Automating Type Soundness Proofs via Decision Procedures and Guided Reductions

Don Syme and Andrew D. Gordon

Microsoft Research, Cambridge, U.K.

Abstract. Operational models of fragments of the Java Virtual Machine and the .NET Common Language Runtime have been the focus of considerable study in recent years, and of particular interest have been specifications and machine-checked proofs of type soundness. In this paper we aim to increase the level of automation used when checking type soundness for these formalizations. We present a semi-automated technique for reducing a range of type soundness problems to a form that can be automatically checked using a decidable first-order theory. Deciding problems within this fragment is exponential in theory but is often efficient in practice, and the time required for proof checking can be controlled by further hints from the user. We have applied this technique to two case studies, both of which are type soundness properties for subsets of the .NET CLR. These case studies have in turn aided us in our informal analysis of that system.

1 Introduction

Formalizations of virtual machines such as the Java Virtual Machine (JVM) or the .NET Common Language Runtime (CLR) have been the focus of considerable study in recent years [4, 11, 13, 14]. Of particular interest have been specifications and proofs of type soundness for these systems, frequently involving machine-checked proofs using interactive theorem provers [15–17]. While the automation available in interactive theorem provers has increased, both the kind of automation applied (e.g. rewriting) and the manner of its application (e.g. tactics) tend to be substantially ad hoc. The proof scripts needed to check these properties are often many thousands of lines long.

In this paper we aim to increase the level of automation applied to this problem, focusing on one particluar automated decision procedure and one particular form of user guidance. We isolate out the user's guidance into a component called a *guided reduction*, which indicates how to extract the relevant facts that make the proof go through for particular cases. Applying the reduction is an automated process that transforms the type soundness problem into a form that can be automatically checked using case-splitting and validity checking within a combination of decidable first-order theories. The particular decision procedure used in this paper is the algorithm used by the Stanford Validity Checker (SVC) [1], which has been successfully applied to large hardware verification proofs. We

have applied this technique to models of subsets of the CLR, which has in turn aided our informal analysis of that system.

This paper is structured as follows. In the remainder of this section we consider the background to this work, including a number of studies of the JVM. In §2 we describe Spark, a model of a fragment of the IL of the CLR, which is used for explanatory purposes in this paper. In §3 we describe *guided reductions*, our new technique for semi-automatically converting high-level statements of type soundness into a form suitable for analysis by an automated decision procedure. In §4 we apply this combination of techniques to two case studies, and in §5 we discuss interesting potential avenues for future work.

1.1 Background: Type Soundness for Virtual Machines

In this section we consider the typical structure of a type soundness specification for a virtual machine. A good supply of examples exists against which to compare this structure, e.g. [10–12, 15], and we have examined these examples to check that they fall within the general structure described here.

A structured operational semantics (SOS) used in a type soundness proof typically has the following components: (a) a formal description of programs; (b) a formal description of typechecking; (c) a formal description of execution; and (d) a type soundness property. The property typically specifies (i) certain errors do not occur during execution and (ii) the machine always makes progress.

Formal descriptions of systems as complex as the JVM or the CLR vary substantially according to the specification methodology used, the exact logic in which the system is formalized, and individual choices about how to model operations in the logic. However, the components above are always recognisable. The primary points of departure between different descriptions of the same system are the use of big-step v. small-step models of execution; the representation of error conditions; the atomicity of execution steps, and the degree of realism of the model of execution, e.g. whether it models features such as optimizations.

We now give example forms of the terms and predicates for the different components of a specifcation. We stress that the exact form of the functions and predicates differ in detail between systems, but the essence of the techniques used do not.

Programs: a type $Prog$ or programs p where $Prog$ is defined via structural types such as lists, finite maps, records, integers, strings, products and sums;

Checking: a predicate $p : \tau_p$, indicating that the program p has the given type τ_p. This is usually defined compositionally in terms of a number of predicates $p \vdash item : \tau_{item}$ indicating that various sub-components $item$ of p are well-typed given the context of the whole program.

Execution: a type $State$ of states s, an initial state s_0, a set of terminal states, and a relation $p \vdash s \rightsquigarrow s'$ indicating that if the machine is in state s running program p then it may take a step to state s'.

Given the relations and functions above, type soundness can be defined as inhabitation of the transition relation:

Proposition 1. *If $p : \tau$ and $p \vdash s_0 \rightsquigarrow^* s$ then either s is terminal or there exists an s' such that $p \vdash s \rightsquigarrow s'$.*

Propositions like this are typically proved via an invariant that specifies good states, i.e. a type *StateType* recording expected shapes S of state structures arising at runtime (stack frames, heap entries etc.), and a predicate $p \vdash s \leq: S$ indicating when a state conforms to a state type. Then the statement becomes:

Proposition 2. *If $p : \tau$ and $p \vdash s \leq: S$ then either s is terminal or there exists an s' such that $p \vdash s \rightsquigarrow s'$ and furthermore $p \vdash s' \leq: S$.*

Whether the proof of such a property is effectively automatable obviously depends on the nature of the relations $:, \rightsquigarrow$ and $\leq:$. We stress that previous work on machine-checking such propositions has applied essentially ad hoc automation techniques. While this paper does not attempt to achieve *complete* automation of the proofs of such properties, it offers a first step in that direction.

