The Guardol Language and Verification System

David Hardin[1], Konrad Slind[1], Michael Whalen[2], and Tuan-Hung Pham[2]

[1] Rockwell Collins Advanced Technology Center
[2] University of Minnesota

Abstract. Guardol is a domain-specific language designed to facilitate the construction of correct network guards operating over tree-shaped data. The Guardol system generates Ada code from Guardol programs and also provides specification and automated verification support. Guard programs and specifications are translated to higher order logic, then deductively transformed to a form suitable for a SMT-style decision procedure for recursive functions over tree-structured data. The result is that difficult properties of Guardol programs can be proved fully automatically.

1 Introduction

A *guard* is a device that mediates information sharing over a network between security domains according to a specified policy. Typical guard operations include reading field values in a packet, changing fields in a packet, transforming a packet by adding new fields, dropping fields from a packet, constructing audit messages, and removing a packet from a stream.

Guards are becoming prevalent, for example, in coalition forces networks, where selective sharing of data among coalition partners in real time is essential. One such guard, the Rockwell Collins Turnstile high-assurance, cross-domain guard [7], provides directional, bi-directional, and all-way guarding for up to three Ethernet connected networks. The proliferation of guards in critical applications, each with its own specialized language for specifying guarding functions, has led to the need for a portable, high-assurance guard language.

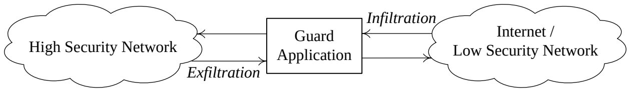

Fig. 1. Typical guard configuration

Guardol is a new, domain-specific programming language aimed at improving the creation, verification, and deployment of network guards. Guardol supports the ability to target a wide variety of guard platforms, the ability to glue together existing or mandated functionality, the generation of both implementations and formal analysis artifacts, and sound, highly automated formal analysis. Messages to be guarded, such as XML, may have recursive structure; thus a major aspect of Guardol is datatype declaration facilities similar to those available in functional languages such as SML [14] or Haskell [16]. Recursive programs over such datatypes are supported by ML-style pattern-matching. However, Guardol is not simply an adaptation of a functional language to guards. In fact, much of the syntax and semantics of Guardol is similar to that of Ada: Guardol is a sequential imperative language with non-side-effecting expressions, assignment, sequencing, conditional commands, and procedures with **in/out** variables. To a first approximation **Guardol** = **Ada** + **ML**. This hybrid language supports writing complex programs over complex data structures, while also providing standard programming constructs from Ada.

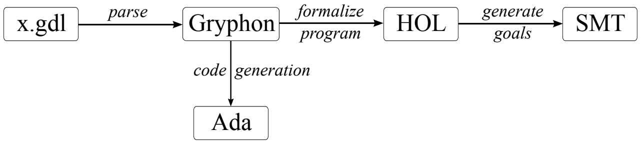

Fig. 2. Guardol system components

The Guardol system integrates several distinct components, as illustrated in Figure 2. A Guardol program in file `x.gdl` is parsed and typechecked by the Gryphon verification framework [13] developed by Rockwell Collins. Gryphon provides a collection of passes over Guardol ASTs that help simplify the program. From Gryphon, guard implementations can be generated—at present only in Ada—from Guardol descriptions. For the most part, this is conceptually simple since much of Guardol is a subset of Ada. However datatypes and pattern matching need special treatment: the former requires automatic memory management, which we have implemented via a reference-counting style garbage collection scheme, while the latter requires a phase of pattern-match compilation [18]. Since our intent in this paper is mainly to discuss the verification path, we will omit further details.

2 An example guard

In the following we will examine a simple guard written in Guardol. The guard applies a platform-supplied *dirty-word* operation DWO over a binary tree of messages (here identified with strings). When applied to a message, DWO can leave it unchanged, change it, or reject it with an audit string via MsgAudit.

```
type Msg       = string;
type MsgResult = {MsgOK : Msg | MsgAudit : string};
imported function DWO(Text : in Msg, Output : out MsgResult);
```

A MsgTree is a binary tree of messages. A MsgTree element can be a Leaf or a Node; the latter option is represented by a record with three fields. When the guard processes a MsgTree it either returns a new, possibly modified, tree, or it returns an audit message.

```
type MsgTree    = {Leaf | Node : [Value : Msg; Left : MsgTree; Right : MsgTree]};
type TreeResult = {TreeOK : MsgTree | TreeAudit : string};
```

The guard procedure takes its input tree in variable Input and the return value, which has type TreeResult, is placed in Output. The body uses local variables for holding the results of recursing into the left and right subtrees, as well as for holding the result of calling DWO. The guard code is written as follows:

```
function Guard (Input : in MsgTree, Output : out TreeResult) =
begin
   var ValueResult : MsgResult;
       LeftResult, RightResult : TreeResult;
   in
   match Input with
   MsgTree'Leaf ⇒ Output := TreeResult'TreeOK(MsgTree'Leaf);
   MsgTree'Node node ⇒ begin
      DWO(node.Value, ValueResult);
      match ValueResult with
      MsgResult'MsgAudit A ⇒ Output := TreeResult'TreeAudit(A);
      MsgResult'MsgOK ValueMsg ⇒ begin
         Guard(node.Left, LeftResult);
         match LeftResult with
         TreeResult'TreeAudit A ⇒ Output := LeftResult;
         TreeResult'TreeOK LeftTree ⇒ begin
            Guard(node.Right, RightResult);
            match RightResult with
            TreeResult'TreeAudit A ⇒ Output := RightResult;
            TreeResult'TreeOK RightTree ⇒
               Output := TreeResult'TreeOK(MsgTree'Node
                            [Value : ValueMsg, Left : LeftTree, Right : RightTree]);
```

