Hybrid Contract Checking via Symbolic Simplification

Dana N. Xu
INRIA Paris-Rocquencourt
na.xu@inria.fr

Abstract
Program errors are hard to detect or prove absent. Allowing programmers to write formal and precise specifications, especially in the form of contracts, is a popular approach to program verification and error discovery. We formalize and implement a hybrid (static and dynamic) contract checker for a subset of OCaml. The key technique is symbolic simplification, which makes integrating static and dynamic contract checking easy and effective. Our technique statically checks contract satisfaction or blames the function violating the contract. When a contract satisfaction is undecidable, it leaves residual code for dynamic contract checking.

Categories and Subject Descriptors D.3 [Software]: Programming Languages

General Terms functional language, verification, debugging

Keywords contract semantics, static, dynamic, hybrid, contract checking, symbolic simplification

1. Introduction
Constructing reliable software is difficult. Formulating and checking (statically or dynamically) logical assertions [2, 5, 15, 17, 36], especially in the form of contracts [7, 12, 13, 29, 40], is one popular approach to error discovery. Static contract checking can catch all contract violations but may raise false alarms and can only check restricted properties; dynamic checking can check more expressive properties but consumes run-time cycles and only checks the paths actually executed, and so is not complete. Consider an OCaml program augmented with a contract declaration:

```ml
(* val f1 : (int -> int) -> int *)
contract f1 = ({x | x >= 0} -> {y | y >= 0})
-> {z | z >= 0}
let f1 g = (g 1) - 1
let f2 = f1 (fun x -> x - 1)
```

The contract of f1 says that f1 will return a non-negative number whenever it is applied to a function that returns a non-negative number when given a non-negative number. Both a static checker and a dynamic checker are able to report that f1 fails its postcondition: a static checker relies on the unsoundness of ∀g : int → int, (g 1) ≥ 0 ⇒ (g 1) − 1 ≥ 0 while a dynamic checker evaluates (((fun x -> x - 1) 1) - 1) to -1, which violates the contract {z | z >= 0}. However, a dynamic checker cannot tell that the argument (fun x -> x - 1) fails f1’s precondition because there is no witness at run-time, while a static checker can report this contract violation because ∀x : int, x ≥ 0 ⇒ x − 1 ≥ 0 does not hold. On the other hand, a static checker usually gives three outcomes: (a) definitely no bug; (b) definitely a bug; (c) possibly a bug. Here, a bug refers to a contract violation. As static and dynamic checking can be complementary, we may want to invoke a dynamic checker when the outcome is (c). This ensures that no contract violations can escape while maintaining expressiveness.

Following the formalization in [40], but this time for a strict language, we first give a denotational semantics for contract satisfaction, i.e., we define what it means for an expression e to satisfy its contract t (written e ∈ t) without knowing how to check it. Next, we define a wrapper t that takes e and t and produces a term e ⊲ t with contract checks inserted at appropriate places in e. If a contract check is violated, a special constructor BAD signals the violation where the label l precisely captures the function at fault. All we have to do is to check the reachability of BAD in the term e ⊲ t. We symbolically simplify the term e ⊲ t, aiming to simplify Bads away. If some Bads constructors remain, we either report it as a compile-time error or leave the residual code for dynamic checking. We make the following contributions:

- We clarify the relationship between static contract checking and dynamic contract checking ([2]). A new observation is that, after static checking, we should prune away some more unreachable code before going to dynamic checking. Such unreachable code, however, is essential during static checking. We show the correctness of this pruning ([6]) with the telescoping property studied (but not used for such purpose) in [7, 40].

- We define e ∈ t and e ⊲ t and prove a theorem “e ⊲ t is crash-free ⇐⇒ e ∈ t” ([4]). “Crash-free” means BAD is not reachable under any context. Such a formalization is tricky and its correctness proof is non-trivial. We rework the proofs from [41] for a strict language.

- We design a novel SL machine that augments symbolic simplification with contextual information synthesis for checking the reachability of BAD statically ([5]). The checking is automatic and modular and we prove its soundness. Moreover, the SL machine produces residual code for dynamic checking.

- We design a logicization technique that transforms expressions to logical formulae. The key contribution is to deal with non-total terms ([5]).

2. Overview
Assertions [17] state logical properties of an execution state at arbitrary points in a program; contracts specify agreements concerning the values that flow across a boundary between distinct parts of a program (modules, procedures, functions, classes). If an agreement is violated, contract checking is supposed to provide precise...
blaming of the function at fault. Contracts [29] and higher-order contracts [12] were to be checked at run-time when they were first introduced. To perform dynamic contract checking (DCC), a function must be called to be checked. For example:

\[
\text{contract inc = } \{x \mid x > 0\} \rightarrow \{y \mid y > 0\}
\]

\[
\text{let inc = fun } v \rightarrow v + 1
\]

\[
\text{let hl = inc 0}
\]

A dynamic checker wraps the inc in hl with its contract \( t_{\text{inc}} \) (a shorthand for the contract of inc):

\[
\text{let hl = (inc } \text{bad } \text{if } \text{hl} = \text{inc } \text{bad } \text{if } t_{\text{inc}} 0)
\]

where \( f \) is \((2, 5, "inc")\) the (row,col) source location where inc is defined and \( f \) is \((3, 10, "hl")\) the source location of the call site with caller’s name. This wrapped hl expands to:

\[
(\lambda x, \lambda y = \text{inc} (\lambda x = x + 1) \text{if } x > 0 \text{then } x \text{else BAD}(3, 10, "hl"))
\]

In the upper box, the argument of inc is guarded by the check \( x > 0 \); in the lower box, the result of inc is guarded by the check \( y > 0 \). If a check succeeds, the original term is returned, otherwise, the special constructor BAD is reached and blame is raised. In this case, hl calls inc with 0, which fails inc’s precondition. Running the above wrapped code, we get BAD\((3,10, "hl")\), which blames hl.

With the DCC algorithm, given a function \( f \) and a contract \( t \), to check that the callee \( f \) and its caller agree on the contract \( t \) dynamically, a checker wraps each call to \( f \) with its contract:

\[
\text{let } f = \text{bad } \text{if } \text{hl} = \text{bad } \text{if } t_{\text{inc}} 0
\]

, which behaves the same as \( f \) except that (a) if \( f \) disobeys \( t \), it blames \( f \), signaled by BAD; (b) if the contract uses \( f \) in a way not permitted by \( t \), it blames the caller of \( f \), signaled by BAD where "?" is filled with a caller name and the call site.

