Generation of verification conditions for Abadi and Leino's Logic of Objects

[Extended Abstract]

Francis Tang[*]
LFCS, Division of Informatics,
University of Edinburgh,
Edinburgh EH9 3JZ,
Scotland, UK
francis.tang@dcs.ed.ac.uk

Martin Hofmann[†]
Institut für Informatik der
Ludwig-Maximilians-Universität München,
Oettingenstraße 67,
D-80538 München, Germany
mhofmann@informatik.uni-muenchen.de

ABSTRACT

We consider the problem of verification condition generation for Abadi and Leino's program logic (AL) for objects. We provide an algorithm which to a given judgement J in AL computes a formula ϕ in first-order fixpoint logic such that ϕ is equivalent to the existence of a proof of J in AL. Moreover, we show that if J is sufficiently annotated, e.g., with loop invariants, then ϕ will be purely first-order. The *verification condition* ϕ summarises the mathematical content of a correctness proof in AL while hiding all syntactic detail. We hope that in the presence of appropriate lemmas it will in many cases be possible to delegate the task of proving ϕ to a semi-automatic theorem prover so that program verification in AL would essentially amount to formulating appropriate invariants and lemmas. An object-oriented version of Euclid's algorithm looks promising in this direction.

The steps of the algorithm are as follows: (1) infer a typing derivation D of J. (2) Turn D into a skeleton proof of J which contains predicate variables in place of actual assertions. (3) Conjoin all logical side-conditions appearing in this skeleton and existentially quantify all predicate variables. The resulting second-order formula is equivalent to the existence of a proof. (4) Apply simplification rules to obtain the desired formula in fixpoint logic or perhaps in pure first-order logic.

1. INTRODUCTION AND OVERVIEW

Formal verification of OO programs still lags way behind the current technology in languages and type systems. Existing OO program logics are either non-modular [5] or incomplete [1] and in any case cumbersome to use. The present paper makes a contribution towards the latter problem: we present a method allowing one to distil the essential mathematical content of a proof by automating a large part of the syntactic overhead that comes with a formal proof in the logic AL of [1]. Though the underlying language of AL appears to be simple, it nevertheless exhibits phenomena such as aliasing, static-bound (mutual) method recursion, as well as dynamic-bound method recursion resulting from a higher-order store.

More precisely, we describe an algorithm which can automatically construct proofs in AL using an oracle for ordinary mathematical statements. For suitably annotated programs, e.g. by loop invariants, the queries to the oracle will be rather simple and as we hope provable by a semi-automatic theorem prover provided with appropriate lemmas as hints. In this way formal verification reduces to annotation and supply of lemmas. Of course, complete verification of realistic programs will remain an unassailable goal, but we believe that after further elaboration the results presented here will permit relatively painless assertion of certain properties of crucial program fragments. We remark that the modularity of the program logic AL naturally enables the independent verification of program fragments.

Supposing we are presented with a program a, a specification A (of the result of evaluating a) and a transition relation T (expressing the state change induced by evaluating a), to find a proof of $\vdash a : A :: T$ is to build a proof tree concluding in $\vdash a : A :: T$. Building the proof tree requires firstly choosing the correct proof rule at each position, and secondly, instantiating these rules with suitable choices of transition relations and specifications. Here "suitability" not only requires that the hypotheses and conclusions of the instantiated rules "fit" in the proof tree, but also the instantiated side-conditions are verifiable. We propose an approach to generate a so-called *verification condition* (VC), a logical formula expressing a sufficient condition for the existence of a proof. (Our generated VC is a necessary condition also.)

The first of these two tasks, that of choosing the correct proof rule, is made trivial by an equivalent, syntax-directed reformulation of the logic rules. Thus proofs now have a canonical shape which we consider as a *skeleton proof*. We *flesh out* the skeleton by introducing variables for the unknown specifications and transition

[*]Supported by EPSRC Studentship 98318205 and DAAD grant A/01/45666.
[†]Partially supported by the EU TYPES and APPSEM projects.

relations. The side conditions (and structure of the skeleton proof itself) give us constraints on these unknowns, including subspecification constraints on the unknown specifications. However, from type-inference, we can introduce further variables and replace the constraints on specifications with constraints on their component transition relations.

We thus have a partial proof: we know its shape, but it has unknown transition relations that must satisfy constraints, some induced by the structure, others by the side conditions. From the constraints Φ and unknowns $\vec{X}$, we obtain a second-order formula $\exists\vec{X}.\Phi$, also referred to as the *second-order verification condition*, which is logically equivalent to the existence of a proof.

We present an algorithm that finds a solution of such a constraint system, i.e. a formula ϕ equivalent to $\exists\vec{X}.\Phi(\vec{X})$ which does not contain second-order quantifiers. We refer to this formula ϕ as the *first-order, fixpoint verification condition*. It is obtained by appropriately instantiating the quantified variables in such a way that equivalence is preserved. In general, the "solution" ϕ contains least fixpoint operators.

While this is of course better than a second-order existential formula (e.g. in the case of a finite model, polynomial-time model checking algorithms would apply), the ideal would be a *first-order (fixpoint-free) verification condition*, which one hopes can be verified with an automatic theorem prover and some cleverly formulated lemmas. To that end we allow the programmer to provide annotations within a which induce further constraints on the unknown transition relations. We show that it is then possible to eliminate all fixpoint operators from the verification condition, provided enough (and correct!) annotations are provided.

At this point, one may question, why not approach the VCG problem as in previous work [8, 13, 12], and define a function by recursion over the program syntax, which gives a more local solution, as opposed to our global approach of collecting constraints and solving the resulting constraints system. However in AL, each method body is verified *once* with respect to *one* specification, which is used for all its invocations. So, for example in *let* $x{=}a$ *in* b, program b may invoke method m, whose body is defined in a, and vice-versa. Perhaps reassuringly, we know from [2] that only well-typed programs can be verified. That is, the VCG problem is at least as hard as the type inference problem. A glance at the state-of-the-art [20, 9] in type inference for object-calculi, suggests constraint systems.

Two remarks are in order: We are deliberately sloppy about the distinction between syntax and semantics of formulas and also about the precise language these could be defined in. So, notions like "first-order" are to be understood informally to mean that neither second-order quantifiers nor fixpoint operators are explicitly used. All this could be readily formalised at the expense of clarity. Second, to prevent possible misunderstanding, we emphasise that in general it is of course not possible to transform a second-order existential formula into an equivalent fixpoint, let alone first-order formula. This is possible only for the particular formulas that arise when gleaning constraints from a proof skeleton in AL.

The rest of this article proceeds as follows. After an overview of related work in Section 2, we describe AL (Section 4) and its underlying programming language (Section 3). We then describe how we can generate a second-order VC in Section 5 which we show can be simplified to a first-order fixpoint VC in Section 6. In Section 7 we introduce annotations which we show when used correctly, can eliminate all fixpoints from the first-order fixpoint VC. In Section 8, we illustrate our approach using an example.

2. RELATED WORK

Our verification condition generation approach depends upon the existence of a least shape typing. The authors have previously, in [25], developed an algorithm that infers a typing that in particular is of least shape based on [21, 20].

In [13], Homeier and Martin formally verify a VCG, for a simple while-language, notably without procedures. Subsequently, in Homeier's PhD dissertation [12], procedures are a feature of the underlying language. However, the VCG considers only *well-formed* programs, which, in particular, ban the possibility of aliased variables (Sec. 10.4.1). In contrast, correctness of our algorithm is not machine checked and our OO-language embraces both aliasing and a higher-order store.

Compaq Research's ESC project [6] also uses VCG technology. However, they compile the source language (ESC/Java and, earlier, ESC/Modula-3) into the language of guarded commands before generating the VCs. These VCs are passed to an automatic theorem prover that attempts to prove them without further interaction from the user. The motivational lineage of ESC is pragmatic. Consequently, the tool aims to catch *more* errors (than type-checking alone), but not *all* errors. Thus the tool can afford to be unsound; i.e. a program passed by the tool need not necessarily satisfy its specification. Our design goals are different in the following respects: soundness is not considered "harmful", but indeed, its availability considered crucial; secondly, the user is expected to work with the resulting VCs, and therefore it is important that they correspond in a direct way to the original program, in the same way that a C++ programmer is not expected to debug assembly code generated by the compiler.