The guard processes a tree by a pattern-matching style case analysis on the Input variable. There are several cases to consider. If Input is a leaf node, processing succeeds. This is accomplished by tagging the leaf with TreeOK and assigning to Output. Otherwise, if Input is an internal node (MsgTree′Node node), the guard applies DWO to the message held at the node and recurses through the subtrees (recursive calls are marked with boxes). The only complication arises from the fact that an audit may arise from a subcomputation and must be immediately propagated. The code essentially lifts the error monad of the external operation to the error monad of the guard.

Specifying guard properties Many verification systems allow programs to be annotated with assertions. Under such an approach, a program may become cluttered with assertions and assertions may involve logic constructs. Since we wanted to avoid clutter and wanted to shield programmers, as much as possible, from learning the syntax of a logic language, we decided to express specifications using Guardol programs. The key language construct facilitating verification is the *specification* declaration: it presents some code to be executed, sprinkled with assertions (which are just boolean program expressions).

Following is the specification for the guard. The code runs the guard on the tree t, putting the result in r, which is either a TreeOK element or an audit (TreeAudit). If the former, then the returned tree is named u and Guard_Stable, described below, must hold on it. On the other hand, if r is an audit, the property is vacuously true.

```
spec Guard_Correct = begin
    var t : MsgTree; r : TreeResult;
    in if (∀(M : Msg). DWO_Idempotent(M)) then begin
        Guard(t, r);
        match r with
            TreeResult'TreeOK u ⇒ check Guard_Stable(u);
            TreeResult'TreeAudit A ⇒ skip;
    else skip;
```

The guard code is essentially parameterized by an arbitrary policy (DWO) on how messages are treated. The correctness property simply requires that the result obeys the policy. In other words, suppose the guard is run on tree t, returning tree u. If DWO is run on every message in u, we expect to get u back unchanged, since all dirty words should have been scrubbed out in the passage from t to u. This property is a kind of idempotence, coded up in the function Guard_Stable; note that it has the shape of a *catamorphism*, which is a simple form of recursion exploited by our automatic proof component.

```
function Guard_Stable (MT : in MsgTree) returns Output : bool = begin
    var R : MsgResult;
    in
    match MT with
        MsgTree'Leaf ⇒ Output := true;
        MsgTree'Node node ⇒ begin
            DWO(node.Value, R);
            match R with
                MsgResult'MsgOK M ⇒ Output := (node.Value = M);
                MsgResult'MsgAudit A ⇒ Output := false;
            Output := Output and Guard_Stable(node.Left) and Guard_Stable(node.Right);
```

The success of the proof depends on the assumption that the external dirty-word operation is idempotent on messages, expressed by the following program.

```
function DWO_Idempotent(M : in Msg) returns Output : bool = begin
    var R1, R2 : MsgResult;
    in
    DWO(M, R1);
    match R1 with
        MsgResult'MsgOK M2 ⇒ begin
            DWO(M2, R2);
            match R2 with
                MsgResult'MsgOK M3 ⇒ Output := (M2 = M3);
                MsgResult'MsgAudit A ⇒ Output := false;
        MsgResult'MsgAudit A ⇒ Output := true;
```

DWO_Idempotent calls DWO twice, and checks that the result of the second call is the same as the result of the first call, taking into account audits. If the first call returns an audit, then there is no second call, so the idempotence property is vacuously true. On the other hand, if the first call succeeds, but the second is an audit, that means that the first call somehow altered the message into one provoking an audit, so idempotence is definitely false.

3 Generating verification conditions by deduction

Verification of guards is performed using HOL4 [19] and OpenSMT [2]. First, a program is mapped into a formal AST in HOL4, where the operational semantics of Guardol are encoded. One can reason directly in HOL4 about programs using the operational semantics; unfortunately, such an approach has limited applicability, requiring expertise in the use of a higher order logic theorem prover. Instead, we would like to make use of the high automation offered by SMT systems. An obstacle: current SMT systems do not understand operational semantics.[3] We surmount the problem in two steps. First, *decompilation into logic* [15] is used to deductively map properties of a program in an operational semantics to analogous properties over a mathematical function equivalent to the original program. This places us in the realm of proving properties of recursive functions operating over recursive datatypes, an undecidable setting in general. The second step is to implement a decision procedure for functional programming [20]. This procedure necessarily has syntactic limitations, but it is able to handle a wide variety of interesting programs and their properties fully automatically.