Later, [7, 40] give formal declarative semantics for contract satisfaction that not only allow us to prove the correctness of DCC against this semantics, but also to check contracts statically.

The essence of static contract checking (SSC) is:

splitting \( \text{bad } \text{if } \text{hl} = \text{bad } \text{if } t_{\text{inc}} 0 \) into two halves: \( \text{bad } \text{if } \text{hl} = \text{bad } \text{if } t_{\text{inc}} t \) and \( \text{bad } \text{if } \text{hl} = \text{bad } \text{if } t_{\text{inc}} t \)

The \( \Rightarrow \) ("ensures") and the \( \Leftarrow \) ("requires") are dual to each other. The special constructor UNR ("unreachable"), does not raise blame, but stops execution. (Those who are familiar with assert and assume can think of \((\text{if } \text{p then } \text{e UNR})\) as \((\text{assert } \text{p}; \text{e})\) and \((\text{if } \text{p then } \text{e UNR})\) as \((\text{assume } \text{p}; \text{e})\).)

SSC is modular and is performed at the definition site of each function. For example, \((\lambda v. v + 1) \Rightarrow t_{\text{f3}}\) expands to:

\[
\text{\lambda x_1. let } y = (\lambda v. v + 1) \text{if } x > 0 \text{then } x \text{else UNR'} \text{in if } y > 0 \text{then } y \text{else BAD}(2, 5, "inc")
\]

At the definition site of a function, \( f = e \), we assume \( f \)'s precondition and assert its postcondition. If all BADs in \( f \Rightarrow t \) are not reachable, we know \( f \) satisfies its contract \( t \). One way to check reachability of BAD is to symbolically simplify the fragment. In the above case, inlining gives:

\[
\text{\lambda x_1. let } y = (\lambda v. v + 1) \text{if } x > 0 \text{then } x \text{else UNR'} \text{in if } y > 0 \text{then } y \text{else BAD}(2, 5, "inc")
\]

In this paper, besides symbolic simplification, we collect contextual information in logical formula form and consult an SMT solver to check the reachability of BAD. An SMT solver usually deals with formulae in first-order logic (FOL). In this section, we present formulae in higher-order logic while §5 gives the details of the generation of formulae in FOL. For the two subexpressions of the RHS of \( y \), we have:

\[
\exists x_2, (\nu v, x_2(v) = v + 1) \Rightarrow (\exists x_3, (x_1 > 0 \Rightarrow x_3 = x_1) \\land (\neg (x_1 = 1)) \Rightarrow \text{BAD}((7, 10, "f")\})
\]

One can think of the existentially quantified \( x_2 \) (and \( x_3 \)) as denoting the expression itself. For the RHS of \( y \), we have:

\[
\forall y, \exists x_2, (\nu v, x_2(v) = v + 1) \land (\exists x_3, (x_1 > 0 \Rightarrow x_3 = x_1) \\land (\neg (x_1 = 1)) \Rightarrow \text{BAD}((7, 10, "f")\})
\]

We check the validity of a formula collected from the path to \( \text{BAD}(2, 5, "inc")\), i.e., \( \exists x_1, Q_1 \Rightarrow y > 0, \) by consulting an SMT solver. Since it is valid, we know that the BAD\((2,5, "inc")\) is not reachable, thus inc satisfies its contract.

Consider the function \( f_1 \) and its contract \( t_{f_1} \) in §1. So \( f_1 \Rightarrow t_{f_1} \)

is \((\lambda g, (g - 1) \Rightarrow t_{f_1})\), which expands to:

\[
\text{\lambda x_1. let } z = (\lambda g. (g - 1)) \text{if } x > 0 \text{then } y - 1 \text{else UNR'} \text{in if } z > 0 \text{then } z \text{else BAD}(4, 5, "f")
\]

After applying some conventional simplification rules, we have:

\[
R1: \ \text{\lambda x_1. let } z = (\lambda g. (g - 1)) \text{if } x > 0 \text{then } y - 1 \text{else UNR'} \text{in if } z > 0 \text{then } z \text{else BAD}(4, 5, "f")
\]

We see that the inner BAD\((4,5, "f")\) has been simplified away, because \( z = x_2 = 1 \) and \((\text{if } x > 0 \text{then } \text{BAD}(4, 5, "f")\) is simplified to \( z \). As we cannot prove \( \forall y, \forall x_2, (\exists x_3, (x_1 > 0 \Rightarrow x_3 = x_1) \\land (\neg (x_1 = 1)) \Rightarrow \text{BAD}(4, 5, "f")\} \)

We can either report this potential contract violation at compile-time or leave this residual code \( R1 \) for DCC to achieve hybrid checking.

Hybrid contract checking (HCC) performs SSC first and runs the residual code as in DCC. In SSC, \( f_1 \Rightarrow t_{f_1} \) checks whether \( f_1 \) satisfies its postcondition by assuming its precondition holds. At each call site of \( f_1 \), we wrap the function with \( \Leftarrow \). For example:

\[
\text{let } f_3 = f_1 \text{unr}
\]

where unr is a difficult function for an SMT solver and zut’s contract is \( \{x \mid x \text{ true}\} \). Suppose \( \text{zut } \Leftarrow \{x \mid x \text{ true}\} = \text{zut} \), then we have then the term \( f_3 \Rightarrow t_{f_3} \) to be:

\[
(\lambda v. v + 1) \Rightarrow (f_1 \Rightarrow t_{f_3}) \Rightarrow v \Rightarrow v > 0
\]

which requires \( f_3 \) to satisfy \( f_1 \)'s precondition and assumes \( f_1 \) satisfies its postcondition because \( f_1 \Rightarrow t_{f_1} \) has been checked. During SSC, a top-level function is never inlineled. We do not have to know its detailed implementation at its call site as it has been guarded by its contract with \( f \Rightarrow t \). The \( f_3 \Rightarrow t_{f_3} \) expands to:

\[
\text{let } v = 
\text{let } z = f_1
\]

