  var $int x;                          7
  x := 3;                              8
  x := f(x);                           9
  x := f(x,x);                         10
  if x then                            11
    write 1;                           12
  fi                                   13
  if (x < 5) then                      14
    write 1;                           15
  else                                 16
    write false;                       17
  fi                                   18
end                                    19
```

Figure 6: Checking Types

Figure 6 shows an example of a program with type errors. Running the type checking policy over the program, the following error messages are generated:

```
ERROR on line 10: Type failure: too many arguments provided in call to function f.
ERROR on line 11: Type failure: expression x should have type $bool, but has type $int.
ERROR on line 17: Type failure: write expression false has type $bool, expected type $int.
```

In the first error, function `f` expects one parameter but is given two. In the second, the conditional expression should be a boolean, but instead an integer is provided, and unlike in languages such as C no automatic coercion is performed. In the final error message, the expression given to the write statement should be an integer, but is instead a boolean. The policy pretty printer, part of the generic analysis support defined for the framework (the basic pretty printer is actually shared between frameworks, but most of the logic is language specific), is used to generate the error messages, and is extended by the policy to correctly print the annotations.

$$(u, u) \curvearrowright plus \to u \tag{3.9}$$

$$\langle k \rangle \frac{(u, u') \curvearrowright plus}{issueWarning(1, msg) \curvearrowright \$fail} ... \langle / k \rangle, \quad \text{if } u =/= u' \text{ and } u' =/= \$cons \tag{3.10}$$

$$(u, u') \curvearrowright times \to u\ u' \tag{3.11}$$

$$V \curvearrowright \texttt{if}(Kt, Kf) \to Kt \curvearrowright Kf \tag{3.12}$$

$$\langle k \rangle \underline{(u, u') \curvearrowright checkAssign}\ ... \langle / k \rangle, \quad \text{if } u == u' \text{ or } u' == \$cons \tag{3.13}$$

$$\langle k \rangle \frac{(u, u') \curvearrowright checkAssign}{issueWarning(1, msg)} ... \langle / k \rangle, \quad \text{if } u =/= u' \text{ and } u' =/= \$cons \tag{3.14}$$

Figure 7: SILF UNITS Policy Rules

3.3. Defining a Units Policy for SILF

The UNITS policy for SILF is similar to that defined for C in the C Policy Framework (CPF) [Hil08], and is only presented at a high level here to show the similarity to the rules for the type checking policy. The complete policy is available for download on the SILF Policy Framework site [Hil].

A program is considered unit safe if it properly follows a number of unit rules, such as only adding values with matching units. Figure 7 shows several UNITS rules. The first, Rule 3.9, is for addition, where, if both units match, the result is the same unit; Rule 3.10 is an error case for addition, where the units don't match and the second unit isn't a constant (which can be converted to any unit). Rule 3.11 is a rule for multiplication, where the resulting unit is the product of the operand units. Rules 3.12, 3.13, and 3.14 are rules for statements. Rule 3.12 handles conditionals, and is similar to Rule 3.8, except there is no need to check the value computed by the guard – any errors found in the guard expression will have already been reported, and the guard is not expected to have a specific unit (a more stringent requirement would be to enforce that the guard has no unit, but that is not done here). Rules 3.13 and 3.14 then show the regular and error cases for assignment. In Rule 3.13, the assignment is safe if the value being assigned either has the same unit or is a constant; in Rule 3.14, this condition does not hold, so an error message is issued.

4. Related Work

Many different tools and techniques have been developed around the use of annotations for program analysis. The earliest precursor to the work presented here was developed as a prototype to check the unit safety of programs written in BC [Che03]. JML [Bur03], the Java Modeling Language, provides support for code annotations, and has been used in a number of analysis tools, such as tools for runtime and static analysis. Spec# [Bar05] extends the C# language with support for code annotations and several type annotations. Eiffel [Mey88] includes direct language support for code annotations such as preconditions (require) and postconditions (ensure). None of these systems provide the same extensibility

of type annotations as seen with both the CPF and the SILF policy framework, while extensions to the provided code annotation languages (where allowed) are formalized in first-order logic.

Specifically for C, a number of annotation-based systems have been developed. LCLint [Eva94], now Splint, uses program annotations to detect potential errors in C programs, and provides limited abilities to add new annotations by allowing attributes and constraints to be defined for various C language objects. C-UNITS [Ros03], another conceptual precursor to the CPF, uses annotations to check unit safety for a limited subset of C, but is not extensible, offering no clear way to either support other analysis domains or cover unsupported features of C. Caduceus [Fil04, Fil07] provides an annotation language similar to JML; programs are verified by transforming them into a simpler language, called Why, which is then further processed to generate proof tasks for various theorem provers. Frama-C [fra] provides an extensible analysis framework, with various analyses built in OCaml as "plugins" to the core Frama-C tool. Frama-C uses the ACSL annotation language [Bau08], which is based on the annotation language used in Caduceus. To support new concepts they must be formalized in first-order logic using "logic specifications", and type annotations are not supported. Systems targeted at specific domains include VCC [Coh08], for verification of concurrent C programs, and HAVOC [Cha07], aimed at programs, such as device drivers, that perform low-level memory manipulation. CQUAL [Fos99] provides support for user-defined type annotations, referred to as type qualifiers, but cannot natively support some complex domains like units. It also does not support code annotations, such as function contracts. The current version of CPF includes new functionality over that discussed in earlier work [Hil08], including support for type annotations and `modifies` clauses and more complete support for heap-allocated values in C.