Translating programs to footprint functions First, any datatypes in the program are defined in HOL (every Guardol type can be defined as a HOL type). Thus, in our example, the types MsgTree, MsgResult, and TreeResult are translated directly to HOL datatypes. Recognizers and selectors, *e.g.*, isMsgTree_Leaf and destMsgTree_Node, for these types are automatically defined and used to translate pattern-matching statements in programs to equivalent if-then-else representations. Programs are then translated into HOL *footprint*[4] function definitions. If a program is recursive, then a recursive function is defined. (Defining recursive functions in higher order logic requires a termination proof; for our example termination is automatically proved.) Note that the HOL function for the guard (following) is second order due to the external operator DWO: all externals are held in a record *ext* which is passed as a function argument.

```
Guard ext Input =
if isMsgTree_Leaf Input then
    TreeResult_TreeOK MsgTree_Leaf
else
    let ValueResult = ext.DWO ((destMsgTree_Node Input).Value, ARB)
    in if isMsgResult_MsgAudit ValueResult then
        TreeResult_TreeAudit (destMsgResult_MsgAudit ValueResult)
    else
        let LeftResult = Guard ext ((destMsgTree_Node Input).Left, ARB)
        in if isTreeResult_TreeAudit LeftResult then
            LeftResult
        else
            let RightResult = Guard ext ((destMsgTree_Node Input).Right, ARB)
            in if isTreeResult_TreeAudit RightResult then
                RightResult
            else
                TreeResult_TreeOK(MsgTree_Node {Value := destMsgResult_MsgOK ValueResult;
                                                Left  := destMsgResult_MsgOK LeftResult;
                                                Right := destMsgResult_MsgOK RightResult})
```

[3] Reasons for this state of affairs: there is not one operational semantics because each programming language has a different semantics; moreover, the decision problem for such theories is undecidable.

[4] Terminology lifted from the separation logic literature.

3.1 Guardol operational semantics

The operational semantics of Guardol (see Fig. 3) describes program evaluation by an inductively defined judgement saying how statements alter the program state. The formula $\mathsf{STEPS}\ \mathit{\Gamma}\ \mathit{code}\ s_1\ s_2$ says "evaluation of statement *code* beginning in state s_1 terminates and results in state s_2". (Thus we are giving a so-called *big-step* semantics.) Note that $\mathit{\Gamma}$ is an environment binding procedure names to procedure bodies. The semantics follows an approach taken by Norbert Schirmer [17], wherein he constructed a *generic* semantics for a large class of sequential imperative programs, and then showed how to specialize the generic semantics to a particular programming language (a subset of C, for him). Similarly, Guardol is another instantiation of the generic semantics.

Evaluation is phrased in terms of a *mode* of evaluation, which describes a computation state. A computation state is either in Normal mode, or in one of a set of abnormal modes, including Abrupt, Fault, and Stuck. Usually computation is in Normal mode. However, if a Throw is evaluated, then computation proceeds in Abrupt mode. If a Guard command returns false, the computation transitions into a Fault mode. Finally, if the Stuck mode is entered, something is wrong, *e.g.*, a procedure is called but there is no binding for it in $\mathit{\Gamma}$.

$$\mathit{[Skip]}\frac{}{\textsf{STEPS }\Gamma\textsf{ Skip (Normal s) (Normal s)}} \qquad \mathit{[Basic]}\frac{}{\textsf{STEPS }\Gamma\textsf{ (Basic f) (Normal s) (Normal (f s))}}$$

$$\mathit{[Seq]}\frac{\textsf{STEPS }\Gamma\textsf{ c}_1\textsf{ (Normal s}_1\textsf{) s}_2 \qquad \textsf{STEPS }\Gamma\textsf{ c}_2\textsf{ s}_2\textsf{ s}_3}{\textsf{STEPS }\Gamma\textsf{ (Seq c}_1\textsf{ c}_2\textsf{) (Normal s}_1\textsf{) s}_3} \qquad \begin{array}{l}\mathit{[withState]}\\ \dfrac{\textsf{STEPS }\Gamma\textsf{ (f s}_1\textsf{) Normal s}_1\textsf{) s}_2}{\textsf{STEPS }\Gamma\textsf{ (withState f) (Normal s}_1\textsf{) s}_2}\end{array}$$

$$\begin{array}{l}\mathit{[Cond\text{-}True]}\\ \dfrac{P(s_1) \qquad \textsf{STEPS }\Gamma\textsf{ c}_1\textsf{ (Normal s}_1\textsf{) s}_2}{\textsf{STEPS }\Gamma\textsf{ (Cond P c}_1\textsf{ c}_2\textsf{) (Normal s}_1\textsf{) s}_2}\end{array} \qquad \begin{array}{l}\mathit{[Cond\text{-}False]}\\ \dfrac{\neg P(s_1) \qquad \textsf{STEPS }\Gamma\textsf{ c}_2\textsf{ (Normal s}_1\textsf{) s}_2}{\textsf{STEPS }\Gamma\textsf{ (Cond P c}_1\textsf{ c}_2\textsf{) (Normal s}_1\textsf{) s}_2}\end{array}$$

$$\mathit{[Call]}\frac{M.p \in \textsf{Dom}(\Gamma) \qquad \mathit{\Gamma}(M.p) = c \qquad \textsf{STEPS }\Gamma\textsf{ c (Normal s}_1\textsf{) s}_2}{\textsf{STEPS }\Gamma\textsf{ (Call M.p) (Normal s}_1\textsf{) s}_2}$$

$$\mathit{[Call\text{-}NotFound]}\frac{M.p \notin \textsf{Dom}(\Gamma)}{\textsf{STEPS }\Gamma\textsf{ (Call M.p) (Normal s) Stuck}}$$