\[
(\lambda x_2. \text{let } y = \text{zut (let } x = x_2 \text{in if } x > 0 \text{then } \text{zut (\nu v, x_2(v) = v + 1) \Rightarrow (\exists x_3, (x_1 > 0 \Rightarrow x_3 = x_1) \\land (\neg (x_1 = 1)) \Rightarrow \text{BAD}((7, 10, "f")\}) in if } y > 0 \text{then } y \text{else BAD}(7, 10, "f")\}) in if } y > 0 \text{then } y \text{else BAD}(7, 10, "f")\}) \text{in if } y > 0 \text{then } y \text{else BAD}(7, 10, "f")\}) \text{in if } y > 0 \text{then } y \text{else BAD}(7, 10, "f")\})
\]
As \( a \) is dual to \( b \), the RHS of \( v \) is actually a copy of the earlier \( f1 \mapsto f4 \) but swapping the BAD and UNR and substituting \( x1 \) with \( z \). We now know the source location of the call site of \( f1 \) and its caller’s name, the UNR becomes BAD\((4,5,f^{−1}1)\) and the BAD\((4,5,f^{−1}1)\) becomes UNR\((7,10,f^{−1}1)\). At definition site where the caller is unknown, we use the location of \( f1 \), i.e., \((4,5, "f^{1}1")\). Once its caller is known, we use \((7,10, "f^{1}1")\). It is easy to get source location so we do not elaborate it further.

As an SMT solver says valid for \( \forall v. (2z \geq 0 \land v = z) \Rightarrow v \geq 0 \), the \( f3 \mapsto f3 \) can be simplified to (say R2):

\[
\text{let } z = f1 (x z). \text{ let } y = z u t \text{ (let } x = x z \text{ in) if } x > 0 \text{ then (x else UNR\((7,10,f^{−1}1)\) in if } y > 0 \text{ then } y \text{ else BAD\((7,10,f^{−1}1)\) in} if z \geq 0 \text{ then } z \text{ else UNR\((7,10,f^{−1}1)\)}
\]

leaving one BAD. We can either report this potential contract violation at compile-time or continue to check. For SCC, we have checked \( f1 \mapsto f4 \), but for DCC, to invoke \( f1 \mapsto f4 \), we must use the residual code R1. However, the UNR clauses are useful for SCC, but redundant for DCC. We can remove \( f1 \mapsto f4 \) with a simplification rule:

\[
\text{if } e_0 \text{ then } e_1 \text{ else UNR} \Rightarrow e_1 \tag{rmUNR}
\]

(We shall explain why it is valid to apply this rule even if \( e_0 \) may diverge or crash in §6. Intuitively, UNR is indeed unreachable and \( e_0 \) has been checked before this program point.) Applying the rule [rmUNR] to R1 and R2, and simplifying a bit, we get:

\[
\text{let } z = \text{let } y = (y - 1) \text{ in (x \mapsto x) in if } y \geq 0 \text{ then } y \text{ else BAD\((4,5,f^{−1}1)\) in if } y > 0 \text{ then } y \text{ else BAD\((7,10,f^{−1}1)\)}
\]

respectively, which is the residual code being run. We show in §6 that HCC blames a function \( f1 \); if \( f1 \) CCS blames \( f1 \).

**Summary**

Given a definition \( f = e \) and a contract \( t \), to check that \( e \) satisfies \( t \) (written \( e \in t \)), we perform these steps. (1) Construct \( e \mapsto t \). (2) Simplify \( e \mapsto t \) as much as possible to \( e' \), consulting an SMT solver when necessary. (3) If no bad is in \( e' \), then there is no contract violation, while if there is a BAD in \( e' \), we give error (or warning) message for a definite (or potential) bug at compile-time. (4) For a function \( f \) not satisfying its contract, create its residual code \( f' \) by simplifying \( e' \) with the rule [rmUNR], and run the program with each \( f \) being replaced by \( f' \).

3. **The language**

The language presented in this paper, named M, is pure and strict, and is a subset of OCaml with parametric polymorphism.

3.1 **Syntax**

Figure 1 gives the syntax of language M. A program contains a set of data type declarations, contract declarations and function definitions. Expressions include integers \( n \), variables, lambda abstractions, applications, constructors and match expressions. We have top-level \( \text{let rec} \) but for the ease of presentation, we omit local \( \text{let rec} \). (It is possible to allow global \( \text{let rec} \) by assuming that a local recursive function is given a contract or using contract inference [21] to infer its contract. Even if [21] is not modular, it is enough to infer a contract for a local function.) Pairs are a special case of constructed terms. A local \( \text{let} \) expression \( let \ x = e \ in \ t \) is syntactic sugar for \( (\lambda x. e) \ t \). An if expression if \( e_0 \) then \( e_1 \) else \( e_2 \) is syntactic sugar for \( \text{match } e_0 \text{ with } \{ \text{true } \mapsto e_1; \text{false } \mapsto e_2 \} \).

We assume all top-level functions are given a contract. Contract checking is done after the type checking phase in a compiler so we assume all expressions, contexts, and contracts are well-typed and use the type information (presented as a superscript, e.g., \( e^c \) for \( t^c \)) whenever necessary. Type-checking material is in [39].

The two contract exceptions (also called blames) BAD and UNR are adapted from [40]. They are for internal usage, and are not visible to programmers. The label \( l \) captures source location and function name, which are useful for error reporting as well as for the examination of the correctness of blaming. But we may omit the label \( l \) when it is not the focus of the discussion.

It is possible for programmers to write:

\[
\text{let head xs = match xs with} \tag{raise Error}
| [] \rightarrow \text{raise Error}
| x : \mathbf{l} \rightarrow x
\]

where \( \text{raise} \) : \( \forall a \). Exception \( \rightarrow a \). The Error has type Exception, which is a built-in data type for exceptions. As we do not have try-with in language M (leaving it as future work), a preprocessing step converts \( \text{raise} \) Error to BAD\(\text{header} \).

We have four forms of contracts. The \( p \) in a predicate contract \( \{ x \mid p \} \) refers to a boolean expression in the same language M. Dependent function contracts allow us to describe dependency between input and output of a function. For example, \( \{ y \mid y > 0 \} \rightarrow \{ x \mid z > 0 \} \) says that the input is greater than 0 and the output is greater than the input. We can use a shorthand \( \{ x \mid x > 0 \} \rightarrow \{ z \mid z > 0 \} \) by assuming \( x \) scopes over the RHS of \( \rightarrow \). The \( \rightarrow \) is right associative. Similarly, dependent tuple contracts allow us to describe dependency between two components of a tuple. For example, \( \{(y \mid y > 0), \{ z \mid z > x \} \} \) has short hand \( \{(x \mid x > 0), \{ z \mid z > x \} \} \). Contract Any is a universal contract that any expression satisfies. We support higher-order contracts, e.g., \( k : \{(x \mid x > 0), \{ y \mid y > x \} \} \rightarrow \{ z \mid k 5 > z \} \) for a function let \( f \ g = g 2 \).
Definition 1 (Semantically Equivalent). Two expressions $e_1$ and $e_2$ are semantically equivalent, namely $e_1 \equiv e_2$, iff $\forall C, (C[e_1])_{\text{bool}}^\top \rightarrow \top \leftrightarrow (C[e_2])_{\text{top}}^\top \rightarrow \top$ for $i = 1, 2, r \in \{\text{BAD}, \text{UNR}\}$.