The LOOP project [26, 14] of Nijmegen, is superficially similar to the work described here: the LOOP tool takes as input, annotated Java code (JML), compiling it to theorem prover theories (currently PVS and Isabelle), which contain theorems to be proved by the programmer. Like our approach, soundness is maintained, and furthermore, the resulting proof obligations are derived from a program logic [14] defined directly on the source language itself. In some ways, LOOP is more ambitious since it considers almost the whole of Java, including exceptions. However, we suggest that our VCG automatically discharges more proof obligations, or equivalently, our VCs are "easier" to prove than the theorems from LOOP. Specifically, in our case, invariants can be specified in the input through annotations, and no longer need to be provided when proving the VCs. In contrast, at present, though loop invariants can be specified in the JML input, the LOOP tool does not propagate this information, and thus they must be provided again whilst interacting with the theorem prover.

In response to a frequently asked question we should also point out that all the existing approaches deal with Java-style, class-based object-oriented programming (OOP) in which types and the subtyping relation are explicitly given. Abadi and Leino's logic, on the other hand, deals with delegation-based OOP where anonymous objects can be created on-the-fly, and types are inferred from the shape of objects, i.e. methods and fields present. This poses a number of additional difficulties for VC generation and type inference.

We do not attempt to advocate in this paper the delegation-based approach but simply consider it a worthwhile endeavour and actually the duty of the academic programming language research community to study both settings until such time when the superiority of one or the other approach is irrefutably and convincingly decided.

3. PROGRAMMING LANGUAGE

Abadi and Leino in [2] define a simple OO-programming language with minimal features. Objects are created directly since we do not have classes, and therefore there is no inheritance, only delegation. Nevertheless, it is an imperative language (versus functional) and together with methods in objects, we observe phenomena associated with higher-order stores, such as dynamic-bound method recursion over and above the static-bound method recursion admitted by the type system. Furthermore, the language exhibits aliasing, and unlike in other approaches, we embrace this feature rather than banning it through syntactic means or otherwise.

The syntax of programs is defined by

$$\begin{aligned} a, b ::= x \mid \textit{true} \mid \textit{false} \mid \textit{if } x \textit{ then } a_0 \textit{ else } a_1 \mid \textit{let } x{=}a \textit{ in } b \mid \\ [f_i{=}x_i{}^{i=1..k}, m_j{=}\varsigma(y_j)b_j{}^{j=1..\ell}] \mid x.f \mid x.m() \mid x.f{:=}y \end{aligned}$$

where variables a, b are understood to range over programs, x, y, z over (program) variables, f over field names and m over method names. The program

$$[f_i{=}x_i, m_j{=}\varsigma(y_j)b_j]$$

creates an object with fields f_i initially assigned with the values of x_i and methods m_j. The ς symbol binds variable y_j to *self* in each method body b_j. Methods with several parameters can be encoded by initially assigning to fields and then using the fields within the method body. We write $x.f$ to look up the value of a field, $x.m()$ to invoke a method and $x.f{:=}y$ to overwrite the value of field f in x with the value of y. Other than by the ς-binder, program variables are introduced by the let construction: program *let* $x{=}a$ *in* b executes a, identifying x with its return value before executing b.

There are some base types with appropriate constants such as *Bool* and object types of the form $[f_i{:}A_i{}^{i=1..k}, m_j{:}B_j{}^{j=1..\ell}]$ which contain objects providing fields f_i and methods m_i.

There are no *recursive types*, but we believe that the type inference mechanism described in [25] as well as the verification condition generation presented here an in [24] can be readily extended to encompass them. In support of this belief we note that Palsberg's algorithms on which [25] is based work for recursive types and that the apparatus for verification condition generation we present here is to a considerable extent generic.

If variable x has type $[f_i{:}A_i, m_j{:}B_j]$ then $x.f_i$ has type A_i and B_j is the return type of method m_j. There is a structural subtyping judgment $A <: B$ defined in the usual way with the notable feature that object types are covariant in the methods and invariant in the fields. As an auxiliary notion we say that a type A has smaller shape than A' if hereditarily all fields (and methods) of A are fields (and methods) of A'.

4. A LOGIC OF OBJECTS

In AL, types are generalised to specifications of the form

$$[f_i{:}A_i{}^{i=1..k}, m_j{:}\varsigma(y_j)B_j{::}U_j{}^{j=1..\ell}]$$

where A, B and their variants now range over specifications and T, U and their variants range over transition relations. Note that B, U can also depend on y_j.

Semantically, we have values which are either elements of base types or store locations. A store is a partial function mapping location-field name pairs to values. A transition relation of arity n semantically is a relation over initial and final stores, a *return value* and n values. Semantic entailment of transition relations is precisely set inclusion of their graphs. Note that variables are mapped to values by stacks, as elaborated on in [2].

Syntactically, transition relations are, up to α-equivalence, pairs $(\vec{x}, T)$ where $\vec{x}$ is a sequence of (program) variables, and T is logical formula over the variables $\vec{x}$, initial and final stores $\grave{\sigma}$ and $\acute{\sigma}$, and the return value r. In this paper we use a logic with predicate symbols including Res, T_{fsel}, T_{fupd} and T_{obj}, whose standard interpretations are as one would expect from their context, but their precise meanings (and the precise grammar of formulas) are not important for our considerations here. The *arity* of $(\vec{x}, T)$ is defined to be the length of $\vec{x}$. We take a natural ordering of "strength" between transition relations, by saying that $(\vec{x}, T)$ is stronger than $(\vec{x}, T')$ written $(\vec{x}, T) \subseteq (\vec{x}, T')$ precisely when T entails T'. Note that the strength relation is only defined between transition relations of the same arity, but transition relations of different arities can be related by a (sometimes implicit) weakening. Similarly, when there is no ambiguity, we often write simply the logical formula with free variable occurrences, leaving the first component implicit.

Subtyping extends to specifications in the obvious way using entailment of transition relations.

In AL, we derive judgments of the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash a : A :: T \;\;,$$

where what appears to the left of $\vdash$ is referred to as the context. One should think of T, a *dynamic specification*, as describing the execution (or dynamic) behaviour of a, and A, a *static specification*, as describing the properties of the resulting value (assuming a terminates). Here each x_i is a free variable that can occur in a, A and T, as well as in $A_{i'}$ ($i < i'$). Furthermore, we can assume that the value of x_i has specification A_i. Thus we have enough information to deduce the specification and execution behaviour of $x.m()$ simply by looking up x in the context and opening up its specification. This higher-order compositional property of the logic allows us to reason about method invocation without the need to look up the text of the method body. In contrast, program logics of de Boer [4, 5] and von Oheimb [18, 19] use the fact that the texts of method bodies are available.

Table 1 gives the rules of AL. It differs from the version in [1] in that subsumption is built into the rules thus simplifying constraint generation. One can show that this formulation is equivalent to the one from loc. cit.