$$\begin{array}{l}\mathit{[Fault\text{-}Sink]}\\ \dfrac{}{\textsf{STEPS }\Gamma\textsf{ c (Fault f) (Fault f)}}\end{array} \qquad \begin{array}{l}\mathit{[Stuck\text{-}Sink]}\\ \dfrac{}{\textsf{STEPS }\Gamma\textsf{ c Stuck Stuck}}\end{array} \qquad \begin{array}{l}\mathit{[Abrupt\text{-}Sink]}\\ \dfrac{}{\textsf{STEPS }\Gamma\textsf{ c (Abrupt s) (Abrupt s)}}\end{array}$$

$$\mathit{[While\text{-}True]}\frac{P(s_1) \qquad \textsf{STEPS }\Gamma\textsf{ c (Normal s}_1\textsf{) s}_2 \qquad \textsf{STEPS }\Gamma\textsf{ (While P c) s}_2\textsf{ s}_3}{\textsf{STEPS }\Gamma\textsf{ (While P c) (Normal s}_1\textsf{) s}_3}$$

$$\mathit{[While\text{-}False]}\frac{\neg P(s)}{\textsf{STEPS }\Gamma\textsf{ (While P c) (Normal s) (Normal s)}}$$

Fig. 3. Evaluation rules

3.2 Decompilation

The work of Myreen [15] shows how to decompile assembly programs to higher order logic functions; we do the same here for Guardol, a high-level language. A decompilation theorem

$$\begin{array}{l} \vdash \forall s_1\ s_2.\ \forall x_1 \ldots x_k. \\ \quad s_1.proc.v_1 = x_1 \wedge \cdots \wedge s_1.proc.v_k = x_k \wedge \\ \quad \mathsf{STEPS}\ \Gamma\ \boxed{code}\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2) \\ \quad \Rightarrow \\ \quad \texttt{let}\ (o_1, ..., o_n) = \boxed{f(x_1, \ldots, x_k)} \\ \quad \texttt{in}\ s_2 = s_1\ \texttt{with}\{proc.w_1 := o_1, \ldots, proc.w_n := o_n\} \end{array}$$

essentially states that evaluation of *code* implements footprint function f. The antecedent $s_1.proc.v_1 = x_1 \wedge \cdots \wedge s_1.proc.v_k = x_k$ equates $x_1 \ldots x_k$ to the values of program variables $v_1 \ldots v_k$ in state s_1. These values form the input for the function f, which delivers the output values which are used to update s_1 to s_2.[5] Presently, the decompilation theorem only deals with code that starts evaluation in a Normal state and finishes in a Normal state.

The decompilation algorithm Now we consider how to prove decompilation theorems for Guardol programs. It is important to emphasize that *decompilation is an algorithm*. It always succeeds, provided that all footprint functions coming from the Guardol program have been successfully proved to terminate.

Before specifications can be translated to goals, the decompilation theorem

$$\vdash \forall s_1\ s_2.\ \ldots \mathsf{STEPS}\ \Gamma\ \boxed{\mathsf{Call}(qid)}\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2) \Rightarrow \cdots$$

is formally proved for each procedure *qid* in the program, relating execution of the code for procedure *qid* with the footprint function for *qid*.

Table 1. Rewrite rules in the decompilation algorithm

Condition	Rewrite rule
$code = \mathsf{Skip}$	$\vdash \mathsf{STEPS}\ \Gamma\ \mathsf{Skip}\ \mathsf{s}_1\ \mathsf{s}_2 = (\mathsf{s}_1 = \mathsf{s}_2)$
$code = \mathsf{Basic(f)}$	$\vdash \mathsf{STEPS}\ \Gamma\ \mathsf{Basic}\ (\mathsf{f})\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2) = (\mathsf{s}_2 = \mathsf{f}\ \mathsf{s}_1)$
$code = \mathsf{Seq}(\mathsf{c}_1, \mathsf{c}_2)$	$\vdash \mathsf{STEPS}\ \Gamma\ (\mathsf{Seq}(\mathsf{c}_1, \mathsf{c}_2))\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2) =$ $\exists t. \mathsf{STEPS}\ \Gamma\ \mathsf{c}_1\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{t})\ \wedge \mathsf{STEPS}\ \Gamma\ \mathsf{c}_2\ (\mathsf{Normal}\ \mathsf{t})\ (\mathsf{Normal}\ \mathsf{s}_2)$
$code = \mathsf{Cond}(\mathsf{P}, \mathsf{c}_1, \mathsf{c}_2)$	$\vdash \mathsf{STEPS}\ \Gamma\ (\mathsf{Cond}(\mathsf{P}, \mathsf{c}_1, \mathsf{c}_2))\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2) =$ $\mathsf{if}\ \mathsf{P}\ \mathsf{s}_1\ \mathsf{then}\ \mathsf{STEPS}\ \Gamma\ \mathsf{c}_1\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2)$ $\mathsf{else}\ \ \mathsf{STEPS}\ \Gamma\ \mathsf{c}_2\ (\mathsf{Normal}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_2)$
$code = \mathsf{withState\ f}$	$\vdash \mathsf{STEPS}\ \Gamma\ (\mathsf{withState\ f})\ (\mathsf{Normal}\ \mathsf{s}_1)\ \mathsf{s}_2 = \mathsf{STEPS}\ \Gamma\ (\mathsf{f}\ \mathsf{s}_1)\ (\mathsf{Normal}\ \mathsf{s}_1)\ \mathsf{s}_2$
$code = \mathsf{Call}\ qid$	depend on whether the function is recursive or not