Related Work Wright and Felleisen's 1994 work presented a systematic syntactic approach to a range of type soundness proofs for source languages [18], and we have used many aspects of their methodology in this paper.

No prior work has attempted to systematically apply decision procedures or other particular automated techniques to type soundness proofs. However, there has been considerable work on using interactive theorem proving for these kinds of proofs [9, 11, 16, 17]. Syme's work on Java used a more restrictive proof style and applied decision procedures to prove resulting obligations [15]. There have been other efforts to formalize aspects of virtual machine descriptions but without mechanized proof checking [12, 4], as well as a set of extensive Abstract State Machine (ASM) descriptions of the JVM [13]. The work presented in this paper has also been inspired by Norrish's treatment of C [10] and the general background of HOL theorem proving [5].

2 Spark

We now give a concrete example of a type soundness specification that serves to motivate our techniques to substantially automate type soundness proofs. A larger case study is discussed in §4.2. Our example is motivated by the instruction set of the CLR [8] and is called Spark.

We describe execution and verification of Spark programs by programming functions in the Caml dialect of ML [7]. Our code avoids all the imperative features of ML and use no recursion. Hence, we can directly interpret our ML data structures and procedures as mathematical sets and total functions, respectively. We import our code into the DECLARE theorem prover [15], interpreting the ML definitions as phrases of higher order logic.

A program in the Spark bytecode language consists of a single method implementation, itself consisting of an array of instructions, paired with a signature. We use ML type definitions to describe indexes for particular program addresses,

arguments, and local variables, and to define numeric constants and the instruction set. Here the types `int` and `float` are the primitive type of integers and IEEE floating point numbers.

The Spark intermediate language:

```
type addr = int                 bytecode address
type arg_idx = int              argument index
type loc_idx = int              local variable index
type const =                    constant
  | Const_I of int                  integer
  | Const_F of float                IEEE f.p. number
type instr =                    instruction
  | I_ret                           exit the method
  | I_ldarg   of arg_idx            load an argument
  | I_starg   of arg_idx            store into an argument
  | I_ldc     of const              load an integer or float
  | I_ldloc   of loc_idx            load a local
  | I_stloc   of loc_idx            store into a local
  | I_br      of addr               unconditional branch
  | I_ble     of addr               conditional branch
  | I_pop                           pop an element off the stack
  | I_add                           addition
  | I_mul                           multiplication
```

The metadata accompanying a method implementation is a signature, which describes the number and types of its arguments, the type of its result, and the number and types of its local variables.

Item types, method signatures, methods:

```
type itemT                      item type
  = I                               signed integer
  | F                               IEEE f.p. number
type msig =                     method signature
 {argsT: itemT list;                argument types
  retT:  itemT;                     return types
  locsT: itemT list}                local variable types
type meth =                     method
 {msig:   msig;                     method signature
  instrs:  instr list}              method implementation
```

2.1 The Spark Execution and Verification Semantics

Our description of the execution of individual instructions is a 30 line ML function `step` that acts as a functional description of a deterministic transition relation. Its type involves the types `item` and `state` as follows:

Items, states, steps:

```
type 'a option = None | Some of 'a
type item                                          item
 = Int of int                                          integer
 | Float of float                                      IEEE f.p. number
type state =                                       execution state
 {args:  item list;                                    items in arguments
  locs:  item list;                                    items in local variables
  stack: item list;                                    items on the stack
  pc:    addr}                                         program counter
val step: meth -> state -> state option            type of step function
```

For space reasons we omit the full definition of `step` in this paper.

We represent verification checks by a relation that relies on being given a summary of information that would typically be inferred during the execution of a series of verification checks. Pusch and Nipkow have shown how to formalize the link between a verification algorithm and a relational view of the checks made during verification [11]. We follow their approach of defining the verification checks at particular instructions so that they could be shared between an algorithmic and relational specification.

A precondition on an address is simply a list of types representing the shape of the stack prior to execution of that address. The ML type `stackT` represents a precondition. Preconditions for all or some of the addresses in a method are represented by ML values of type `methT`, a list indexed by addresses.