5. SECOND-ORDER VERIFICATION CONDITIONS

Let us introduce some notation to allow us to define the constraint system induced by a program. For a subterm occurrence a and types $A, A_1, \ldots, A_n$, suppose the judgement

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A$$

$$\frac{}{E \vdash x_j : A :: T} \quad \left\{ \begin{array}{l} E(x_j) <: A \\ \mathit{Res}(x_j) \subseteq T \end{array} \right\}$$

$$\frac{}{E \vdash \mathit{true} : \mathit{Bool} :: T} \quad \left\{ \mathit{Res}(\mathit{true}) \subseteq T \right\}$$

$$\frac{}{E \vdash \mathit{false} : \mathit{Bool} :: T} \quad \left\{ \mathit{Res}(\mathit{false}) \subseteq T \right\}$$

$$\frac{}{E \vdash n : \mathit{Nat} :: T} \quad \left\{ \mathit{Res}(n) \subseteq T \right\}$$

$$\frac{E \vdash x : \mathit{Bool} :: \mathit{Res}(x) \quad E \vdash a_0 : A_0 :: T_0 \quad E \vdash a_1 : A_1 :: T_1}{E \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : A' :: T'} \quad \left\{ \begin{array}{l} T_0[\mathit{true}/x] \subseteq T[\mathit{true}/x] \\ T_1[\mathit{false}/x] \subseteq T[\mathit{false}/x] \\ A_0[\mathit{true}/x] <: A[\mathit{true}/x] \\ A_1[\mathit{false}/x] <: A[\mathit{false}/x] \end{array} \right\}$$

$$\frac{E \vdash a : A :: T \quad E, x : A \vdash b : B :: U}{E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : A' :: T'} \quad \left\{ \begin{array}{l} T ;_x U \subseteq T' \\ B <: A' \end{array} \right\}$$

for $A = [f_i{:}{A_i}^{i=1..k}, m_j{:}\varsigma(y_j)B_j{::}{U_j}^{j=1..\ell}]$

$$\frac{E \vdash x_i : A_i :: \mathit{Res}(x_i)^{1 \le i \le k} \quad E, y_j{:}A \vdash b_j : B_j :: {U_j}^{1 \le j \le \ell}}{E \vdash [f_i{=}{x_i}^{i=1..k}, m_j{=}\varsigma(y_j){b_j}^{j=1..\ell}] : A' :: T'} \quad \left\{ \begin{array}{l} A <: A' \\ T_{\mathsf{obj}}(x_1, \dots, x_k) \subseteq T' \end{array} \right\}$$

$$\frac{E \vdash x : [f{:}A] :: \mathit{Res}(x)}{E \vdash x.f : A' :: T'} \quad \left\{ \begin{array}{l} A <: A' \\ T_{\mathsf{fsel}}(x, f) \subseteq T' \end{array} \right\}$$

$$\frac{E \vdash x : [m{:}\varsigma(y)B{::}U] :: \mathit{Res}(x)}{E \vdash x.m() : A :: T} \quad \left\{ \begin{array}{l} B[x/y] <: A \\ U[x/y] \subseteq T \end{array} \right\}$$

$$\frac{E \vdash x_j : A :: \mathit{Res}(x_j) \quad E \vdash x_k : A'' :: \mathit{Res}(x_k)}{E \vdash x_j.f{:=}x_k : A' :: T'} \quad \left\{ \begin{array}{l} A <: [f{:}A''] \\ A <: A' \\ T_{\mathsf{fupd}}(x_j, f, x_k) \subseteq T' \end{array} \right\}$$

Table 1: Rules for Abadi-Leino program logic. In this presentation, the subsumption rule has been incorporated into each rule, thus the rules are syntax-directed.

occurs in the inferred least shape type derivation. We write $\llbracket a \rrbracket$ for A, and for any variable x_i, we write $[x_i]$ for A_i. Note that although the variable x_i can appear in the contexts of many judgements, $[x_i]$ is still well-defined since x_i it must have the same type in the contexts of all judgements in the same proof. For γ a sequence of field names and method names, and A a type, we overload notation and write $\gamma \in A$ to mean that γ is a path in the type A. We implicitly assume that the symbol γ (and its decorated variants) ranges over sequences ending in a method name. Similarly, α ranges over non-empty sequences of method names, and β ranges over sequences ending in a method name but containing at least one field name. We write $\#\gamma$ for the number of method names in γ.

A variable renaming is an injective, order-preserving function $\sigma : \{1..m\} \to \{1..n\}$ between initial segments of $\mathbb{N}$. Let $\mathsf{dom}(\sigma)$ (domain) denote $\{1..m\}$ and $\mathsf{cod}(\sigma)$ (codomain) denote $\{1..n\}$. Its application to a transition relation $T = T(x_1, \dots, x_m)$ of arity m is written σT and is defined pointwise by

$$(\sigma T)(x_1, \dots, x_n) \stackrel{\text{def}}{=} T(x_{\sigma(1)}, \dots, x_{\sigma(m)}) \ .$$

Recall from the discussion in the overview, we take the inferred types as skeleton specifications, and flesh them out by introducing variables for all unknown transition relations. Note for any subterm occurrence a, there is one transition relation for each $\gamma \in \llbracket a \rrbracket$. Thus for each judgement $E \vdash a : A :: T$ with unknown A and T, we introduce a variable $\llparenthesis a \rrparenthesis$ for T and for each $\gamma \in \llbracket a \rrbracket$, a variable $\llparenthesis a \rrparenthesis_\gamma$ for the component transition relations of A.

The arity of $\llparenthesis a \rrparenthesis$, written $\mathrm{ar}(\llparenthesis a \rrparenthesis)$ is n, where

$$x_1{:}[x_1], \dots, x_n{:}[x_n] \vdash a : \llbracket a \rrbracket$$

is the derived typing judgement of a. Now consider $\llparenthesis [m{=}\varsigma(y)b] \rrparenthesis_m$, which we recall is the transition relation for method m, whose body is b. Since y occurs free in b, its arity is one more than that of $\llparenthesis [m{=}\varsigma(y)b] \rrparenthesis$. Thus the arity of $\llparenthesis a \rrparenthesis_\gamma$ is $\mathrm{ar}(\llparenthesis a \rrparenthesis) + \#\gamma$.

For each variable x and $\gamma \in [x]$, we introduce a second-order variable $\langle x \rangle_\gamma$ to denote the component transition relation of the specification of x that occurs in the verification contexts. If x_j occurs in the context of verification judgement

$$x_1{:}A_1, \dots, x_j{:}A_j, \dots, x_n{:}A_n \vdash a : A :: T$$

then the arity of $\langle x_j \rangle_\gamma$ is $j - 1 + \#\gamma$. This is well-defined since a program variable must occur in the same position and have the same specification in all contexts; we cannot permute variables in contexts.

$$\begin{array}{rl} w_p^{q,\gamma} & : \{1..(p + \#\gamma)\} \to \{1..(q + \#\gamma)\} \\ w_p^{q,\gamma}(i) & \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} i & i \le p \\ i + (q - p) & i > p \end{array} \right. \\ \sigma_p^{q,\gamma} & : \{1..(p + 1 + \#\gamma)\} \to \{1..(q + \#\gamma)\} \\ \sigma_p^{q,\gamma}(i) & \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} i & i \le p + 1 \\ i + (q - p - 1) & i > p + 1 \end{array} \right. \end{array}$$

Table 2: Some basic renamings

Table 2 defines some useful renaming functions. The renaming $w_p^{q,\gamma}$ is a weakening/renaming used for variable instances. The idea is to map the first p variables into the first q slots, and then map the remaining $\#\gamma$ variables (which are introduced by the ς-binders) to the $\#\gamma$ slots after the first q slots. Similarly, the renaming $\sigma_p^{q,\gamma}$ is also a weakening, and differs from $w_p^{q,\gamma}$ only in the use of its formal parameters. It is used to shuffle the variables into the correct slots for method invocation.

Table 4 contains the definition of function Cons which maps a vector of variables $\vec{x}$ and a term a involving $\vec{x}$ to a set of logical formulas involving the previously introduced predicate variables for a. The idea is that $\mathrm{Cons}(\vec{x}, a)$ summarises the constraints arising when trying to construct a proof in AL with subject a.

A detailed discussion of the defining clauses is contained in [24] but must be elided here for space reasons.

We have the following formulation of completeness of Cons. In [24] we show why the restriction on shape does not in fact constitute a loss of generality.

THEOREM 1 (COMPLETENESS). *Given a program a, specification A and transition relation T. If $\vdash a : A :: T$ has a proof whose component specifications have the same shape as their corresponding types in the inferred least shape typing, then*

$$\mathit{Cons}(\varepsilon, a) \cup \{(\!|a|\!)_\gamma = A(\gamma) \mid \gamma \in A\} \cup \{(\!|a|\!) = T\} \qquad (1)$$

has a solution.

Similarly, we have the converse, soundness: given a, A, T, if constraints (1) has a solution, then there is a proof of $\vdash a : A :: T$.