Decompilation proofs are automated by forward symbolic execution of *code*, using an environment of decompilation theorems to act as summaries for procedure calls. Table 1 presents rules used in the decompilation algorithm. For the

[5] In our modelling, a program state is represented by a record containing all variables in the program. The notation $s.proc.v$ denotes the value of program variable v in procedure *proc* in state s. The `with`-notation represents record update.

most part, the rules are straightforward. We draw attention to the Seq, $\mathsf{withState}$, and Call rules. The Seq (sequential composition) rule conjoins the results of simpler commands and introduces an existential formula ($\exists t. \ldots$). However, this is essentially universal since it occurs on the left of the top-level implication in the goal; thus it can be eliminated easily and occurrences of t can thenceforth be treated as Skolem constants. Both blocks and procedure calls in the Guardol program are encoded using $\mathsf{withState}$. An application of $\mathsf{withState}$ stays in the current state, but replaces the current code by new code computed from the current state. Finally, there are two cases with the Call (procedure call) rule:

- The call is not recursive. In this case, the decompilation theorem for *qid* is fetched from the decompilation environment and instantiated, so we can derive
$$\begin{array}{l}\texttt{let}\ (o_1, ..., o_n) = f(x_1, \ldots, x_k) \\ \texttt{in}\ s_2 = s_1\ \texttt{with}\ \{qid.w_1 := o_1, \ldots, qid.w_n := o_n\}\end{array}$$
where f is the footprint function for procedure *qid*. We can now propagate the value of the function to derive state s_2.
- The call is recursive. In this case, an inductive hypothesis in the goal—which is a decompilation theorem for a recursive call, by virtue of our having inducted at the outset of the proof—matches the call, and is instantiated. We can again prove the antecedent of the selected inductive hypothesis, and propagate the value of the resulting functional characterization, as in the non-recursive case.

The decompilation algorithm starts by either inducting, when the procedure for which the decompilation theorem is being proved is recursive, or not (otherwise). After applying the rewrite rules, at the end of each program path, we are left with an equality between states. The proof of this equality proceeds essentially by applying rewrite rules for normalizing states (recall that states are represented by records).

Translating specifications into goals A Guardol specification is intended to set up a computational context—a state—and then assert that a property holds in that state. In its simplest form, a specification looks like

```
spec name = begin
  var decls
  in
   code;
   check property;
  end
```

where *property* is a boolean expression. A specification declaration is processed as follows. First, suppose that execution of *code* starts normally in s_1 and ends normally in s_2, *i.e.*, assume $\mathsf{STEPS}\ \Gamma\ code\ (\mathsf{Normal}\ s_1)\ (\mathsf{Normal}\ s_2)$. We want to show that *property* holds in state s_2. This could be achieved by reasoning with the induction principle for STEPS, *i.e.*, by using the operational semantics; however, experience has shown that this approach is labor-intensive. We instead opt to formally leverage the decompilation theorem for *code*, which asserts that reasoning about the STEPS-behavior of *code* could just as well be accomplished by reasoning about function f. Thus, formally, we need to show

$$\begin{array}{l}(\texttt{let}\ (o_1, ..., o_n) = f(x_1, \ldots, x_k) \\ \texttt{in}\ s_2 = s_1\ \texttt{with}\ \{name.w_1 := o_1, \ldots, name.w_n := o_n\}) \\ \Rightarrow property\ s_2\end{array}$$

Now we have a situation where the proof is essentially about how facts about f, principally its recursion equations and induction theorem, imply the property. The original goal has been freed—by sound deductive steps—from the program state and operational semantics. The import of this, as alluded to earlier, is that a wide variety of proof tools become applicable. Interactive systems exemplified by ACL2, PVS, HOL4, and Isabelle/HOL have extensive lemma libraries and reasoning packages tailored for reasoning about recursively defined mathematical functions. SMT systems are also able to reason about such functions, via universal quantification, or by decision procedures, as we discuss in Section 4.

Setting up complex contexts The form of specification above is not powerful enough to state many properties. Quite often, a collection of constraints needs to be placed on the input variables, or on external functions. To support this, specification statements allow checks sprinkled at arbitrary points in *code*:

$$\texttt{spec}\ name = \texttt{begin}\quad locals\quad \texttt{in}\quad code[\texttt{check}\ P_1, \ldots, \texttt{check}\ P_n]\quad \texttt{end}$$

We support this with a program transformation, wherein occurrences of `check` are changed into assignments to a boolean variable. Let `V` be a boolean program variable not in *locals*. The above specification is transformed into

$$\begin{array}{l}\texttt{spec}\ name = \texttt{begin} \\ \quad locals;\quad \texttt{V} : \texttt{bool}; \\ \texttt{in} \\ \quad \texttt{V} := \texttt{true};\quad code[\texttt{V} := \texttt{V} \wedge P_1, \ldots, \texttt{V} := \texttt{V} \wedge P_n];\quad \texttt{check}(\texttt{V}); \\ \texttt{end}\end{array}$$

Thus `V` is used to accumulate the results of the checks that occur throughout the code. Every property P_i is checked in the state holding just before the occurrence of $\texttt{check}(P_i)$, and all the checks must hold. This gives a flexible and concise way to express properties of programs, without embedding assertions in the source code of the program.