We use BAD to signal that something has gone wrong in a program, which can be a program failure or a contract violation.

Definition 2 (Crash). A closed term $e$ crashes iff $e \rightarrow^* \text{BAD}$.

Our framework only guarantees partial correctness. A diverging program does not crash.

Definition 3 (Diverges). A closed expression $e$ diverges, written $e \uparrow$, iff either $e \rightarrow^* \text{UNR}$ or there is no value val such that $e \rightarrow^* \text{val}$.

At compile-time, one decidable way to check the safety of a program is to see whether the program is syntactically safe.

Definition 4 (Syntactic safety). A (possibly open) expression $e$ is syntactically safe iff BAD $\not\vdash e$. Similarly, a context $C$ is syntactically safe iff BAD $\not\vdash C$.

The notation BAD $\not\vdash e$, means BAD does not syntactically appear anywhere in $e$, similarly for BAD $\not\vdash C$. For example, $\lambda x.e$ is syntactically safe, while $\lambda x.(\text{BAD}, x)$ is not.

Definition 5 (Crash-free Expression). A (possibly open) expression $e$ is crash-free iff $\not\vdash C$. BAD $\not\vdash C$ and $(C[e])_{\text{bool}}^\top \rightarrow C[e]_{\text{top}}^\top \rightarrow^* \text{BAD}$.

The quantified context $C$ serves the usual role of a probe that tries to invoke $e$ into crashing. A crash-free expression may not be syntactically safe, e.g., $\lambda x.\text{if } x * x \geq 0 \text{ then } x + 1 \text{ else } \text{BAD}$. (defined in Figure 4) expands to a particular expression, which behaves the same as $e$ except that it raises blame $r_l$ if $e$ does not obey $l$ and raises $r_l$ if the wrapped term is used in a way that violates $l$.

Lemma 1 (Syntactically safe expression is crash-free).

For ease of presentation, when we do not give label $l$ to BAD or UNR, we mean BAD or UNR for any $l$. Moreover, expressions BAD and UNR are closed expressions even if $l$ is not explicitly bound.

4. Contracts

Inspired by [40], we design a contract satisfaction and checking algorithm for a strict language. As diverging contracts make dynamic contract checking unsound (explained in §4.2) and we do hybrid checking, we focus on total contracts.
Basically, we rework the proofs in [41] for a strict language. Then expressions which diverge whenever applied because of the simplification work is to combine symbolic

We want a divergent contract hides crashes in [4, 27, 35] to build an efficient automatic termination checker. Unlike [12], which assumes there are no exceptions in contracts, we can check the reachability of BAD in a contract; i.e., the correct labels for contract violation. We introduce an SL machine (Figure 5) which

Unlike [12], which assumes there are no exceptions in contracts, any

The superscript \( \tau \) says both \( e \) and \( t \) are well-typed and have the same type \( \tau \). Note that if \( t \) is terminating and \( e \cdot t \) is crash-free, then \( t \) is total. See [39] for a full proof and a completeness theorem. Basically, we rework the proofs in [41] for a strict language.

Unlike [12], which assumes there are no exceptions in contracts, checking our algorithm detects contract exceptions in contracts. The term \( t_2((x_1)_{\overline{r_1}} t_1/x_1) \) in [P2] and [P3] says that, each (function) call in a contract is wrapped with its contract so that, if there is any contract violation in a contract, we report this error. For example:

We only have to prove termination of functions used in contracts, not all the functions in a program. We can adapt ideas in [4, 27, 35] to build an efficient automatic termination checker.

5. Static contract checking and residualization

Theorem 1 (Soundness of contract checking). For all closed expressions \( e \) and closed, terminating contracts \( t' \).

\[
(e \cdot t') \text{ is crash-free } \Rightarrow e \cdot t \text{ is crash-free}
\]

The job of the SL machine is to simplify an expression as much as possible, consulting the logical store when necessary; when it cannot simplify the expression further, it rebuilds the expression.

5.1 The SL machine

In Figure 5, the constant \( n \) and blame \( r \) cannot be simplified further, thus being rebuilt as shown in [S-const] and [S-enx] respectively. One might ask why we rebuild the residual code with undischarged blames to a dynamic checker.

As we perform symbolic simplification rather than evaluation (as for the CEK machine [14]), we only put a variable in the environment \( H \) if it denotes a trivial value. A variable denoting a top-level function is not put in \( H \). Variables in \( H \) are inlined by [S-var1] while variables not in \( H \) are rebuilt by [S-var2].

Each element on the stack is called a stack frame where the hole in a stack frame refers to the expression under simplification or being rebuilt. We use a to represent an expression that has been simplified. The syntax of \( S \) is

\[
S ::= | | (\bullet) :: S | \bullet :: S | (\lambda x. \bullet) :: S | let x = \bullet \in e :: S | (match * with alt) :: S | (let x = \bullet \in e) :: S |
\]

The transitions [S-app], [S-match] and [S-K] implement the conventional simplification rules. Here, \( \overline{p} \) abbreviates a sequence of \( x_1, \ldots, x_n \). We use \( \lambda t \) instead of lambda for easy reading. Rules [S-letL] and [S-matchL] push the argument into the let-body and match-body respectively; rules [S-letR] and [S-matchR] push the function into the let-body and match-body. The rules [S-m-match] and [S-match-alt] are to make an expression less nested. Rule [S-K-match] allows us to simplify an expression like match \( \bullet \) with \( \{ \text{Some } e \rightarrow 5; \text{None } \rightarrow \text{BAD} \} \) to \( 1 \text{let } x = e \in 5 \) which is crash-free.