6. FIRST-ORDER, FIXPOINT VERIFICATION CONDITIONS

We know from the previous theorem, that $\mathrm{Cons}(\varepsilon, a)$ is a VC, albeit expressed as a second-order formula. However, this VC offers us no benefit over finding a proof directly, since to prove this formula, one must find instantiations for the existentially quantified transition relations. Fortunately, we can simplify this VC by automatically finding instantiations that preserve logical equivalence. Indeed, by applying them in the right order, the rules in Tables 10, 7, 8 and 9 allow one to reduce any formula of the form

$$\exists \vec{X}. \quad \bigwedge_{e,e'} e = e' \quad \wedge \quad \bigwedge_i L_i \subseteq X_i \ ,$$

(such is the form of the formulas produced by Cons) to an equivalent formula containing no second-order existential quantifiers, but least fixpoint operators (which we may view as a particular case of second-order quantification by the theorem of Knaster-Tarski) in their stead.

Again, for lack of space we are not able to discuss these rules in any great detail but only mention that they substantially and non-trivially extend the work of Bledsoe [3]. In particular, he only considers the instantiation of a single second-order variable and does not achieve completeness, i.e., the instantiated formula may be false in spite of the original one being provable. Of course, the present improvements are partly enabled by the particular syntactic form of our constraints.

7. FIRST-ORDER VERIFICATION CONDITIONS

While the possibility of eliminating all second-order quantification in favour of fixpoints came as a surprise to us, the presence of fixpoints in a verification condition seems quite natural. After all, when verifying a program with looping behaviour, be it from a while construct, or recursive method invocation, if one wants a purely first-order verification condition, then one would expect to be required to provide an invariant of some sort. This last analogy, suggests that if we allow the programmer to provide further hints to the verification condition generator, in the form of annotations in the program, then we might be able to eliminate some of the fixpoint operators.

Closer examination of the simplification rules reveals that fixpoint operators are only introduced by Rule (freeinst), that is, whenever we find a constraint of the form $L \subseteq X$ and X occurs free in L, and we instantiate for X. Supposing the programmer can provide an explicit instantiation of X to the simplification process as a hint, then at least this fixpoint operator would be eliminated.

Taking this as our motivation, we extend the syntax of the programming language by allowing the user to provide annotations. For our purposes, an annotation ψ is a partial function mapping sequences γ (including ε in this case) to transition relations. The precise syntax used to write such partial functions is not important. Annotated programs, which for the sake of brevity will also be denoted by the symbols a and b, are defined as before except we now allow annotations $a{::}\psi$. Furthermore, annotations of variables can still be used as variables, so for example, we can write $x.f{:=}(y{::}\psi)$.

$$\begin{array}{rcl}
\mathrm{Cons}(\vec{x}, a{::}\psi) & \overset{\text{def}}{=} & \mathrm{Cons}(\vec{x}, a) \\
& \cup & \{(\!|a|\!)_\gamma = \psi(\gamma) \mid \gamma \in [\![a]\!] \cap \mathsf{dom}(\psi)\} \\
& \cup & \{(\!|a|\!) = \psi(\varepsilon) \mid \varepsilon \in \mathsf{dom}(\psi)\} \\
& \cup & \{\psi(\alpha) \subseteq (\!|a{::}\psi|\!)_\alpha \mid \alpha \in [\![a]\!] \cap \mathsf{dom}(\psi)\} \\
& \cup & \{\psi(\beta) = (\!|a{::}\psi|\!)_\beta \mid \beta \in [\![a]\!] \cap \mathsf{dom}(\psi)\} \\
& \cup & \{\psi(\varepsilon) \subseteq (\!|a{::}\psi|\!) \mid \varepsilon \in \mathsf{dom}(\psi)\} \\
& \cup & \{(\!|a|\!)_\gamma = (\!|a{::}\psi|\!)_\gamma \mid \gamma \notin \mathsf{dom}(\psi)\} \\
& \cup & \{(\!|a|\!) = (\!|a{::}\psi|\!) \mid \varepsilon \notin \mathsf{dom}(\psi)\}
\end{array}$$

Table 3: Constraints generation for a program a annotated with ψ. Annotations are partial functions mapping paths to transition relations.

We now add the extra clause as displayed in Table 3 to the definition of Cons. Here we have extended $[\![-]\!]$ to annotated programs. Note that we use the information in ψ whenever it is defined, and simply identify $(\!|a|\!)_\gamma$ and $(\!|a{::}\psi|\!)_\gamma$ whenever it is not. We ignore $\psi(\gamma)$ for $\gamma \notin [\![a]\!]$.

7.1 How Many Annotations?

Of course, if we annotate every single subterm then the verification condition will not contain any fixpoints. On the other hand, it is natural that recursive methods will require an annotation—the invariant. Perhaps unexpectedly, higher-order assignment, i.e., an update of a field of non-trivial object type also may produce fixpoints in verification conditions. Fortunately, this is not entirely an artifact of our approach as such assignment allows one to encode recursion implicitly. Consider the following example

$$\begin{array}{l}
\textit{let } u{=}[f{=}[m{=}\varsigma(y)\textit{true}]] \textit{ in} \\
\quad u.f{:=}[m{=}\varsigma(y)u.f.m()];\ u.f.m() \ .
\end{array}$$

Here we initially create an object, which we call u, with one field f initialised to an object with one method m that computes something trivial. Thereafter, within the main body of the let construct, we update field $u.f$ with another object with method m which invokes method $u.f.m$. When run, the program exhibits non-termination. and in a similar vein it is possible to write programs computing factorial, Fibonacci numbers, etc, recursively.

One can show that if all *recursive* methods and *higher-order* assignments are annotated, no fixpoints will appear in the verification condition. Note that this does not require us to annotate $x.f{:=}y$ in the case where y is not higher-order, i.e. contains no methods. On the other hand, insisting on all higher-order assignments to be annotated would be unnecessarily verbose as there are many examples of harmless such update.

Lacking at present an acceptable yet simple syntactic criterion, we exhibit in the technical report [24] a notion of dependency graph which is associated with each program and has the property that fixpoints will not arise if the dependency graph contains no cycles. We only point out here that due to the presence of equality constraints via conditionals these dependency graphs are nontrivial and, in particular, must be parametrised by truth assignments of the boolean subexpressions. The rationale of these graphs is two fold. On the one hand, they should enable the future development and validation of complete yet not unnecessarily restrictive annotation policies. On the other hand, they may be used in an interactive program verification system which displays all the cycles in a dependency graph asking the programmer for a breaking annotation in each case.

8. AN EXAMPLE

In this section we develop a simple example: an object-oriented version of Euclid's algorithm. We are aware of the simplicity and perhaps naivity of this example and intend to develop more meaningful examples in due course. However, though our prototype implementation can already generate VCs for a larger example, further implementation efforts are required to make the result suitable for human consumption. Another point is that many possibly interesting examples require recursive types which as discussed in Section 3 can presumably be quite easily added, but have not yet been checked in detail.

On the other hand, we felt that delaying publication of our work until such time when more substantial examples would become available would be detrimental to the dissemination and possible impact of this work.

Recall the greatest common divisor (gcd) example as found in [2] and mechanically checked in [11]. The example is resurrected in the following form.

$$
\begin{array}{lll}
\mathrm{gcd} \stackrel{\text{def}}{=} & [& f{=}1, g{=}1, \\
& & m{=}\varsigma(y)\ \ \textit{if}\ y.f < y.g\ \textit{then} \\
& & \qquad\qquad y.g{:=}y.g - y.f;\ y.m() \\
& & \qquad \textit{else if}\ y.g < y.f\ \textit{then} \\
& & \qquad\qquad y.f{:=}y.f - y.g;\ y.m() \\
& & \qquad \textit{else}\ y.f \\
& & \qquad ::I \qquad\qquad\qquad\qquad\qquad]
\end{array}
$$

The program gcd is the same program as seen previously except we now annotate the method body with an invariant I. Note that since the type of the body of method m is integer, its annotation, which in general is a partial function mapping paths γ to transition relations, degenerates to simply a transition relation.

Program gcd computes a higher-order result: a gcd calculator. To illustrate its intended use, we wrap program gcd into program a, with its specification provided by an annotation:

$$
\begin{array}{ll}
a \stackrel{\text{def}}{=} & \textit{let}\ x{=}\mathrm{gcd}\ \ \textit{in}\ ((x.f{:=}n_1).g{:=}n_2).m() \\
& ::\mathrm{pos}(n_1), \mathrm{pos}(n_2) \to r = \mathrm{gcd}(n_1, n_2)\ \ .
\end{array}
$$

The annotated program a insists that if we update the fields f and g with positive constants n_1 and n_2, then invoking method m computes the gcd of n_1 and n_2. Here the predicate symbol pos is interpreted as "is positive".