Stack and method typings:

```
type stackT = itemT list        types of items on stack
type addrT = addr × stackT      address with its type
type methT = stackT list        stack type for each addr
```

Our main subroutine is a function `dests` that simulates an instruction and computes its destinations. It depends on a subroutine `effect` to simulate the effect of running an instruction on a precondition.

```
effect: msig × instr × stackT -> stackT option
dests: meth × addr × stackT -> addrT list option
```

The definition of these functions takes 30 lines of ML code. We omit these definitions for space reasons, though it is important to note that they do contain many details and checks which do not need to be mentioned by our later proof scripts.

The remainder of the checks are defined in relational form (we continue to use program-like syntax for consistency). The relation `hastype`(m, τ_m) is the primary typing predicate and means that the method m is well-typed with respect to the stack type τ_m. For the relation to hold, each of the instructions of the method must be well-typed with respect to the preconditions τ_m.[1][2]

[1] In our actual formulation the precondition on the first instruction is the empty stack.

[2] In the definition, `read : 'a list -> int -> 'a option` indexes into a list.

Typing predicates:

```
let destOK (methT, (daddr, daddrT)) ↔
  match (read methT daddr) with
  | None -> false
  | Some daddrT' -> daddrT = daddrT'
let addr_hastype (addr,m,mT) ↔
  match (read m addr) with
  | None -> true
  | Some(stackT) ->
      match (dests (m,addr,stackT)) with
      | Some(dests) -> ∀d ∈ dests. destOK (mT,d)
      | _ -> false
let hastype (m,mT) ↔
  ∀addr. addr_hastype (addr,m,mT)
```

2.2 Type Soundness for Spark and the Conformance Relations

In this section, we define what it means for an execution state to conform to a method typing, m_τ. The relation $\texttt{stateOK}(s, m_\tau)$ is the primary conformance predicate and means that the execution state s conforms to the preconditions m_τ. For the relation to hold, the arguments and locals must conform to the signature of the method, and there must be a precondition $stackT$ assigned to the current program counter by m_τ such that $s.\texttt{stack}$ conforms to $stackT$.

Conformant items, stack, locals, arguments and states:

```
let valOK(v,vT)  ↔
   match v,vT with
   | Int _, I -> true
   | Float _, F -> true
   | _,_ -> false
let stackOK(stack,stackT)  ↔
    (length(stack) = length(stackT)) ∧
    ∀j < length(stack). valOK(nth stack j,nth stackT j)
```

Likewise for `argsOK` and `locsOK`.

```
let stateOK(s,sT)  ↔
    argsOK(s.args,sT.argsT) ∧
    locsOK(s.locs,sT.locsT) ∧
    match read methT s.pc with
    | Some(stackT) -> stackOK (s.stack,stackT)
    | _ -> false
```

The key type soundness propositions can now be stated:

Proposition 3. *If* $\texttt{hastype}(m, \tau_m)$ *and* $\texttt{stateOK}(s, m_\tau)$ *then there exists and* s' *such that* $\texttt{step}(s) = \texttt{Some}(s')$ *and* $\texttt{stateOK}(s', m_\tau)$.

We now want to substantially automate the proof of this and similar, more complex propositions by using a decision procedure combined with user guidance.

3 Controlling Decision Procedures via Guided Reductions

Recent efforts have shown that it is feasible to machine-check type soundness properties for non-trivial models of execution and verification, certainly including propositions such as Proposition 3. However, the techniques currently require substantial amounts of human guidance.

We reiterate that it is an open question as to whether practical fully automated techniques exist for checking classes of type soundness properties, of which Proposition 3 is but one simple example. It is important to remember that the execution and verification checks can be almost arbitrarily more complex than the example shown in the previous section, all the way up to doing proofs about realistic implementations of fully featured virtual machines. Thus these properties are likely to remain a challenging and fertile area for applying automated techniques for some time to come. However in this paper we do not aim for full automation but rather seek to reduce and limit the amount of human intervention required in the proof effort.

For pragmatic reasons we are interested in using decision procedures to perform our automated reasoning, as it is necessary to produce simple counterexamples for failed proof efforts. We will apply the technique implemented by the SVC [1] (equally applicable would be its successor, CVC [2]). This procedure checks the validity of quantifier-free formula first-order logic with respect to theories for arithmetic, products, arrays (maps), sums and conditionals. Good counterexamples can be generated when proofs fail. SVC does not make use of axioms for finite discrimination or induction over datatypes, and the axioms it uses for indexed data structures apply equally to lists accessed using indexing functions, partial functions, total functions and finite maps. SVC has been successfully used for proofs about abstracted descriptions of microprocessors. It is an open question if other decision procedures (e.g. [6]) can be applied to the kind of proofs described in this paper.

Variations on the "transform and give to a decision procedure" theme are used on an ad hoc basis in theorem proving and the combination of *automatic* transformations with decision procedures occurs often in computing. However our transformations are *not* automatic: i.e. they are non-trivial and represent insight on the part of the user. So in order to demonstrate that we have not resorted to full-blown interactive theorem proving we need a characterization of the input specified by the user.