Recall the Guard_Correct specification. Roughly, it says *If running the guard succeeds, then running* Guard_Stable *on the result returns* true. Applying the decompiler to the code of the specification and using the resulting theorem to map from the operational semantics to the functional interpretation, we obtain the goal

$$\left(\begin{array}{l}(\forall m.\ \mathsf{DWO\_Idempotent}\ ext\ m) \wedge \\ \mathsf{Guard}\ ext\ t = \mathsf{TreeResult\_TreeOK}\ t'\end{array}\right) \Rightarrow \mathsf{Guard\_Stable}\ ext\ t'$$

which has the form required by our SMT prover, namely that the catamorphism Guard_Stable is applied to the result of calling Guard. However, an SMT prover may still not prove this goal, since the following steps need to be made: (1) inducting on the recursion structure of Guard, (2) expanding (once) the definition of Guard, (3) making higher order functions into first order, and (4) elimination of universal quantification. [6]

To address the first two problems, we induct with the induction theorem for Guard, which is automatically proved by HOL4, and expand the definition of Guard one step in the resulting inductive case. Thus we stop short of using the inductive hypotheses! The SMT solver will do that. The elimination of higher order functions is simple in the case of Guardol since the function arguments (*ext* in this case) are essentially fixed constants whose behavior is constrained by hypotheses. This leaves the elimination of the universals; only the quantification on m in $\forall m.$ DWO_Idempotent *ext* m is problematic. We find all arguments of applications of ext.DWO in the body of Guard, and instantiate m to all of them (there's only one in this case), adding all instantiations as hypotheses to the goal.

4 Verification Condition Solving using SMT

The formulas generated as verification conditions from the previous section pose a fundamental research challenge: reasoning over the structure and contents of inductive datatypes We have addressed this challenge through the use of a novel, recently-developed, decision procedure called the Suter–Dotta–Kuncak (SDK) procedure [20]. This decision procedure can be integrated into an SMT solver to solve a variety of properties over recursive datatypes. It uses catamorphisms to create abstractions of the contents of tree-structured data that can then be solved using standard SMT techniques. The benefit of this decision procedure over other techniques involving quantifiers is that it is *complete* for a large class of reasoning problems involving datatypes, as described below.

Catamorphisms In many reasoning problems involving recursive datatypes, we are interested in *abstracting* the contents of the datatype. To do this, we could define a function that maps the structure of the tree into a value. This kind of function is called a catamorphism [12] or *fold* function, which 'folds up' information about the data structure into a single value. The simplest abstraction that we can perform of a data structure is to map it into a Boolean result that describes whether it is 'valid' in some way. This approach is used in the function Guard_Stable in Section 2. We could of course create different functions to summarize the tree elements. For example, a tree can be abstracted as a

[6] Some of these steps are incorporated in various SMT systems, *e.g.*, many, but not all, SMT systems heuristically instantiate quantifiers. For a discussion of SMT-style induction see [10].

number that represents the sum of all nodes, or as a tuple that describes the minimum and maximum elements within the tree. As long as the catamorphism is *sufficiently surjective* [20] and maps into a decidable theory, the procedure is theoretically *complete*. Moreover, we have found it to be *fast* in our initial experiments.

Overview of the Decision Procedure The input of the decision procedure is a formula ϕ of literals over elements of tree terms and tree abstractions ($\mathcal{L}_{\mathcal{C}}$) produced by the catamorphisms. The logic is *parametric* in the sense that we assume a datatype to be reasoned over and catamorphism used to abstract the datatype, and the existence of a decidable theory C that is the result type of the catamorphism function. The syntax of the parametric logic is depicted in Fig. 4.

The syntax of the logic ranges over datatype terms (T and S), terms of a decidable collection theory C. Tree and collection theory formulas F_T and F_C describe equalities and inequalities over terms. The collection theory describes the result of catamorphism applications. E defines terms in the element types contained within the branches of the datatypes. ϕ defines conjunctions of (restricted) formulas in the tree and collection theories. The ϕ terms are the ones solved by the SDK procedure; these can be generalized to arbitrary propositional formulas (ψ) through the use of a DPLL solver [4] which manages the other operators within the formula.

S	$::= T \mid E$	Constructors' arguments
T	$::= t \mid \mathbb{C}_j(S_1, \ldots, S_{n_j}) \mid \mathbb{S}_{j,k_\tau}(T)$	Tree terms
C	$::= c \mid \alpha(T) \mid \mathcal{T}_{\mathcal{C}}$	$\mathcal{C}$-terms
F_T	$::= T = T \mid T \neq T$	Tree (dis)equations
F_C	$::= C = C \mid \mathcal{F}_{\mathcal{C}}$	Formula of $\mathcal{L}_{\mathcal{C}}$
E	$::=$ variables of type $\mathcal{E}_k \mid \mathbb{S}_{j,k_{\mathcal{E}}}(T)$	Expression
ϕ	$::= \bigwedge F_T \wedge \bigwedge F_C$	Conjunctions
ψ	$::= \phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid$ $\phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi$	Formulas