Now applying type inference to a, we infer the following type for gcd.

$$\vdash \mathrm{gcd} : [f{:}\textit{Nat}, g{:}\textit{Nat}, m(){:}\textit{Nat}]\ \ .$$

Using our implementation, computing $\mathrm{Cons}(\varepsilon, a)$ gives a second-order VC Ψ_1, with 60 (existentially quantified) higher-order variables, 6 equality constraints and 50 inequality constraints. We can eliminate all higher-order variables by applying our simplification rules to obtain a VC Ψ_2.

Using specific facts about AL transition relations, for example

$$(\textit{Res}(n);_x U(x)) = U(n)\ \ ,$$

we can simplify Ψ_2 further, to obtain the following more readable formulas. Here we write the VC as an (implicit) conjunction of first-order formulas with $\grave{\sigma}$ and $\acute{\sigma}$ denoting the initial and final stores and r for the result value.

$$
\begin{array}{c}
\left(\begin{array}{l} \grave{\sigma}(y,f) \not< \grave{\sigma}(y,g) \wedge \grave{\sigma}(y,g) \not< \grave{\sigma}(y,f) \wedge \\ T_{\text{fsel}}(y,f) \end{array}\right) \quad \subseteq \quad I \\
\left(\begin{array}{l} \grave{\sigma}(y,f) \not< \grave{\sigma}(y,g) \wedge \grave{\sigma}(y,g) < \grave{\sigma}(y,f) \wedge \\ T_{\text{fupd}}(y,f,\grave{\sigma}(y,f) - \grave{\sigma}(y,g)); I \end{array}\right) \quad \subseteq \quad I \\
\left(\begin{array}{l} \grave{\sigma}(y,f) < \grave{\sigma}(y,g) \wedge \\ T_{\text{fupd}}(y,g,\grave{\sigma}(y,g) - \grave{\sigma}(y,f)); I \end{array}\right) \quad \subseteq \quad I \\
T_{\text{obj}}(1,1);_x T_{\text{fupd}}(x,f,n_1); T_{\text{fupd}}(x,g,n_1); I[x/y] \subseteq (\!|a|\!)
\end{array}
$$

where

$$(\!|a|\!) \equiv \mathrm{pos}(n_1), \mathrm{pos}(n_2) \to r = \mathrm{gcd}(n_1,n_2)\ \ .$$

Recall that $T \subseteq T'$ is an abbreviation for

$$\forall \vec{x}, \grave{\sigma}, \acute{\sigma}, r. T(\vec{x}, \grave{\sigma}, \acute{\sigma}, r) \to T'(\vec{x}, \grave{\sigma}, \acute{\sigma}, r)\ \ .$$

In this case, Ψ_2 is fixpoint-free and so our implementation can pretty-print to a first-order syntax. When we define I by

$$
I \stackrel{\text{def}}{=} \begin{array}{l} \mathrm{pos}(\grave{\sigma}(y,f)) \wedge \\ \mathrm{pos}(\grave{\sigma}(y,g)) \end{array} \quad \to \quad \begin{array}{l} r = \acute{\sigma}(y,f) \wedge r = \acute{\sigma}(y,g) \wedge \\ r = \mathrm{gcd}(\grave{\sigma}(y,f), \grave{\sigma}(y,g))\ \ , \end{array}
$$

SPASS [27], an automated theorem prover for first-order logic with equality, can successfully prove Ψ_2, if we add the following axioms about gcd and subtraction,

$$\forall_{\mathbb{Z}} x, y. \quad x < y \rightarrow \mathrm{pos}(y - x) \tag{2}$$

$$\forall_{\mathbb{Z}} x, y. \quad x \not< y, y \not< x \rightarrow x = y \tag{3}$$

$$\forall_{\mathbb{Z}} x, y. \quad x < y, \mathrm{pos}(x), \mathrm{pos}(y) \rightarrow \mathrm{gcd}(x, y) = \mathrm{gcd}(x, y - x) \tag{4}$$

$$\forall_{\mathbb{Z}} x, y. \quad y < x, \mathrm{pos}(x), \mathrm{pos}(y) \rightarrow \mathrm{gcd}(x, y) = \mathrm{gcd}(x - y, y) \tag{5}$$

$$\forall_{\mathbb{Z}} x. \quad \mathrm{gcd}(x, x) = x \tag{6}$$

$$f \neq g \ . \tag{7}$$

Axiom 7 can be generated by our implementation. SPASS required about 15 seconds to find a proof of Ψ_2, on a 500MHz Pentium III.

As a demonstration of the robustness of this VCG approach, consider a modification of the algorithm where the second branch of the nested if-then-else statement performs an (inline) swap, before performing a recursive method call:

$$\textit{let } z{=}y.f \textit{ in } (y.f{:=}y.g;\ y.g{:=}z;\ y.m()) \ .$$

The generated VC can be automatically discharged by SPASS if we also add the axiom

$$\forall_{\mathbb{Z}} x, y. \quad \mathrm{gcd}(x, y) = \mathrm{gcd}(y, x) \ . \tag{8}$$

Alternatively, if we modify the second branch so that the swap is performed by invoking a sibling method, viz.

$$y.swap();\ y.m() \ ,$$

(and we add a new sibling method $swap$) the resulting VC can also be automatically discharged by SPASS. For reference, the version with the inline swap (resp. swap method) produced a 2nd-order VC with 64 (resp. 70) higher-order variables, 6 (resp. 9) equality constraints and 54 (resp. 55) inequality constraints.

Clearly, the two versions with swapping are implementations of the same algorithm. The fact that the generated VC can be automatically proved using same the axioms is particularly reassuring since this is a demonstration of how our VCG is indifferent to their syntactic distinctions.

9. CONCLUSION AND FURTHER WORK

We have presented a method enabling to separate the syntactic overhead of the AL logic from the actual mathematical content of verification. A first example involving the recursive gcd object from [11] was very promising. While a formalisation [11] in LEGO [23] of the complete proof in AL took three weeks with a resulting proof script of several pages, the generated verification condition was just one line and amounted to the intuitive proof obligation that would arise from a simple imperative program. All the overhead due to objects, methods, etc. has been taken care of.

So far we have a prototype implementation which can generate first-order, fixpoint-free VCs for the example in Section 8 and also for an example based on the dining philosophers scenario [11]. Unfortunately using the latest version of SPASS, the VC from the latter example trips up on a bug, which the authors have been assured will be fixed in a future release. In the meantime, modifications to the implementation are being considered to allow pretty printing to different automated theorem provers, including OTTER [16].

For the future we intend to develop an implementation which could display loops in dependency graphs and so ask the user for more annotations when needed. Alternatively, one could strive for a static type system capable of guaranteeing cycle freeness of dependency graphs. Our experience so far, in particular the presence of implicit recursion described in Section 7.1 leads us to a pessimistic view as to the existence of such a system which would at the same time be reasonably strong and comprehensible. However, recent work on alias analysis and linearity [28, 17, 10] might be useful.

One of the goals of our VCG was to provide increased automation for finding verification proofs. The main bulk of the presentation in [24] is to show that our VCG algorithm is correct. An alternative approach towards the same goal is to embed the program logic into a theorem prover, for example in [11], so that the theorem prover can be used as an assistant for building proofs. Then one can use the automation provided by the theorem prover, for example by implementing tactics [8], to reduce some of the resulting proof burden. In particular, unsound tactics cannot allow incorrect "proofs" to be derived, since the proof checking facility of the theorem prover guards us against such an eventuality. Thus *soundness* of the VCG is then of reduced importance. Nevertheless, we believe that completeness of the VCG algorithm is still important: in its absence, should one obtain an unprovable subgoal from a tactic, one cannot conclude that the goal itself is unprovable. If one chooses carefully how AL is embedded into a suitable theorem prover such as Isabelle/HOL, we believe that the VCG algorithm presented here can be implemented as a tactic.