What does rebuild do? It unwinds the stack. If the stack is empty ([R-done]), indicating the end of the whole simplification process, we return the expression. Otherwise, we examine the stack frame. By [E-enx], the transition \([R-R] \) rebuilds UNR (or BAD) with the rest of the stack. After we finish simplifying one subexpression, we start to simplify the next subexpression (e.g., [R-run]). When all subexpressions are simplified, we rebuild the expression (e.g., [R-lam] and [R-enx]). If current simplified expression is a trivial value and we have stack frame lambda on \( S \), we use [R-beta]; together with [S-var1], they implement a beta-reduction [E-beta]. Bound variables are renamed when necessary.
The logical store $\mathcal{L}$ captures all the ctx-info up to the program point being simplified. (We use if expression to save space, but refer to match-transitions.) Consider:

$$\langle \mathcal{H} | a | \lambda x \cdot (\forall x > 0) \leadsto (\forall x > 0) \text{ then 5 else BAD} \rangle \text{ else UNR} \quad \langle \ldots \rangle$$

The [S-lam] puts $\forall x : \text{int} \in \mathcal{L}$, which is initially empty:

$$\langle \ldots \rangle \text{ if } x > 0 \text{ then } (\ldots) \text{ then 5 else BAD} \text{ else UNR} \quad \langle \ldots \rangle$$

The [S-match] starts to simplify the scrutinee $x > 0$, which is being rebuilt after a few trivial steps.

$$(\langle \ldots \rangle \text{ if } x > 0 \text{ then } (\ldots) \text{ then 5 else BAD} \rangle \text{ else UNR} : \langle \ldots \rangle)$$

Before applying the transition [R-s-save], we check whether $x > 0$ or $\neg(x > 0)$ is implied by $\mathcal{L}$ to see whether the transition [R-s-match] can be applied. The transition [R-s-match] implements [E-match], where the side condition “if $\exists(K \not\Rightarrow a)$, $\mathcal{L} \Rightarrow \langle a | (\forall x \not\Rightarrow a) \rangle” checks if there is any branch $K \not\Rightarrow a$ that matches the scrutinee $a$. But the current information in $\mathcal{L}$ is not enough to show the validity of either $x > 0$ or $\neg(x > 0)$. By [R-s-save], we create this scrutinee to logical formula with $[a]_K(\forall x \not\Rightarrow a)$ (explained later) and put it in $\mathcal{L}$ and simplify both branches. Note that we put $x > 0$ in $\mathcal{L}$ for the true branch while $\neg(x > 0)$ for the false branch.

$$\langle \ldots \rangle \text{ if } x > 0 \text{ then } (\ldots) \text{ then 5 else BAD} \text{ else UNR} \quad \langle \ldots \rangle$$

In the true branch, after a few steps, we rebuild the scrutinee $x > 0$. In this case, $\forall x : \text{int}, x > 0 \Rightarrow x > 0$ is valid. By [R-s-match], we take the true branch, which is a constant 5. As both 5 and UNR cannot be simplified further, we rebuild them by [S-const] and [S-unr] respectively and obtain:

$$\langle \ldots \rangle \text{ if } x > 0 \text{ then } (\ldots) \text{ then 5 else BAD} \text{ else UNR} \quad \langle \ldots \rangle$$

Figure 5. SL machine
Theorem 3 (Correctness of SL machine). For all expression e, if 
\\langle [\emptyset] \; e \; [\emptyset] \rangle \rightsquigarrow [\emptyset] \; e \; [\emptyset] \; a.

The SL is designed in a way such that the simplified a preserves the semantics of the original expression e. The proof of Theorem 3 (in [39]) uses the fact that, if there exists e\_k such that 
\\langle H \; e\_k \; S \; L \rangle \rightsquigarrow [\emptyset] \; e\_k \; [\emptyset] \; S \; L \rangle \rightsquigarrow [\emptyset] \; e\_k \; [\emptyset] \; S \; L \rangle. Then e\_k \equiv a. (See Definition 1 for \equiv.)

Theorem 4 (Soundness of static contract checking). For all closed expression e, and closed and terminating contract t, 
\\langle [\emptyset] \; e \; t \; [\emptyset] \rangle \rightsquigarrow \; e' \; t' \; \text{BAD} \; \not\exists \; e' \; t' \Rightarrow e \; t.

Proof. By Theorem 3, Lemma 1 and Theorem 1.

For open expressions and open contracts, see [39].

5.2 Logicization

We now explain the conversion \langle [] \rangle f, which we call logicization. Figure 6 gives the abstract syntax of the logical formula supported by an SMT solver named Alt-ergo [8], which is an automatic theorem prover for polymorphic first-order logic modulo theories. It uses classical logic and assumes all types are inhabited. First, Alt-ergo allows us to represent data type declaration, e.g.,

type 'a list = Nil | Cons of 'a * ('a list).

In Alt-ergo code with type and logic declarations:

data type declaration in language M:

\[
\text{type } \overset{\wedge}{\text{id}} \; s = K_1 \; \text{of } \overset{\wedge}{\text{id}} \; t_1 \ldots K_n \; \text{of } \overset{\wedge}{\text{id}} \; t_n
\]

its alt-ergo code: type \overset{\wedge}{\text{id}} \; s

\begin{align*}
\text{logic } K : \overset{\wedge}{\text{id}} \; t \rightarrow \overset{\wedge}{\text{id}} \; s
\end{align*}

\text{Figure 7. Converting data type to Alt-ergo code}

\[
[\tau_1 \ldots \tau_n] \; T = [\tau_1 | \ldots | \tau_n] \; T
\]

\[
[\tau_1 \rightarrow \tau_2] = ([\tau_1] \times [\tau_2]) \; \text{arrow}
\]

\text{Figure 8. Converting OCaml types to logic type}

type 'a list

logic nil : 'a list

logic cons : 'a , 'a list -> 'a list

As Alt-ergo supports only first-order logic (FOL), the arguments of a logical function (e.g., cons) are given as a tuple. The type variable 'a is assumed universally quantified at top-level. The conversion algorithm for an arbitrary user-defined data type is in Figure 7.

A conventional way [22] to encode higher-order function to FOL is to define a type arrow and a logical function apply:

type ('a , 'b) arrow

logic apply : ('a , 'b) arrow , 'a -> 'b

where the 'a and 'b refer to a function’s input and output type respectively. Converting types in the language M is easy (Figure 8). Base types int and bool are data types with no parameter.

We now give an example to show what logicization can do.