In this paper we rigorously separate proof checking into three steps:

1. The user specifies a transformation that reduces the problem to one within a recognised, decidable logic;
2. The system automatically applies the transformation;
3. The resulting formula is passed to a decision procedure, which returns OK or a counterexample.

When proof is divided in this away we call it *guided proof checking*, and we call our particular technique *guided reduction*. In contrast, we call techniques where the user interactively applies further proof methods to residue problems

interactive proof checking. Techniques where no human interaction is required are just called decision procedures.

3.1 Motivating Guided Reductions

Unfortunately, not all type soundness problems may be solved immediately by the application of an SVC-like decision procedure. Consider the following:

A Problem Not Immediately Provable by SVC

```
let valOK(v,vT) ↔ some large expression
let stackOK(stack,stackT)  ↔
    (length(stack) = length(stackT)) ∧
    ∀j < length(stack). valOK(el(j,stack),el(j,stackT))

stack0T <> [] ∧
stack1T = hd(stack0T)::hd(stack0T)::(tl(stack0T)) ∧
stack0 <> [] ∧
stack1 = hd(stack0)::hd(stack0)::(tl(stack0)) ∧
stackOK(stack0,stack0T)                              (A)
→ stackOK(stack1,stack1T)                            (B)
```

This is based on the case for the `dup` (duplicate) instruction when proving Proposition 1 from §2.2.

Such a problem is not immediately solvable via the SVC technique for two reasons. First, the operators `length`, `nth` and ∀ lie in theories not understood by the decision procedure, so the procedure regards them as *uninterpreted*. Looking at this another way, the way `stackOK` has been defined as a universal first order predicate has lead us into reasoning about both first-order and equational theories simultaneously. If we were using a first-order theorem prover, we could throw in axioms about these operations and perform a proof search. However it is well known that combining such problems is difficult, and while progress has been made recently to determine forms of such problems that are tractable [3], it is not yet clear if the techniques will scale up to very large verification problems while providing the high-quality counterexample feedback that is required.

Second, the problem statement may include large, irrelevant definitions such as that for `valOK`. A heuristic-based case-splitting decision procedure such as SVC can be easily misled by the presence of such terms. Better heuristics can help, but ultimately these definitions are only needed for some branches of a proof and their presence on other branches greatly hinders both the automation and the interpretation of counterexamples.

The predicate `stackOK` could be defined recursively, e.g. from left-to-right along the list. However the automated routine must then determine how many times to unwind that recursion. In addition, the number of unwindings depends upon the branch of the problem, and on some branches may be indexed by a parameter, for example when n arguments are consumed at a call instruction in a virtual machine. Furthermore, our techniques must cope with *random access* structures such as finite maps, which do not fit nicely into a recursive framework.

3.2 Our Proof Guidance

We now describe our technique to let the user avoid the problems associated with uncontrolled unwinding of large definitions and first-order quantifiers.

A *proof script* is made up of three parts:

- The specification of a set of *problematic predicates and functions*.
- The specification of how and where to apply the fundamental rules associated with problematic predicates. This is called a *guided reduction*.
- The specification of any additional heuristic information necessary for the efficiency of the decision procedure, for example case-split orderings.

This constitutes the full input specified by the user. The well-formedness of the guided reduction can be checked automatically. The process of applying the reduction involves: (a) expanding all definitions of all terms representing applications of rules; (b) expanding the definitions of all non-problematic predicates and functions; and (c) replacing pattern matching by the equivalent test/get form. The problem is then submitted to the decision procedure.

The *problematic predicates and functions* specified in the first part of the proof script are typically: those whose definition is recursive; those whose definition involves operators such as $\forall$ that lie outside the theory supported by the target decision procedure; those whose uncontrolled expansion creates an unacceptable blow-out in proof checking times; and those which are used in rules for other problematic predicates. The non-problematic predicates and functions are typically those which have a lone equational axiom of the form $p(x_1, \ldots, x_n) = Q[x_1, \ldots, x_n]$ where Q contains no problematic predicates.

In our problem domain, problem specifications are "complex but shallow", and the majority of predicates are not problematic. Those in the example from §3.1 are `valOK` and `stackOK`. There are no problematic functions.