Our method is currently restricted to closed programs. However, due to the compositional nature of AL, and the constraints-solving approach to VCG, we foresee no difficulty to extend it to open programs which would allow us to apply verification to certain crucial parts of a program and simply assume correctness of others.

Extensions of the AL logic either by recursive types or with a view to address its incompleteness will lead to new challenges. As already mentioned, an extension with recursive types appears reachable.

As already mentioned, in a finite model fixpoints can be computed quickly (polynomial in the size of the model) by iteration. Thus, if we were able to detect statically, whether the required model was indeed finite then we could use model-checking, i.e., iterative computation of fixpoints to discharge verification conditions automatically. We believe that the model, i.e., the state space is indeed finite in examples like the dining philosophers scenario described in [11].

From a point of view of programming methodology, we may use unsound but complete approximations to proving the VC, and thus detect possible mistakes in the code and/or specification. If these complete methods are "push-button" (automatic) then, for example, they may be applied off-line, e.g. overnight, during the development of a project. For example, we could use model-checking on an arbitrarily sized, finite model and in this way possibly detect unsatisfiability of the verification conditions. Also, we can eliminate fixpoint expressions by replacing a constraint $L \subseteq Y$, where L contains a fixpoint, with finitely many approximations: for example, supposing $L \equiv L'[\mu X.F(X)/Z]$ for some L', the approximations can be

$$L'[F^i(\bot)/Z] \subseteq Y$$

for $i = 0..n$. To see why this is complete, we recall that fixpoints only arise from applications of Rule (notfreeinst) and so $Z \mapsto L'$

is monotone. Certainly $F^i(\bot) \subseteq \mu X.F(X)$, and so

$$L'[F^i(\bot)/Z] \subseteq L'[\mu X.F(X)/X] \ .$$

Therefore $L'[F^i(\bot)/Z] \subseteq Y$ is a consequence of $L \subseteq Y$. Since these approximations are fixpoint-free, they are more likely to be automatically discharged, or disproved, again giving a push-button check. It is noted that ESC uses approximations of VCs for while loops.

Flanagan and Saxe address the problem of exponential blow up of VCs in [7]. It is clear that our VCG can experience the same complexity issues and it would be interesting to see if modifications of their solutions can be applied to ours.

Our global approach to verification condition generation appears to be to quite some degree independent of the program logic at hand though, of course, for simpler logics such as plain Hoare logic the well-known straightforward computation of the verification condition by structural recursion works just as well. It would be interesting to apply our approach to other logics, for example Poetzsch-Heffter and Müller's programming logic [22] for a fragment of Java, and also Leino's later variant [15] of AL, which admits recursively typed objects.

In the later logic of Leino, preliminary investigations suggest that since subtyping is defined by *name matching* (like in Java, as opposed to *structural type matching* as present in AL), the VCG problem is easier. Firstly, type inference is much simpler, and also, since each method has exactly one specification for all implementations (thus specifications are no longer covariant along methods), far fewer constraints (and thus second-order variables) are generated. Of course, these advantages are at the expense of a more restrictive type system, though it appears to be sufficient for modelling Java.

A thorough comparison between AL and the work of Poetzsch-Heffter and Müller remains to be carried out; preliminary inspection suggests that their subtyping relation derives from Java, and so, like Leino's later logic, may give a simpler VCG. However, initial impressions suggest that their logic has a syntax significant larger than what we have considered so far. Thus, it maybe be beneficial to implement the logic in a theorem prover, and its VCG as a tactic, to ensure soundness.

Acknowledgments
The authors would like to thank Lennart Beringer and various anonymous referees for comments on, earlier versions of this paper.

10. REFERENCES

[1] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, April 1997.

[2] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. SRC Research Reports SRC-161, Compaq SRC, September 1998. Revised version of [1].

[3] Woodrow W. Bledsoe. A maximal method for set variables. *Machine Intelligence*, 9:53–100, 1979.

[4] Frank S. de Boer. *Reasoning about dynamically evolving process structures; a proof theory for the parallel object-oriented language POOL*. PhD thesis, The Free University of Amsterdam, 1991.

[5] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structures, FoSSaCS '99*, volume 1578 of *LNCS*, pages 135–149. Springer-Verlag, 1999.

[6] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Reports SRC-159, Compaq SRC, December 1998.

[7] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *28th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 2001.

[8] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[9] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In *FOOL4: 4th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 1997.

[10] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[11] Martin Hofmann and Francis Tang. Implementing a program logic of objects in a higher-order logic theorem prover. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *LNCS*, pages 267–282. Springer-Verlag, 2000.

[12] Peter V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. PhD thesis, University of California, Los Angeles, 1995.

[13] Peter V. Homeier and David F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In T. F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *LNCS*, pages 269–284. Springer-Verlag, 1994.

[14] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussman, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

[15] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing*, 5(4):330–360, Winter 1998.

[16] William W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.

[17] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 2000.

[18] David von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 168–180. Springer-Verlag, 1999.

[19] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[20] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[21] Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[22] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[23] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[24] Francis Tang and Martin Hofmann. Reducing proof burden in object-oriented verification. Technical report, University of Edinburgh, 2001. To appear, see `www.dcs.ed.ac.uk/home/fhlt` for an online version.

[25] Francis Tang and Martin Hofmann. Type inference for objects with base types. Technical report, University of Edinburgh, 2001. To appear, see `www.dcs.ed.ac.uk/home/fhlt` for an online version.

[26] Joachim van den Berg and Bart Jacobs. The loop compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag, 2001.

[27] Christoph Weidenbach et al. System description: SPASS version 1.0.0. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 378–382, Trento, Italy, July 7–10, 1999. Springer-Verlag.

[28] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *9th International Conference on Compiler Construction*, number 1781 in LNCS, pages 1–16. Springer-Verlag, 2000.

APPENDIX

$$\begin{tikzcd}
\exists\vec{X}.\ \bigwedge_{e,e'} e = e' \ \wedge\ \bigwedge_i L_i \subseteq X_i \arrow[d, "\text{embedding}"] \\
\exists\vec{X}.\ \bigwedge_i v_i \Rightarrow e_{0,i} = e_{1,i} \ \wedge\ \bigwedge_i v'_i, L_i \subseteq e'_i \ \wedge\ \Psi_0 \arrow[d, "\text{eliminate equality constraints}"] \\
\exists\vec{X}.\ \bigwedge_i v'_i, L_i \subseteq e'_i \ \wedge\ \Psi_0 \arrow[d, "\text{simplification of } - \lhd - \rhd -"] \\
\exists\vec{X}.\ \bigwedge_i T_i \subseteq e'_i \ \wedge\ \Psi_0 \arrow[d, "\text{adjoints}"] \\
\exists\vec{X}.\ \bigwedge_i T_i \subseteq X_i \ \wedge\ \Psi_0 \arrow[d, "\text{minimal set instantiations}"] \\
\text{First-order formula } \Psi_0
\end{tikzcd}$$

Table 5: Schematic representation of reduction algorithm. In the first step, we choose ε for each v_i, v'_i to transform the generated VC in to the form suitable our simplification rules. We first apply the equality elimination rules in Table 10, until they have all been eliminated. Then we eliminate all $- \lhd - \rhd -$ expressions using the rules in Table 7. Then we apply the rules in Table 8 to transform the VC into a form suitable for the rules in Table 9.

$$\frac{}{\sigma(\sigma' e) \rightsquigarrow (\sigma \circ \sigma')e} \quad \text{(rw-renren)}$$

$$\frac{}{\sigma(e_0 \lhd x \rhd e_1) \rightsquigarrow (\sigma e_0) \lhd x \rhd (\sigma e_1)} \quad \text{(rw-renif)}$$

Table 6: Expression rewriting. These rules allow us to pull renamings inside of $- \lhd - \rhd -$ expressions.