(* val len : ('a list -> int *)

contract len = \{x | true\} -> \{y | y >= 0\}

let len s = match s with | [] -> 0

| x::u -> 1 + len u

(* val append : 'a list -> 'a list -> 'a list *)

contract append = \{xs | true\} -> \{ys | true\}

| \[] | \[] -> \[]

| x::u -> x | append xs ys

let append xs ys = match xs with

| \[] -> ys

| x::u -> x | append u ys

The function len computes the length of a list and the function append append lists. Let e\_a and e\_b stand for the definition and contract of append respectively. Applying only simplification rules (including reduction rules) to e\_a \overset{\wedge}{\text{id}} \; e\_b, we get (R3):

\[
\lambda x_1 . \lambda x_2 . \text{match } v_1 \text{ with } [\emptyset] \rightarrow\text{if len } v_2 = \text{len } v_1 + \text{len } v_2 \text{ then } v_2 \text{ else } \text{BAD} \uparrow

| x \rightarrow \text{if } (\text{len } (\text{append } u \; v_2) = \text{len } u + \text{len } v_2)

\text{then append } u \; v_2 \text{ else } \text{BAD} \downarrow

The simplification approach in [38] and the model-checking approach in [33] involve inferring top-level functions, while we do not. Instead, we axiomatize the top-level function definitions that were called in contracts and lift expressions under checking to logic level and consult an SMT solver. The challenge is to deal with non-total expressions (e.g., BAD) in our source code. In the literature about converting functional code (in an interactive theorem prover) to SMT formulae [1, 6, 9, 28], expressions are converted to a logical form directly. In [1], given a non-recursive function definition \( f = e \), they first \( \eta \)-expand \( e \) to get \( f = \lambda x_1 \ldots x_n . e' \) where \( e' \) does not contain \( \lambda \); if it is a recursive function, they as-
sume e is in a particular form such that all lambdas are at top-level and the function performs an immediate case-analysis over one of its arguments. Then, they form \( \forall x, f(x_1, \ldots, x_n) = [e_1] \) where \([\cdot] \) converts an expression to logical form. (On the other hand, \([\cdot] \) uses λ-lifting method: λ-abstractions are translated from inside out, where each λ-abstraction is replaced by a call to a newly defined functions, so \( \forall x, f(x_1, \ldots, x_n) = [e_1]; \ldots; \forall x_1, f = f(1) \).

This is fine for converting total terms, e.g., \( [\mathbb{S}] = 5 + [x] = x, \) etc., but what are \([\mathcal{B}]) \) and \([\mathcal{U}]) \)? Our key idea is not to convert an expression directly to a corresponding logical term, but form equality with \([\cdot] \) recursively (defined in Figure 9). The subscript \( f \) in \([e] \) denotes the expression \( e \). Moreover, we perform neither \( \eta \)-expansion (which does not preserve semantics in the presence of non-total terms) nor λ-lifting, and yet we allow arbitrary forms of recursive functions. We have such flexibility because we convert λ-abstraction and partial application directly with the help of apply. (Note that our logicization \([\cdot] \) can also produce higher-order logic formula for interactive proving by replacing \((apply(f, x)) \) with \((f(x)) \) and not converting the types.) No logicization work in the literature (including \([6, 9, 28, 34]) \) deal with non-total terms. The work \([6]) \) uses approaches in \([9, 28]) \) to deal with polymorphism while Alt-ergo itself supports polymorphism.

Our framework can systematically generate Alt-ergo code, like below, to show that those BADs in R3 are unreachable.

\[
\begin{align*}
\text{logic len: } \langle \text{a list, int} \rangle \\
\text{logic append: } \langle \text{a list,} \langle \text{a list, a list} \rangle \rangle \\
\text{axiom len_def_1 : forall a: } \langle \text{a list} \rangle. \text{ s = nil } \rightarrow \\
\text{apply(len,s) = 0} \\
\text{axiom len_def_2 : forall a: } \langle \text{a list} \rangle. \text{ forall x: } \langle \text{a list} \rangle. \\
\text{forall 1 : } \langle \text{a list} \rangle. \text{ s = cons(x,1) } \rightarrow \\
\text{apply(len,s) = 1 + apply(len,1)} \\
\text{goal app_1 : forall v1,v2: } \langle \text{a list} \rangle. \text{ v1 = nil } \rightarrow \\
\text{apply(len,v1,v2) = apply(len,v1) + apply(len,v2)} \\
\text{goal app_2 : forall v1,v2,1: } \langle \text{a list} \rangle. \text{forall x: } \langle \text{a list} \rangle. \text{ v1 = cons(x,1) } \rightarrow \\
\text{apply(len,apply(apply(app_1),v2))} \\
= \text{apply(len,1) + apply(len,v2) } \rightarrow \\
(\exists y: \langle \text{a list} \rangle. \text{ y = apply(apply(app_1),v2)} \) and \( \text{apply(len,cons(x, y))} \) \\
= \text{apply(len,v1) + apply(len,v2)} \\
\end{align*}
\]

To make an SMT solver’s life easier (i.e., multiple small axioms are better than one big axiom), we have two axioms for \( \text{len} \), one for each branch, which are self-explanatory. As a constructor is always fully applied, we do not encode its application with apply. The \( \rightarrow \) (in axioms and goals) is a logical implication.

For example, the axiom \text{len_def_1} \ is generated by:

\[
[\lambda s. \langle \text{a list} \rangle. \text{match s with } [\text{nil} \rightarrow 0] \rangle \text{len} \\
= \forall s: \langle \text{a list} \rangle. \text{match s with } [\text{nil} \rightarrow 0] \text{apply(len,s))} \\
= \forall s: \langle \text{a list} \rangle. \exists x_0. \langle \text{a list} \rangle. [\text{nil} \rightarrow 0] \text{apply(len,s)} \\
= \forall s: \langle \text{a list} \rangle. \exists x_0. \langle \text{a list} \rangle. \text{ s = nil } \rightarrow \\
(\text{apply(len,s) = 0}) \\
\]

Letting \( x_0 \) be \( s \), we get a more readable version (axiom \text{len_def_1}).

An algorithm that partially eliminates redundant existentially quantified variables can be found in \([39]).