We assume that a set of *rules* is available for problematic predicates. Often a rule is simply the definitions of a predicate, but it may also be a useful lemma, often one that follows immediately from its definition—see [15] for how such results can be derived automatically from definitional forms such as equations involving quantifiers or least-fixed-point operators. We consider the following forms of rules for a problematic predicate p:

Definitional. A rule of the form $p(\bar{\mathbf{x}}) \leftrightarrow Q[\bar{\mathbf{x}}]$.
Weakening. A rule of the form $p(\bar{\mathbf{x}}) \rightarrow Q[\bar{\mathbf{x}}]$.
Strengthening. A rule of the form $Q[\bar{\mathbf{x}}] \rightarrow p(\bar{\mathbf{x}})$.
Indexed (Weakening). A rule of the form $\forall \bar{\mathbf{y}}.\, p(\bar{\mathbf{x}}) \rightarrow Q[\bar{\mathbf{x}}, \bar{\mathbf{y}}]$.
Indexed (Strengthening). A rule of the form $\forall \bar{\mathbf{y}}.\, Q[\bar{\mathbf{x}}, \bar{\mathbf{y}}] \rightarrow p(\bar{\mathbf{x}})$.

Multiple such rules can exist for each problematic predicate and function. For example, here are two rules for `stackOK`, derived from the definition in §3.1.

```
stackOKHead  |- stackOK(stack,stackT) ↔
                  match (stack,stackT) with
                  | [],[] -> true
```

```
                  | (h::t),(hT::tT) -> valOK(h,hT)  ∧ stackOK(t,tT)
                  | _,_ -> false
stackOKHeadW |- stackOK(stack,stackT) ∧
                stack <> [] ∧
                stackT <> []
                → valOK(hd stack,hd stackT) ∧
                   stackOK(tl stack,tl stackT)
```

After a trivial rearrangement of quantifiers, `stackOKHead` is a definitional rule and `stackOKHeadW` is weakening rule. We rely on rules being in one of the above forms: this paper does not consider induction principles for inductive relations or recursive term structures, for example.

3.3 The Algebra of Guided Reductions

The second part of a proof script is a specification of how and where to apply the fundamental rules associated with problematic predicates, called a *guided reduction*. Consider the following informal specification of a guided reduction for the problem specified in §3.1 above: *"If the instruction is `dup`, apply the `stackOKHead` rule once to the input stack (as characterized by fact (A)) and twice to the output stack (as characterized by fact (B))."*

Informally, applying this reduction to the problem in §3.1 means replacing the specified facts and goals with the right of rule `stackOKHead` and leaving remaining instances of `stackOK` and `valOK` uninterpreted. The problem is then immediately solvable by a decision procedure such as SVC. In this example we have effectively used guided reductions for controlled rewriting of `stackOKHead`. Note that uncontrolled rewriting using rule `stackOKHead` would not terminate.

The above example indicates that guided reductions should be combinators that can be used together in meaningful ways, for example chaining, disjunction and conditionals, where the conditionals are based on abstracted criteria (a binary instruction) rather than primitive syntactic criteria (e.g. a list of specific instructions).

Formally, a guided reduction r for a predicate p is a specification of a replacement predicate for p using one of the following forms:

Identity. The predicate p itself.

A Rule Operator. An operator corresponding to one of the rules for p supplied with appropriate guided reductions as arguments, as specified below.

A Monotone Combinator. Guided reductions can be combined using combinators monotone or anti-monotone in each of their arguments: examples are given in below.

Guided reductions are categorized as *weakening* and/or *strengthening*. An *information preserving* rule is one that is both weakening and strengthening. The fundamental property required of a guided reduction is that a weakening reduction r for p must satisfy $\forall\bar{\mathbf{x}}.p(\bar{\mathbf{x}}) \rightarrow r(\bar{\mathbf{x}})$, and a strengthening reduction must satisfy $\forall\bar{\mathbf{x}}.r(\bar{\mathbf{x}}) \rightarrow p(\bar{\mathbf{x}})$. This is easily demonstrated for each of the forms we describe below.

Identity Reductions Each problematic predicate can itself be used as a guided reduction indicating that no reduction should be performed. These are information preserving. For example, the guided reduction `stackOK` is an information preserving guided reduction.

Rule Reductions Each definitional rule r of the form $p(\bar{\mathbf{x}}) \leftrightarrow P[\bar{\mathbf{x}}, q_1, \ldots, q_m]$ for problematic predicates $p, q_1, \ldots, q_m$ gives rise to an operator R parameterized by predicate variables $V_1, \ldots, V_n$, one for each occurrence of a problematic predicate in P (i.e. we have $n \geq m$). $R(Q_1, \ldots, Q_n)(\bar{\mathbf{x}})$ holds if and only if $P[\bar{\mathbf{x}}, V_1, \ldots, V_n]$. Here the replacement of $Q_1, \ldots, Q_n$ replaces the n individual occurrences of $q_1, \ldots, q_m$ in P.