Fig. 4. Syntax of the parametric logic

In the procedure, we have a single datatype τ with m constructors. The j-th constructor ($1 \leq j \leq m$), $\mathbb{C}_j$, has n_j arguments ($n_j \geq 0$), whose types are either τ or $\mathcal{E}$, an element type. For each constructor $\mathbb{C}_j$, we have a list of selectors $\mathbb{S}_{j,k}$ ($1 \leq k \leq n_j$), which extracts the k-th argument of $\mathbb{C}_j$. For type safety, we may put the type of the argument to be extracted as a subscript of its selector. That is, each selector may be presented as either $\mathbb{S}_{j,k_\tau}$ or $\mathbb{S}_{j,k_{\mathcal{E}}}$. The decision procedure is parameterized by $\mathcal{E}$, a collection type $\mathcal{C}$, and a catamorphism function $\alpha : \tau \rightarrow \mathcal{C}$. For example, the datatype MsgTree has two constructors Leaf and Node. The former has no argument while the latter has three arguments corresponding to its Value, Left, and Right. As a result, we have three selectors for Node, including Value: MsgTree → Msg, Left : MsgTree → MsgTree, and

Right : MsgTree → MsgTree. In addition, a tree can be abstracted by the catamorphism Guard_Stable : MsgTree → bool to a boolean value. In the example, $\mathcal{E}$, $\mathcal{C}$, and α are Msg, bool, and Guard_Stable, respectively.

Implementing Suter-Dotta-Kuncak in OpenSMT We have created an implementation of the SDK decision procedure. As described earlier, the decision procedure operates over conjunctions of theory literals and generates a problem in a *base theory* (denoted C) that must be solved by another existing decision procedure. Thus, the decision procedure is not useful by itself; it must be integrated into an SMT solver that supports reasoning over complex Boolean logic formulas (rather than only conjunctions) and contains the theories necessary to solve the terms in C.

In DPLL [4], theory terms are partitioned by solver: each solver "owns" the terms in its theory. The SMT solver partitions the terms for each theory in a *purification* step. However, the SDK decision procedure as presented in [20] requires an unusual level of supervisory control over other other decision procedures within an SMT solver. That is, terms from the element theories, the collections theory, and the tree theory have to be provided to the SDK procedure where they are manipulated (possibly adding disjunctions) and eventually passed back to a SMT solver to be further processed. Thus, to implement SDK, we have re-architected a standard DPLL solver by forwarding all theory terms to the SDK procedure, letting it perform purification and formula manipulation, and passing the resulting problem instance (which no longer contains datatype terms) into an inner instance of the SMT solver to solve the (local) subproblem and return the result to the outer instance. This architecture is shown in Figure 5.

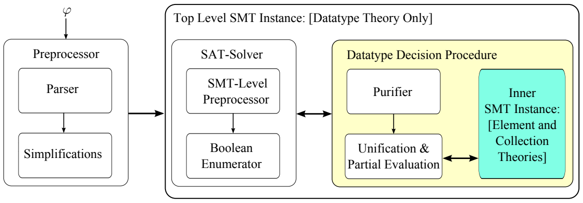
φ
Preprocessor
Parser
Simplifications
Top Level SMT Instance: [Datatype Theory Only]
SAT-Solver
SMT-Level Preprocessor
Boolean Enumerator
Datatype Decision Procedure
Purifier
Unification & Partial Evaluation
Inner SMT Instance: [Element and Collection Theories]

Fig. 5. Architecture for SMT solver containing SDK

We have chosen to implement SDK on OpenSMT as it supports a sufficiently rich set of theories and is freely available for extension and commercial use. In addition, OpenSMT is fast and has a simple interface between decision procedures and the underlying solver. We have also made a handful of extensions to the tool to support reasoning over mutually recursive datatypes. A drawback of the current release of OpenSMT is that it does not support quantifiers; on the other hand, quantifiers add a source of incompleteness to the solving process that we are trying to avoid.

4.1 Experimental Results

To test our approach, we have developed a handful of small benchmark guard examples. For timings, we focus on the SMT solver which performs the interesting part of the proof search. The results are shown in Table 2, where the last guard is the running example from the paper. In our early experience, the SDK procedure allows a wide variety of interesting properties to be expressed and our initial timing experiments have been very encouraging, despite the relatively näive implementation of SDK, which we will optimize in the near future. For a point of comparison, we provide a translation of the problems to Microsoft's Z3 in which we use universal quantifiers to describe the catamorphism behavior. This approach is incomplete, so properties that are falsifiable (i.e., return SAT) often do not terminate (we mark this as 'unknown'). A better comparison would be to run our implementation against the Leon tool suite developed at EFPL [21]. Unfortunately, it is not currently possible to do so as the Leon tool suite operates directly over the Scala language rather than at the SMT level. All benchmarks were run on Windows 7 using an Intel Core I3 running at 2.13 GHz.

Table 2. Experimental results on guard examples

Test #	OpenSMT-SDK	Z3
Guard 1	5 sat (0.83s) / 1 unsat (0.1s)	5 unknown / 1 unsat (0.06 s)
Guard 2	3 unsat (0.28s)	1 unknown / 2 unsat (0.11 s)
Guard 3	3 sat (0.33s) / 4 unsat (0.46s)	1 unknown / 2 sat (0.17s) / 4 unsat (0.42 s)
DWO	1 unsat (0.34s)	1 unsat (0.11s)

5 Discussion

As a programming language with built-in verification support, Guardol seems amenable to being embedded in an IVL (Intermediate Verification Language) such as Boogie [11] or Why [3]. However, our basic aims would not be met in such a setting. The operational semantics in Section 3.1 defines Guardol program execution, which is the basis for verification. We want a strong formal connection between a program plus specifications, both expressed using that semantics, and the resulting SMT goals, which do not mention that semantics. The decompilation algorithm achieves this connection via machine-checked proof in higher order logic. This approach is simpler in some ways than an IVL, where there are two translations: one from source language to IVL and one from IVL to SMT goals. To our knowledge, IVL translations are not machine-checked, which is problematic for our applications. Our emphasis on formal models and deductive transformations should help Guardol programs pass the stringent certification requirements imposed on high-assurance guards.