5. Conclusions

In this paper we introduced *policy frameworks*, a flexible, modular technique for adding new analysis policies and annotation languages. We presented an implemented policy framework for SILF, a simple imperative language, along with two examples of existing policies for SILF, one for types and one for units of measurement. These policies illustrate the reuse within a policy framework of the policy core, while also sharing some framework functionality with the CPF, showing that reuse across frameworks is possible as well.

In the future, we plan to extend the same concepts used here to other programming languages, potentially Java or OCaml. Of special interest is to see if it would be possible to then extend this technique to multi-language analysis (for instance, to calls from OCaml into C code). We are also working to relate the abstract semantics developed for analysis more closely with parallel efforts to develop concrete K semantics of various languages. Finally, we are investigating integrating current work on policy frameworks with work on Rascal [Kli09b, Kli09a], a language for source code analysis and transformation which should allow analysis support to be developed for significant real-world languages.

References

[Bar05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Proceedings of CASSIS'04*, *LNCS*, vol. 3362, pp. 49–69. Springer, 2005.

[Bau08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2008.

[Ber89] Jan A. Bergstra, Jan Heering, and Paul Klint. *Algebraic Specification*. ACM Press, 1989.

[Bur03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In *Proceedings of FMICS'03*, *ENTCS*, vol. 80, pp. 75–91. 2003.

[Cha07] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A Reachability Predicate for Analyzing Low-Level Software. In *Proceedings of TACAS'07*, *LNCS*, vol. 4424, pp. 19–33. Springer, 2007.

[Che03] Feng Chen, Grigore Roşu, and Ram Prasad Venkatesan. Rule-Based Analysis of Dimensional Safety. In *Proceedings of RTA'03*, *LNCS*, vol. 2706, pp. 197–207. Springer, 2003.

[Coh08] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. A Practical Verification Methodology for Concurrent Programs, 2008.

[Eva94] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of FSE'94*, pp. 87–96. ACM Press, 1994.

[Fil04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover Verification of C Programs. In *Proceedings of ICFEM'04*, *LNCS*, vol. 3308, pp. 15–29. Springer, 2004.

[Fil07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Proceedings of CAV'07*, *LNCS*, vol. 4590, pp. 173–177. Springer, 2007.

[Fos99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of PLDI'99*, pp. 192–203. ACM Press, 1999.

[fra] Frama-C. `http://frama-c.cea.fr`.

[Gog77] Joseph A. Goguen, James W. Thatcher, Eric G. Wagner, and Jesse Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[Gog96] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[Hil] Mark Hills and Grigore Roşu. SILF Policy Framework. `http://fsl.cs.uiuc.edu/index.php/SILF_Policy_Framework`.

[Hil07] Mark Hills, Traian Florin Şerbănuţă, and Grigore Roşu. A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In *Proceedings of WRLA'06*, *ENTCS*, vol. 176, pp. 215–231. Elsevier, 2007.

[Hil08] Mark Hills, Feng Chen, and Grigore Roşu. A Rewriting Logic Approach to Static Checking of Units of Measurement in C. In *Proceedings of RULE'08*. Elsevier, 2008. To Appear.

[Hil09] Mark Hills. Memory Representations in Rewriting Logic Semantics Definitions. In *Proceedings of WRLA'08*, *ENTCS*, vol. 238(3), pp. 155–172. Elsevier, 2009.

[Kli09a] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-Programming with RASCAL. In *Proceedings of GTTSE'09*, pp. 185–238. Universidade do Minho, 2009.

[Kli09b] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, vol. 0, pp. 168–177. IEEE Computer Society, Los Alamitos, CA, USA, 2009.

[Mes04] J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, *LNAI*, vol. 3097, pp. 1–44. Springer, 2004.

[Mes07] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[Mey88] Bertrand Meyer. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[nis] The NIST Reference on Constants, Units, and Uncertainty. http://physics.nist.gov/cuu/Units/.

[Ros03] Grigore Rosu and Feng Chen. Certifying Measurement Unit Safety Policy. In *Proceedings of ASE'03*, pp. 304 – 309. IEEE, 2003.

[Ros07] Grigore Rosu. K: A Rewriting-Based Framework for Computations – Preliminary version. Tech. Rep. Department of Computer Science UIUCDCS-R-2007-2926, University of Illinois at Urbana-Champaign, 2007.