For example, the rule `stackOKHead` gives rise to the following operator:[3]

```
stackOKHead V1 V2 (stack,stackT) ::=
   match (stack,stackT) with
     [],[] -> true
   | (h::t),(hT::tT) -> V1(h,hT)  ∧ V2(t,tT)
   | _,_ -> false
```

A position within a nesting of first order connectives is defined as *positive*, *negative* or *neutral* in the usual way, i.e. according to the markup scheme: $+ \wedge +$; $+ \vee +$; $- \rightarrow +$; $0 \leftrightarrow 0$; $\neg -$; $\forall +$; $\exists +$. For example, the variables V_1 and V_2 occur in positive positions on the right side of the definition of `stackOKHead`.

A guided reduction of the form $R(A_1, \ldots, A_n)$ is weakening (likewise strengthening) if each A_i is: weakening (likewise strengthening) when the corresponding position in P is positive; and strengthening (likewise weakening) when the corresponding position in P is negative; and information-preserving when the corresponding position in P is neutral.

For example, the guided reduction `stackOKHead`(p_1, p_2) is weakening if both p_1 and p_2 are weakening, and strengthening if both p_1 and p_2 are strengthening. In other words, each rule defines an operator that is monotone, anti-monotone or neutral in each of its predicate arguments according to the way the predicates corresponding to the arguments are used in the body of that rule.

Weakening (likewise strengthening) indexed rules of the form $\forall \bar{\mathbf{y}}.p(\bar{\mathbf{x}}) \rightarrow Q[\bar{\mathbf{x}}, \bar{\mathbf{y}}]$ for a problematic predicate p give rise to an operator for building weakening (likewise strengthening) guided reductions. For example, the following indexed rule justifies random access into a list:

```
argsOKPoint |- ∀ i. argsOK(args,argsT) →
                    match (read args i, read argsT i) with
                    | None,None -> true
                    | Some v, Some vT -> valOK(v,vT)
                    | _,_ -> false
```

For Spark this rule follows from the definition of argument conformance in §2.2. This gives the following operator for building weakening guided reductions:

[3] Here we use curried syntax for the extra higher-order arguments.

```
argsOKPointW i V (args,argsT) ::=
    match (read args i, read argsT i) with
    | None,None -> true
    | Some v, Some vT -> V(v,vT)
    | _,_ -> false
```

In other words, this operator lets us pick out an index i at which to reveal the fact that `valOK` holds, and furthermore to reveal additional information about that value by giving an appropriate argument for V. Thus the operators give a compact notation for supplying important instantiations and chaining inferences. In effect we are taking advantage of the fact that in "complex but shallow" problem domains specifying a few critical inferences can open the way for automation to do very useful amounts of work.

Once again conditions apply for the argument given to predicate variables such as V. In particular, if they are used in a positive (likewise negative) position on the right of the definition then they must be given a weakening (likewise strengthening) guided reduction. For example, the guided reduction `argsOKPointW 3 argsOK` is a weakening reduction, because `argsOK` is information preserving.

Combining Reductions It is now easy to write operators to combine guided reductions:

```
(p1 OR p2) ::= λx. p1(x) ∨ p2(x)
(p1 AND p2) ::= λx. p1(x) ∧ p2(x)
```

The operator `OR` is typically applied to guided reductions used to transform goals, and effectively describes multiple ways of proving the same goal. The operator `AND` is applied to guided reductions used to transform facts, and effectively describes how to derive multiple pieces of information from the same fact. Guided reductions built using these operators are weakening/strengthening if both arguments weakening/strengthening.

The following reductions discard facts/goals and are weakening/strengthening:

```
DoNotUse ::= λx. true
DoNotProve ::= λx. false
```

The `if/then/else` operator lets the user choose an appropriate reduction based on a condition. It is strengthening if both `p1` and `p2` are strengthening, likewise weakening. The `=>` operators let us conditionally extract extra information from a fact or goal for use on branches of a proof where the guard holds:

```
(if g then p1 else p2) ::= λx. (if g then p1(x) else p2(x))
(g => p1) && p2 ::= (if g then p1 else DoNotUse) AND p2
(g => p1) || p2 ::= (if g then p1 else DoNotProve) OR p2
```

3.4 Guided Reductions as Term Replacement and an Example

When authoring a guided reduction the user directly replaces uses of problematic predicates by applications of these operators. The correctness of this process can

be determined syntactically, by checking that the weakening (likewise strengthening) guided reductions are only applied to facts (likewise goals).

Guided reductions could be authored in other forms, e.g. as tactics in a theorem prover such as HOL or Isabelle. However, there are important practical benefits to representing guided reductions by predicate-replacement: (a) the terms are type-checked in combination with the term defining the problem itself, which captures many errors early on; (b) the terms may involve proof constants from the problem specification; and (c) the terms occur directly in position rather than as a later, disassociated operation, reducing the fragility of the guided-reduction vis-a-vis reorderings and restructurings of the problem statement.