Higher order logic plays a central role in our design. HOL4 implements a foundational semantic setting in which the following are formalized: program ASTs, operational semantics, footprint functions (along with their termination proofs and induction theorems), and decompilation theorems. Decompilation extensively uses higher order features when composing footprint functions corresponding to sub-programs. Moreover, the backend verification theories of the SMT system already exist in HOL4. This offers the possibility of doing SMT proof reconstruction [1] in order to obtain higher levels of assurance. Another consequence is that, should a proof fail or take too long in the SMT system, it could be performed interactively.

As future work we plan to investigate both language and verification aspects. For example, the current Guardol language could be made more user-friendly. It is essentially a monomorphic version of ML with second order functions, owing to Guardol's external function declarations. It might be worthwhile to support polymorphic types so that the repeated declarations of instances of polymorphic types, *e.g.*, option types and list types, can be curtailed. Additionally, programs could be considerably more terse if exceptions were in the language, since explicitly threading the error monad wouldn't be needed to describe guard failures.

An interesting issue concerns specifications: guard specifications can be about *intensional* aspects of a computation, *e.g.*, its structure or sequencing, as well as its result. For example, one may want to check that data fuzzing operations always occur before encryption. However, our current framework, which translates programs to *extensional* functions, will not be able to use SMT reasoning on intensional properties. Information flow properties [5] are also intensional, since in that setting one is not only concerned with the value of a variable, but also whether particular inputs and code structures were used to produce it. Techniques similar to those in [22] could be used to annotate programs in order to allow SMT backends to reason about intensional guard properties.

Planned verification improvements include integration of string solvers and termination deferral. In the translation to SMT, strings are currently treated as an uninterpreted type and string operations as uninterpreted functions. Therefore, the system cannot reason in a complete way about guards where string manipulation is integral to correctness. We plan to integrate a string reasoner (*e.g.*, [8]) into our system to handle this problem. Finally, a weak point in the end-to-end automation of Guardol verification is termination proofs. If the footprint function for a program happens to be recursive, its termination proof may well fail, thus stopping processing. There are known techniques for defining partial functions [6, 9], obtaining recursion equations and induction theorems constrained by termination requirements. These techniques remove this flaw, allowing the deferral of termination arguments while the partial correctness proof is addressed.

Acknowledgements The TACAS reviewers did a well-informed and thorough job, kindly pointing out many mistakes and infelicities in the orginal submission.

References

1. S. Böhme, A. Fox, T. Sewell, and T. Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In *Proceedings of Certified Programs and Proofs*, volume 7086 of *LNCS*. Springer, 2011.

2. R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT solver. In *Proceedings of TACAS*, volume 6015 of *LNCS*, 2010.
3. J-C. Filliâtre. *Deductive Program Verification.* Thèse d'habilitation, Université Paris 11, December 2011.
4. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proceedings of CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
5. J. Goguen and J. Meseguer. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
6. D. Greve. Assuming termination. In *Proceedings of ACL2 Workshop*, ACL2 '09, pages 114–122. ACM, 2009.
7. Rockwell Collins Inc. Turnstile High Assurance Guard Homepage. http://www.rockwellcollins.com/sitecore/content/Data/Products/Information_Assurance/Cross_Domain_Solutions/Turnstile_High_Assurance_Guard.aspx.
8. A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, and M. Ernst. HAMPI: A solver for string constraints. In *Proceedings of ISSTA*, 2009.
9. A. Krauss. *Automating recursive definitions and termination proofs in higher order logic.* PhD thesis, TU Munich, 2009.
10. K. R. Leino. Automating induction with an SMT solver. In *Proceedings of VMCAI*, volume 7148 of *LNCS*. Springer, 2012.
11. K. R. Leino and P. Ruemmer. A polymorphic intermediate verification language: Design and logical encoding. In *Proceedings of TACAS*, volume 6015 of *LNCS*, 2010.
12. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *Proceedings of FPCA*, volume 523 of *LNCS*, 1991.
13. S. Miller, M. Whalen, and D. Cofer. Software model checking takes off. *CACM*, 53:58–64, February 2010.
14. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
15. M. Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, 2009.
16. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.
17. N. Schirmer. *Verification of sequential imperative programs in Isabelle/HOL.* PhD thesis, TU Munich, 2006.
18. P. Sestoft. ML pattern match compilation and partial evaluation. In *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 446–464, 1996.
19. K. Slind and M. Norrish. A brief overview of HOL4. In *Proceedings of TPHOLs*, volume 5170 of *LNCS*, pages 28–32, 2008.
20. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *Proceedings of POPL*, pages 199–210. ACM, 2010.
21. P. Suter, A. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In Eran Yahav, editor, *Proceedings of Static Analysis*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.
22. M. Whalen, D. Greve, and L. Wagner. Model checking information flow. In David Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications.* Springer, 2010.