**Theorem 5** (Logicization for axioms). *Given closed definition \( f = e' \), the logical formula \( \exists : \tau, [e]_f \) is valid.*

\[
\begin{align*}
\{ \emptyset \in \{ +, -, *, / \} \\
\{ \emptyset \in \{ \geq, \leq, = \} \\
\text{let (rec) } f = e_f \rightarrow \\
\text{[BAD]}_f = \langle \text{false} \rangle \text{ for axioms} \\
\text{[UNR]}_f = \langle \text{false} \rangle \\
\text{[n]}_f = \langle \text{false} \rangle \\
\text{[e}_1 + e_2]_f = \exists x_1: \langle \text{e}_1 \rangle. \exists x_2: \langle \text{e}_2 \rangle. \text{ [x]_f = x_1 + x_2} \\
\text{[e}_1 \times e_2]_f = \exists x_1: \langle \text{e}_1 \rangle. \exists x_2: \langle \text{e}_2 \rangle. \text{ [x]_f = x_1 \times x_2} \\
\text{[e}_1 \equiv e_2]_f = \exists x_1: \langle \text{e}_1 \rangle. \exists x_2: \langle \text{e}_2 \rangle. \text{ [x]_f = x_1 \equiv x_2} \\
\text{[e]\forall x_1: } \langle \text{e}_1 \rangle. \exists x_2: \langle \text{e}_2 \rangle. \text{ [x]_f = x_1 \forall x_2} \\
\text{[e]\exists x_1: } \langle \text{e}_1 \rangle. \exists x_2: \langle \text{e}_2 \rangle. \text{ [x]_f = x_1 \exists x_2} \\
\text{[let x' = e in e']_f = } \exists x: \langle \text{e} \rangle. \langle \text{e} \rangle (\text{apply(f, x)}) \\
\text{[let x = e in e']_f = } \exists x: \langle \text{e} \rangle. \langle \text{e} \rangle (\text{apply(f, x)}) \\
\text{[\lambda x. e]_f = } \exists x: \langle \text{e} \rangle. \langle \text{e} \rangle (\text{apply(f, x)}) \\
\text{[\lambda x. e_1 \ldots e_n]_f = } \exists x: \langle \text{e}_1 \ldots \text{e}_n \rangle. \langle \text{e}_1 \ldots \text{e}_n \rangle (\text{apply(f, x)}) \\
\text{match e with K e_1 \ldots e_n]_f = } \langle \forall x: \langle \text{e} \rangle. \langle \text{e} \rangle (\text{apply(f, x)}) \rangle \\
\end{align*}
\]

**Figure 9.** Convert expression to logical formula.

Next, what query (i.e., goal) shall we make? All we want is to check if the branch leading to BAD is reachable or not. So our task is to examine the scrutine for a match expression. For example, the goal \text{app_1} states that the ctx-info \( \text{v1=\text{nil}} \), which is from the pattern matching \text{match v1 with} \([\text{v1}] \rightarrow \ldots \), implies the scrutine. By \([\text{S-lam}]) and \([\text{R-ssave}]) we have \( [\text{e}]=\) a list, \( \text{forall v2:} \langle \text{a list} \rangle, \text{v1 = \text{nil}} \). The scrutine is \([\text{len v2 = len v1 + len v2}] \) true. That is, we want to check whether \( \text{len v2 = len v1 + len v2} \) is equivalent to true. Alt-ergo says \text{valid} for both goals. Thus, we know both \text{BAD} and \text{UNR} are not reachable.

**Theorem 6** (Logicization for goals: validity preservation). *For all (possibly open) expression \( e' \), for all \( f[e] \), \( \text{if } \gamma, \text{ then } f[e']_f \) is valid and \( e \rightarrow e' \) for some \( e' \), then \([e']_f \) is valid.*

More details on design choices are in \([39]). Here, we highlight a few. (1) Only functions called in contracts are converted to Alt-ergo axioms. (2) In Figure 9, there are two conversions for BAD, true for axioms. This is for generating a harmless axiom \text{true} for the crashing branch of a partial function called in contracts. (3) For goals, the \([e]_f \) collects ctx-info before a scrutine for a match expression, thus, \([\text{BAD}]) \) false, which implies everything, which is what we want.

### 5.3 Discussion and preliminary experiments

One might notice that the SL machine simplifies terms under lambda and the body of match expression while we do not have such execution rules in Figure 2. As we rebuild blame and do not inline recursive functions (i.e., no crashing and no looping during simplification), the SL machine does not violate call-by-value execution.

One might worry that the rule \([\text{S-match}]) causes exponential code explosion for static analysis (although no run-time overhead). From our current observation, quite often the scrutine is if \( b \) then \( d \) else \( e \) where \( e \) is BAD or UNR. As blamwe trigger the SL machine to immediately rebuild the blame with the rest of the stack, applying the rule \([\text{S-match}]) , we do not have duplication but have a desired smaller formula for the SMT solver. We
have implemented a prototype based on the source code of ocamlc-3.12.1. Table 1 shows the results of preliminary experiments, which are done on a PC running Ubuntu Linux with a quad-core 2.93GHz CPU and 3.2GB memory. We take some examples from [26] and OCaml stlib and time the static checking. The column Ann gives the LOC count for contract annotations. One advantage of the SL machine is that it allows rules to be easily added or removed. This paper focuses on the theory of hybrid contract checking. We leave optimization and rigorous experimentation on tuning the strength of symbolic simplification and the frequency of calling an SMT solver as future work.

### 6. Hybrid contract checking

We have explained with examples how SCC, DCC, HCC work in §2. Programmers may choose to have SCC only, DCC only, or HCC. In this section, we summarize their algorithm. Given a program $f_i \in T_i$, $f_i = e_i$ for $1 \leq i \leq n$. Suppose $f_i$ is the current function under contract checking; $f_j$ is a function called in $f_i$ (including $f_i$’s recursive call); $\lambda$ is the SL machine; rmUNR implements the rule $\text{[rmUNR]}$ (mentioned earlier in §2).

\[
(\text{if } e_0 \text{ then } e_1 \text{ else } \text{UNR}) \Rightarrow e_1 \text{ \text{[rmUNR]}}
\]

We have:

- **SCC**: $\lambda e_i[(f_j \triangleright_l t_{f_j}) / f_j] \triangleright_l t$
- **DCC**: $e_i[(f_j \triangleright_l t_{f_j}) / f_j]$
- **HCC**: $f_i = \lambda !. \text{rmUNR}(\lambda e_i[(f_j \triangleright_l t_{f_j}) / f_j] \triangleright_l t)$

In HCC, the residual code $f_i$’s parameter "$\triangleright_l t$" waits for a caller’s name. For example, if an SMT solver cannot prove the goal $\text{app}_2$ in §5.2 (although it can), recalling R3 in §5.2, the residual code append$\triangleright_l t$ is:

\[
\lambda !. \lambda t_1. \lambda t_2. \lambda v_1. \text{match } t_1 \text{ with }\\ \begin{cases} [1] & \rightarrow t_2; \\ t & \rightarrow \text{if } \text{len}(x) = 1 \text{ then } t_2 \text{ else BAD} \end{cases}\]

which says that we only have to check the postcondition for the second branch. (If all BADs are simplified away during SCC, a residual code of a function is its original definition.)