Finally, back to our example. The guided reduction in §3.3 can be formalized by replacing the predicate `stackOK` in formulae (1) and (2) in §3.1 by

```
(i = I_dup) => (stackOKHead valOK stackOK) && stackOK
(i = I_dup) => (stackOKHead valOK (stackOKHead valOK stackOK)) || stackOK
```

respectively. This replacement is justified because the guided reductions are respectively weakening and strengthening, which can be automatically checked. The application of the guided reduction simply involves expanding all definitions of operators and non-problematic predicates and functions and applying the decision procedure, which then checks the validity of the resulting formula.

4 Case Studies

4.1 Case Study 1: Spark

We now consider the use of our techniques to prove Proposition 3. The overall proof script required to prove the first part of Proposition 3 is:[4]

Proof script for Spark Soundness (1)

```
Proposition:
  hastype(1) (m,methT) ∧
  stateOK(2) (s0,methT)
  --> step (m,s0) <> None
Problematic predicates: stateOK, stackOK, hastype, locsOK, argsOK
Replace (1) by: hastypePointW pc0
Replace (2) by:
  stateOKRule
   ((is_stloc i) => stackOKHead1 stackOK &&
    (is_starg i) => stackOKHead1 stackOK &&
    (is_binop i) => stackOKHead2 stackOK &&
    (is_ret i) => stackOKHead2 stackOK &&
    (is_ble i) => stackOKHead1 stackOK &&
    (is_pop i) => stackOKHead1 stackOK &&
    stackOK)
   ((is_stloc i or is_ldloc i) => locsOKPointW loc0 &&
```

[4] This is a tidied-up version of the actual proof script, which is a little more arcane.

```
    locsOK)
   ((is_starg i or is_ldarg i) => argsOKPointW arg0 &&
    argsOK)
Where pc0  ::= s0.pc
      i ::= match (read m.instrs pc0) with Some(i) -> i
      loc0 ::= (match i with I_stloc x -> x | I_ldloc x -> x)
      arg0 ::= (match i with I_starg x -> x | I_ldarg x -> x)
```

The rules for the problematic predicates `hastypePointW`, `stackOKHead1`, `stackOKHead2`, `argsOKPointW` and `locsOKPointW` are derived immediately from the definitions given in §2. The rule `stateOKRule` is also derived from the definition of `stateOK` in §2.2 and composes guided reductions for the input stack, locals and arguments.

In practice we prove the full soundness property in one step using a similar script, the proposition being:

```
hastype(m,methT) ∧ stateOK(s0,methT)
→ match step(s0) with
  | None -> false
  | Some(s1) -> state_ok(s1,methT)
```

The really promising thing about these proof scripts is just how much has *not* been mentioned. In particular, if we examine the definitions in §2, no mention has been made of functions such as `step`, `dests` or `effect`. The decision procedure is fed a very large term with all the definitions of these functions expanded. The process of case-splitting through all the instructions and all the failure/success cases implicit in the execution and verification semantics happens automatically.

After applying this reduction, the resulting formula is passed to the decision procedure and a counterexample, if any, is returned. The the expanded problem sent to the decision procedure would run for hundreds of pages (many subterms are shared within the problem). Our implementation of the SVC decision procedure takes 14.4s to prove the first part of the soundness proof (using a 750 MHz Pentium III), with 5217 case-splits and 1377 unique terms constructed.

We have found that proof times can be dramatically reduced by simple and natural case-split orderings. For example, if we specify that the first split should be on the kind of instruction, the time reduces to 0.25s with 109 case-splits.

Producing Counterexamples. Consider what happens if we omit a check from our verification rules, in particular if we omit the check that the type of the item on the stack for the instruction `starg` matches the type expected for the argument slot. Type soundness no longer holds, and a 30 line counterexample is printed, containing, among other things:

```
is_starg i0
is_F (nth m.mref.argsT (starg_get0 i0))
is_Int (hd s0.stack)
```

This suggests that the verifier is unsound when the instruction is a `starg`, a floating point number is expected, but the first value on the stack is an integer

value—the bug has been detected! Counterexample predicates like these can also be solved to give a sample input (with some unknowns) that exposes the error. The counterexample could be made concrete by searching for arbitrary terms which satisfy any remaining non-structural constraints, but we have not implemented this.

4.2 Case Study 2: Investigating BIL

Our second case study consists of verifying the type soundness of a small-step term rewriting system corresponding to the BIL fragment described in [4]. The fragment included a subtyping relation with appropriate rules. Some aspects of this proof are beyond the scope of this paper, in particular the use of guided reductions in the presence of inductively defined relations over recursive term structures. Apart from this the core technique used was as described in §3.

The BIL instructions for which we have verified the corresponding soundness property include loading a constant; sequencing; conditionals; loops; virtual call instructions; loading/storing arguments; boxing inline values to objects; allocating new objects; creating new inline objects; loading the address of an argument; loading the address of a field in an object; loading/storing via a pointer.