Writing n for $|\vec{x}|$,

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, x_j.m()) \overset{\text{def}}{=}\ & \{\sigma^{n,\alpha}_{j-1}\langle x_j\rangle_{m\alpha} \subseteq \llparenthesis x_j.m()\rrparenthesis_\alpha \mid \alpha \in \llbracket x_j.m()\rrbracket\} \\
\cup\ & \{\sigma^{n,\beta}_{j-1}\langle x_j\rangle_{m\beta} = \llparenthesis x_j.m()\rrparenthesis_\beta \mid \beta \in \llbracket x_j.m()\rrbracket\} \\
\cup\ & \{\sigma^{n}_{j-1}\langle x_j\rangle_{m} \subseteq \llparenthesis x_j.m()\rrparenthesis\} \\
\mathrm{Cons}(\vec{x}, x_j) \overset{\text{def}}{=}\ & \{w^{n,\alpha}_{j-1}\langle x_j\rangle_{\alpha} \subseteq \llparenthesis x_j\rrparenthesis_\alpha \mid \alpha \in \llbracket x_j\rrbracket\} \\
\cup\ & \{w^{n,\beta}_{j-1}\langle x_j\rangle_{\beta} = \llparenthesis x_j\rrparenthesis_\beta \mid \beta \in \llbracket x_j\rrbracket\} \\
\cup\ & \{\mathit{Res}(x_j) \subseteq \llparenthesis x_j\rrparenthesis\}
\end{aligned}$$

For $a_{\text{obj}} \equiv [f_i = z_i{}^{i=1..k}, m_j = \varsigma(y_j)b_j{}^{j=1..\ell}]$,

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, a_{\text{obj}}) \overset{\text{def}}{=}\ & \textstyle\bigcup_{i=1}^{k} \mathrm{Cons}(\vec{x}, z_i) \\
\cup\ & \textstyle\bigcup_{j=1}^{\ell} \mathrm{Cons}(\vec{x}y_j, b_j) \\
\cup\ & \{\llparenthesis z_i\rrparenthesis_\gamma = \langle y_j\rangle_{f_i\gamma} \mid i = 1..k,\ j = 1..\ell,\ f_i\gamma \in [y_j]\} \\
\cup\ & \{\llparenthesis b_j\rrparenthesis_\gamma = \langle y_{j'}\rangle_{m_j\gamma} \mid j, j' = 1..\ell,\ m_j\gamma \in [y_j]\} \\
\cup\ & \{\llparenthesis b_j\rrparenthesis = \langle y_{j'}\rangle_{m_j} \mid j, j' = 1..\ell,\ m_j \in [y_j]\} \\
\cup\ & \{\llparenthesis z_i\rrparenthesis_\gamma = \llparenthesis a_{\text{obj}}\rrparenthesis_{f_i\gamma} \mid i = 1..k,\ f_i\gamma \in \llbracket a_{\text{obj}}\rrbracket\} \\
\cup\ & \{\llparenthesis b_j\rrparenthesis_\alpha \subseteq \llparenthesis a_{\text{obj}}\rrparenthesis_{m_j\alpha} \mid j = 1..\ell,\ m_j\alpha \in \llbracket a_{\text{obj}}\rrbracket\} \\
\cup\ & \{\llparenthesis b_j\rrparenthesis_\beta = \llparenthesis a_{\text{obj}}\rrparenthesis_{m_j\beta} \mid j = 1..\ell,\ m_j\beta \in \llbracket a_{\text{obj}}\rrbracket\} \\
\cup\ & \{\llparenthesis b_j\rrparenthesis \subseteq \llparenthesis a_{\text{obj}}\rrparenthesis_{m_j} \mid j = 1..\ell,\ m_j \in \llbracket a_{\text{obj}}\rrbracket\} \\
\cup\ & \{T_{\text{obj}}(z_1 \cdots z_k) \subseteq \llparenthesis a_{\text{obj}}\rrparenthesis\}
\end{aligned}$$

For $a_{\text{let}} \equiv \mathit{let}\ x{=}a\ \mathit{in}\ b$,

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, a_{\text{let}}) \overset{\text{def}}{=}\ & \mathrm{Cons}(\vec{x}, a) \\
\cup\ & \mathrm{Cons}(\vec{x}x, b) \\
\cup\ & \{\llparenthesis a\rrparenthesis_\gamma = \langle x\rangle_\gamma \mid \gamma \in \llbracket a\rrbracket\} \\
\cup\ & \{\llparenthesis b\rrparenthesis_\alpha \subseteq w^{n+1,\alpha}_{n}\llparenthesis a_{\text{let}}\rrparenthesis_\alpha \mid \alpha \in \llbracket a_{\text{let}}\rrbracket\} \\
\cup\ & \{\llparenthesis b\rrparenthesis_\beta = w^{n+1,\beta}_{n}\llparenthesis a_{\text{let}}\rrparenthesis_\beta \mid \beta \in \llbracket a_{\text{let}}\rrbracket\} \\
\cup\ & \{\llparenthesis a\rrparenthesis; \llparenthesis b\rrparenthesis \subseteq \llparenthesis a_{\text{let}}\rrparenthesis\}\}
\end{aligned}$$

For $a_{\text{if}} \equiv \mathit{if}\ x_j\ \mathit{then}\ a_0\ \mathit{else}\ a_1$,

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, a_{\text{if}}) \overset{\text{def}}{=}\ & \{\llparenthesis x_j\rrparenthesis = \mathit{Res}(x_j)\} \\
\cup\ & \mathrm{Cons}(\vec{x}, a_0) \\
\cup\ & \mathrm{Cons}(\vec{x}, a_1) \\
\cup\ & \{\llparenthesis a_0\rrparenthesis_\alpha \lhd x \rhd \llparenthesis a_1\rrparenthesis_\alpha \subseteq \llparenthesis a_{\text{if}}\rrparenthesis_\alpha \mid \alpha \in \llbracket a_{\text{if}}\rrbracket\} \\
\cup\ & \{\llparenthesis a_0\rrparenthesis_\beta \lhd x \rhd \llparenthesis a_1\rrparenthesis_\beta = \llparenthesis a_{\text{if}}\rrparenthesis_\beta \mid \beta \in \llbracket a_{\text{if}}\rrbracket\} \\
\cup\ & \{\llparenthesis a_0\rrparenthesis \lhd x \rhd \llparenthesis a_1\rrparenthesis \subseteq \llparenthesis a_{\text{if}}\rrparenthesis\} \\
\mathrm{Cons}(\vec{x}, x_j.f) \overset{\text{def}}{=}\ & \{w^{n,f\gamma}_{j-1}\langle x_j\rangle_{f\gamma} = \llparenthesis x_j\rrparenthesis_{f\gamma} \mid f\gamma \in \llbracket x_j\rrbracket\} \\
\cup\ & \{\llparenthesis x_j\rrparenthesis_{f\alpha} \subseteq \llparenthesis x_j.f\rrparenthesis_\alpha \mid \alpha \in \llbracket x_j.f\rrbracket\} \\
\cup\ & \{\llparenthesis x_j\rrparenthesis_{f\beta} = \llparenthesis x_j.f\rrparenthesis_\beta \mid \beta \in \llbracket x_j.f\rrbracket\} \\
\cup\ & \{T_{\text{fsel}}(x_j, f) \subseteq \llparenthesis x_j.f\rrparenthesis\} \\
\mathrm{Cons}(\vec{x}, x_j.f{:=}x_k) \overset{\text{def}}{=}\ & \mathrm{Cons}(\vec{x}, x_j) \\
\cup\ & \{\llparenthesis x_j\rrparenthesis = \mathit{Res}(x_j)\} \\
\cup\ & \mathrm{Cons}(\vec{x}, x_k) \\
\cup\ & \{\llparenthesis x_k\rrparenthesis = \mathit{Res}(x_k)\} \\
\cup\ & \{\llparenthesis x_j\rrparenthesis_{f\gamma} = \llparenthesis x_k\rrparenthesis_\gamma \mid \gamma \in \llbracket x_k\rrbracket\} \\
\cup\ & \{\llparenthesis x_j\rrparenthesis_\alpha \subseteq \llparenthesis x_j.f{:=}x_k\rrparenthesis_\alpha \mid \alpha \in \llbracket x_j.f{:=}x_k\rrbracket\} \\
\cup\ & \{\llparenthesis x_j\rrparenthesis_\beta = \llparenthesis x_j.f{:=}x_k\rrparenthesis_\beta \mid \beta \in \llbracket x_j.f{:=}x_k\rrbracket \\
\cup\ & \{T_{\text{fupd}}(x_j, f, x_k) \subseteq \llparenthesis x_j.f{:=}x_k\rrparenthesis\}\}
\end{aligned}$$

Table 4: Constraints generating function. Here γ is implicitly assumed to range over non-empty sequences of field and method names and ending in a method name, α over non-empty sequences of method names and β over those γ containing at least one field name.