**Lemma 2** (Telescopery property [7, 40]). For all expression $e$, total contract $t$, blames $r_1, r_2, r_3, r_4$, $e \uparrow t$ $t \uparrow r_1 \quad e \uparrow \uparrow t \quad t \uparrow r_2 \quad t \uparrow r_3 \quad e \uparrow \uparrow t \quad t \uparrow r_4 \quad t \uparrow r_4 \quad t \uparrow r_4$.

Precondition of a function is checked at caller sites. An $f_j \triangleright_l t_{f_j}$ is the simplified $f_j \triangleright_l t_{f_j}$, inspecting [HCC], each $f_j$ at caller sites is replaced by $(f_j \triangleright_l t_{f_j}) \triangleright_l t_{f_j}$, which is $(f_j \triangleright_l t_{f_j}) \text{ unapplied} t_{f_j}$.

By the telescoping property, we have:

\[
(f_j \triangleright_l t_{f_j}) \triangleright_l t_{f_j} = f_j \triangleright_l t_{f_j} \text{ [T1]}
\]

which is the same as in DCC. This shows that [HCC] blames $f$ if and only if [DCC] blames $f$.

Moreover, [T1] justifies the correctness of applying the rule $\text{[rmUNR]}$ because all UNRs are indeed unreachable as BAD$^l$ is invoked before UNR$^l$ for the same $t$. That is, $e \uparrow p \text{ then } e \uparrow e \text{ else } BAD$ is invoked before $(i f p \text{ then } e \uparrow e \text{ else } UNR^l)$ for the same $p$, maybe different $e$. So it is safe to apply the rule $\text{[rmUNR]}$ even if $p$ diverges or crashes. See [39] for more details.

### 7. Related work

Contract semantics were first formalized for a strict language [7, 11] and later for a lazy language [40]. This paper adapts and formalizes some of their ideas on contract satisfaction and contract checking. Detailed design difference is explained in §4.

Pre/post-condition specification using logical formulae [2, 15, 17, 34] allows programmers to existentially quantify over infinite domains or express metaproperties that are not expressible in contracts. We like the idea of ghost refinement [36], which separates properties that can be converted to program code from the metaproperties expressed only as logical formulae. As there are always limits to automatic static checking, it is practical to convert some difficult checks to dynamic checks. Unlike pre/post-condition specification, refinement types and contracts allow us to study sub-contract relations [11, 41], recursive contracts [7], and polymorphic contracts [3]. Contracts also enjoy interesting mathematical properties [7, 11, 39, 40].

One might recall hybrid refinement type checking (HTC) [13, 24]. In theory [16], (picky, i.e. our) contract checking is able to give more blame than refinement type checking in the presence of higher-order dependent function contracts. That is partly why [36] invents a Kind checker to report ill-formed refinement types. As discussed in §4.2, we check $e \uparrow t$ for crash-freeness in one-go and do not have to check $t$ to be crash-free separately. In practice, the $\eta$ and $\ell$ in the SL machine serve a similar purpose as the typing environment in HTC. But symbolic simplification gives more flexibility in such ways as teasing out the path sensitivity analysis with the rule $\text{[S-m-match]}$, etc. We hope this work opens a venue to compare HCC and HTC in practice. Notably, VeriFast [20] (for verifying C and Java code) suggests that symbolic execution is faster than the verification condition generation method [2, 15].

Kho et al. [3] mix type checking and symbolic execution. Besides they do not generate residual code, they require programmers to place block annotations $\{., .\}$ for type checking and $\{., .\}$ for symbolic execution while our SL machine systematically simplifies subterms and consults the logical store for checking at the appropriate program point. Moreover, their symbolic expression is given in linear arithmetic, which is more restrictive than ours.

Our approach is different from [36], which extracts proofs of refinement types from an SMT solver and injects them as terms in the generated bytecode RDCIL (like proof carrying code) during refinement type checking. Theirs has a security motivation.

Some work [25, 26, 32, 33] suggests converting programs to a higher-order recursive scheme (HORS), which generates (possibly infinite) trees, and specify properties in the form of a trivial automaton and do model checking to see whether HORS satisfies its desired property. Our approach is completely different although we both do reachability checking. They work on automata while we work on programs directly. Our approach is modular while theirs is not. They deal with local let rec while we do not, but we could infer local contract with method in [21] or inline the local let rec function for a fixed number of times. They deal with protocol checking while we do not, except where a protocol checking problem can be converted to checking the reachability of BAD.

The contextual information synthesis and conversion of expression to logical formula is inspired by the use of the application $\bullet$ in [18, 19], which makes conversion of higher-order functions easier. But we use the technique in different contexts.
Many papers on program verification [2, 10, 15, 30, 31, 37] focus on memory leaks, array bound checks, etc. and a few handle higher-order functions and recursive predicates. Our work focuses on more advanced properties and precise blaming of functions at fault. Contract checking in the imperative world is lead by [10], which statically checks contract satisfaction at the bytecode CIL level and runs dynamic checking separately. Residualization has not been done in [10]. We may adapt some ideas in [20] to extend our framework for program with side effects.

8. Conclusion

We have formalized a contract framework for a pure, strict, higher-order subset of OCaml. We propose a natural integration of static contract checking and dynamic contract checking. With the SL machine, our approach gives precise blame at both compile-time and run-time in the presence of higher-order functions. In the near future, besides rigorous experimentation and case-studies, we plan to add user-defined exceptions, allow side effects in program and hidden side effects in contracts, do contract or invariant inference.

Acknowledgments

I would like to thank Xavier Leroy, Francois Pottier, Nicolas Pouillard, Martin Berger, Simon Peyton Jones, Michael Greenberg, and the anonymous reviewers for their comments.

References