The components of the specification are: (a) a term model of programs consisting of 20 lines of ML datatype definitions; (b) a pseudo-functional small-step execution semantics comprising 180 lines of ML code, including some uninterpreted operations (e.g. a function that resolves virtual call dispatch is assumed); (c) a functional type checker comprising 90 lines of ML code and some uninterpreted operations; and (d) a specification of conformance akin to that in [15].

The proof assumes the following lemmas, which we have proved by inspection: (a) weakening/strengthening rules about the problematic predicates; (b) 11 lemmas about recursive operations such as "write into a nested location within a struct (inline value) given a path into that value"; (c) one lemma about the existence of a heap typing that records the types of all allocations that will occur during execution; and (d) a lemma connecting the typechecking process to the term-conformance predicate, of the kind stated and proved in Chapter 7 of [15].

The overall guided reduction was specified in tabular form, with 41 rows (each corresponding to one rewrite action in the execution semantics), and 7 columns (specifying guided reductions for the input heap, stack frames, input term, execution step, output heap, output stack frames and output term). The table was sparse, with 60% of entries indicating that no special reasoning was needed for that item on that branch of the proof. This left around 120 entries each a couple of identifiers long. In contrast the proof performed in [15] took around 2000 lines of proof script, despite using considerable automation.

We executed the proof for each the instruction independently, and each instruction took under 10s to verify. We found mistakes in both our verifier and our model of execution during this process.

5 Conclusions and Future Directions

This paper has presented a new semi-automated technique for mechanically checking the type soundness of virtual machines, and two case studies applying that technique. It is the first time SVC-like decision procedures have been extensively applied to a problem domain that was previously exclusively tackled using interactive theorem proving.

The manual part of the proof technique is based on an algebra of guided reductions built using combinators that are automatically derived from definitions and rules for the predicates and functions being manipulated. The guided reductions allow the user to control the unwinding of recursive definitions and to give instantiations for certain crucial first-order rules. This gives a compact but controlled way of specifying the information necessary for different parts of a proof, and the proof hints can be combined to express finite proof search and conditional guidance. The automated part of the proof uses SVC-like validity checking for a quantifier free theory of arithmetic and structured terms. Although exponential in theory this has proved efficient and controllable in practice, sometimes by giving hints for case-split orderings. This mirrors experiences with using these algorithms for hardware verification [1].

We have also described two case studies applying these techniques to fragments of the CLR's intermediate language. When compared to interactive theorem proving, these case studies have certainly benefited from the increased use of automation. We found that the semi-automatic proof checking process was effective in helping us understand aspects of the second, larger case study. The results from the case-studies indicate that the problem domain is highly automatable and that it is worthwhile to pursue a disciplined combination of proof guidance and proof automation.

With regard to future possible directions, it is certain that further automation can be applied in this problem domain, perhaps even achieving fully automated checking for important classes of soundness properties. It is also likely that properties other than type soundness can benefit from the approach we have taken in this paper. In addition, applying our present combination of techniques to new specifications will reveal if they transfer in practice. For example, applying the techniques outlined in this paper to the recent extensive ASM descriptions of the JVM [13] would determine if they scale to larger formal models.

The proof guidance technique described in this paper is novel, especially the automatic generation of combinators for a proof algebra from a specification of basic axioms for problematic predicates and functions. We have not described how inductive and other second-order proof techniques fit into this framework. It would also be very interesting to apply similar techniques to other problem domains. In particular there is a strong need for disciplined ways of decomposing hardware verification properties into problems that can be independently model checked. Guided reductions may have a role to play here.

References

1. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.
2. C. Barrett, D. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In *Frontiers of Combining Systems (FROCOS)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, April 2002.
3. A. Degtyarev and A. Voronkov. Equality reasoning in sequent-based calculi. In *Handbook of Automated Reasoning, Volume I*, pages 611–706. Elsevier Science and MIT Press, 2001.
4. A. Gordon and D. Syme. Typing a multi-language intermediate code. In *27th Annual ACM Symposium on Principles of Programming Languages*, January 2001.
5. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.
6. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.
7. Xavier Leroy. *The Objective Caml system, documentation and user's guide*. INRIA, Rocquencourt, 1999. Available from `http://caml.inria.fr`.
8. Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.
9. Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.
10. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
11. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *TACAS'99*, Lecture Notes in Computer Science. Springer Verlag, 1999.
12. Z. Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *Lecture Notes in Computer Science*, pages 271–312. Springer Verlag, 1999.
13. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine*. Springer Verlag, 2001.
14. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings POPL'98*, pages 149–160. ACM Press, 1998.
15. D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.
16. M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, May 1996.
17. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *Lecture Notes in Computer Science*, pages 119–156. Springer Verlag, 1999.
18. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.