For $v_t \stackrel{\text{def}}{=} v, x{=}\mathtt{tt}$ and $v_f \stackrel{\text{def}}{=} v, x{=}\mathtt{ff}$,

$$\frac{\exists\vec{X}.\quad v, L \subseteq e_0 \triangleleft x \triangleright e_1 \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v_t, L \subseteq e_0 \quad\wedge\quad v_f, L \subseteq e_1 \quad\wedge\quad \Phi} \qquad x \text{ not in } v$$

(if-elimr)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{tt}, v', L \subseteq e_0 \triangleleft x \triangleright e_1 \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v, x{=}\mathtt{tt}, v', L \subseteq e_0 \quad\wedge\quad \Phi}$$

(if-betar1)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{ff}, v', L \subseteq e_0 \triangleleft x \triangleright e_1 \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v, x{=}\mathtt{ff}, v', L \subseteq e_1 \quad\wedge\quad \Phi}$$

(if-betar2)

Writing $L(L')$ to denote $L''[L'/X]$ where there is exactly one occurrence of X in L'',

$$\frac{\exists\vec{X}.\quad v, L(L_0 \triangleleft x \triangleright L_1) \subseteq e \quad\wedge\quad \Phi}{\exists\vec{X}.\ v_t, L(L_0) \subseteq e \quad\wedge\quad v_f, L(L_1) \subseteq e \quad\wedge\quad \Phi} \qquad x \text{ not in } v$$

(if-eliml)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{tt}, v', L(L_0 \triangleleft x \triangleright L_1) \subseteq e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v, x{=}\mathtt{tt}, v', L(L_0) \subseteq e \quad\wedge\quad \Phi}$$

(if-betal1)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{ff}, v', L(L_0 \triangleleft x \triangleright L_1) \subseteq e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v, x{=}\mathtt{ff}, v', L(L_1) \subseteq e \quad\wedge\quad \Phi}$$

(if-betal2)

Table 7: Simplification of $- \triangleleft - \triangleright -$ expressions. Here we, may assume that our verification conditions have form $\exists\vec{X}.\,(\bigwedge_i v_i', L \subseteq e_i') \wedge \Psi_0$ where Ψ_0 is first-order formula.

$$\frac{\exists\vec{X}.\quad T \subseteq \sigma X_1 \quad\wedge\quad \Phi}{\exists\vec{X}.\quad (\exists_\sigma T) \subseteq X_1 \quad\wedge\quad \Phi}$$

(adj-ren)

$$\frac{\exists\vec{X}.\quad T_0 \subseteq X_1 \quad\wedge\quad T_1 \subseteq X_1 \quad\wedge\quad \Phi}{\exists\vec{X}.\quad (T_0 \vee T_1) \subseteq X_1 \quad\wedge\quad \Phi}$$

(collate)

Table 8: Miscellaneous simplification rules. Here we assume that our VCs have form $\exists\vec{X}.\,(\bigwedge_i T_i \subseteq e_i) \wedge \Psi_0$ where Ψ_0 is a first-order formula, and there are no occurrences of $- \triangleleft - \triangleright -$ subexpressions (except for maybe in Ψ_0).

$$\frac{\exists\vec{X}X.\quad T \subseteq X \quad\wedge\quad \Phi}{\exists\vec{X}.\quad \Phi[\mu X.T/X]} \qquad X \text{ free in } T$$

(freeinst)

$$\frac{\exists\vec{X}X.\quad T \subseteq X \quad\wedge\quad \Phi}{\exists\vec{X}.\quad \Phi[T/X]} \qquad X \text{ not free in } T$$

(notfreeinst)

$$\frac{\exists\vec{X}X.\quad \Phi}{\exists\vec{X}.\quad \Phi} \qquad X \text{ not free in } \Phi$$

(falseinst)

Table 9: Minimal set instantiations. Here we assume that our VCs have form $\exists\vec{X}.\,(\bigwedge_i T_i \subseteq X_i) \wedge \Psi_0$ where Ψ_0 is a first-order formula, each $X \in \vec{X}$ occurs at most once to the right of a constraint $T \subseteq X$, and there are no occurrences of $- \triangleleft - \triangleright -$ subexpressions (except maybe in Ψ_0).

$$\frac{\exists\vec{X}.\quad v \Rightarrow e = e' \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v \Rightarrow e' = e \quad\wedge\quad \Phi}$$

(eq-sym)

$$\frac{\exists\vec{X}.\quad v \Rightarrow e_0 = e_1 \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v \Rightarrow e_0' = e_1 \quad\wedge\quad \Phi} \qquad \{e_0 \rightsquigarrow e_0'\}$$

(eq-resp)

$$\frac{\exists\vec{X}X.\quad \varepsilon \Rightarrow X = e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad \Phi[e/X]} \qquad X \text{ not free in } e$$

(eq-inst)

$$\frac{\exists\vec{X}.\quad \varepsilon \Rightarrow X = e \quad\wedge\quad \Phi}{\exists\vec{X}X'.\quad \varepsilon \Rightarrow X = e[X'/X] \quad\wedge\quad \Phi} \qquad X \text{ free in } e$$

(eq-idem)

For π_1, π_2 the pullback of σ, τ,

$$\frac{\exists\vec{X}.\quad v \Rightarrow \sigma X_1 = \tau X_2 \quad\wedge\quad \Phi}{\exists\vec{X}Z.\quad v \Rightarrow X_1 = \pi_1 Z \quad\wedge\quad v \Rightarrow X_2 = \pi_2 Z \quad\wedge\quad \Phi}$$

(eq-pullback)

For ρ the equaliser of σ, τ,

$$\frac{\exists\vec{X}.\quad v \Rightarrow \sigma X_1 = \tau X_1 \quad\wedge\quad \Phi}{\exists\vec{X}Z.\quad v \Rightarrow X_1 = \rho Z \quad\wedge\quad \Phi}$$

(eq-equaliser)

$$\frac{\exists\vec{X}.\quad v \Rightarrow \sigma X_1 = \tau\phi \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v \Rightarrow X_1 = (\sigma^{-1} \circ \tau)\phi \quad\wedge\quad \Phi} \qquad \sigma^{-1} \circ \tau \text{ a renaming}$$

(eq-const)

For $v_t \stackrel{\text{def}}{=} v, x{=}\mathtt{tt}$ and $v_f \stackrel{\text{def}}{=} v, x{=}\mathtt{ff}$,

$$\frac{\exists\vec{X}.\quad v \Rightarrow e_0 \triangleleft x \triangleright e_1 = e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v_t \Rightarrow e_0 = e \quad\wedge\quad v_f \Rightarrow e_1 = e \quad\wedge\quad \Phi} \qquad x \text{ not in } v,$$

(eq-if-elim)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{tt}, v' \Rightarrow e_0 \triangleleft x \triangleright e_1 = e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v, x{=}\mathtt{tt}, v' \Rightarrow e_0 = e \quad\wedge\quad \Phi}$$

(eq-beta1)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{ff}, v' \Rightarrow e_0 \triangleleft x \triangleright e_1 = e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v, x{=}\mathtt{ff}, v' \Rightarrow e_1 = e \quad\wedge\quad \Phi}$$

(eq-beta2)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{tt} \Rightarrow X_1 = e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v \Rightarrow X_1 = e \triangleleft x \triangleright X_1 \quad\wedge\quad \Phi}$$

(eq-if-intro1)

$$\frac{\exists\vec{X}.\quad v, x{=}\mathtt{ff} \Rightarrow X_1 = e \quad\wedge\quad \Phi}{\exists\vec{X}.\quad v \Rightarrow X_1 = X_1 \triangleleft x \triangleright e \quad\wedge\quad \Phi}$$

(eq-if-intro2)

Table 10: Equality elimination rules.