A Certificate Infrastructure for Machine-Checked Proofs of Conditional Information Flow

Torben Amtoft[1], Josiah Dodds[2], Zhi Zhang[1], Andrew Appel[2], Lennart Beringer[2], John Hatcliff[1], Xinming Ou[1], and Andrew Cousino[1]

[1] Kansas State University, CIS Department, 234 Nichols Hall, Manhattan KS 66506
{tamtoft,zhangzhi,hatcliff,xou,acousino}@ksu.edu
[2] Princeton University, Dept. of Comp. Sci., 35 Olden Street, Princeton NJ 08540
{jdodds,appel,eberinge}@cs.princeton.edu

Abstract. In previous work, we have proposed a compositional framework for stating and automatically verifying complex conditional information flow policies using a relational Hoare logic. The framework allows developers and verifiers to work directly with the source code using source-level code contracts. In this work, we extend that approach so that the algorithm for verifying code compliance to an information flow contract emits formal certificates of correctness that are checked in the Coq proof assistant. This framework is implemented in the context of SPARK – a subset of Ada that has been used in a number of industrial contexts for implementing certified safety and security critical systems.

1 Introduction

Network and embedded security devices have complex information flow policies that are crucial to fulfilling device requirements. We have previously explained [1, §1] how devices (such as "separation kernels") developed following the MILS (Multiple Independent Levels of Security) architecture must be certified to very stringent criteria such as Common Criteria EAL 6/7 and DCID 6/3, and that many previous information-flow analyses (based on type systems) are too weak to specify these systems; the notion of *conditional information flow* is needed. We have also explained [2] that in these real applications, one must be able to trace information flow through individual array elements, rather than "contaminating" the entire array whenever an assignment is done.

SPARK (a safety-critical subset of Ada) is being used by various organizations, including Rockwell Collins[1] and the US National Security Agency (NSA) [3], to engineer information assurance systems including cryptographic controllers, network guards, and key management systems. To guarantee analyzability and conformance to embedded system resource bounds, SPARK does not include pointers and heap-based data. Thus, SPARK programs use arrays and for-loops to implement complex data structures. SPARK provides automatically checked procedure annotations that specify information flows (dependences) between procedure inputs and outputs. In the certification process, these annotations play a key role in justifying conformance to information flow and

[1] See the 2006 press release at http://212.113.201.96/sparkada/pdfs/praxis_rockwell_final_pr.pdf

P. Degano and J.D. Guttman (Eds.): POST 2012, LNCS 7215, pp. 369–389, 2012.

separation policies relevant to MILS development; however, the standard SPARK annotation language is too weak to express the flow policies needed to verify/certify many real embedded information assurance applications.

Due to the lack of precision in SPARK and other conventional language-based security frameworks, policy adherence arguments are often reduced to informal claims substantiated by manual inspections that are time-consuming, tedious, and error-prone. Some past certification efforts have created models of software code in theorem provers and proved that the models of code comply with security policies. While this strategy can provide high degrees of confidence and support very precise policy declarations, it has the disadvantages of (a) leaving "trust gaps" between source code and models (their correspondence has in past efforts been only manually verified by inspection), (b) requiring intensive manual efforts, and (c) inhibiting developers from proceeding with information flow specification and verification during the development process.

In our previous work [2,1] we extended SPARK's procedure annotations to conditional information flow and fine-grained treatment of structured data, necessary for the automatic analysis and verification of many programs, and we developed a compositional framework for stating and automatically verifying complex array-oriented and conditional information flow policies using a relational Hoare logic. Although our Secure Information Flow Logic (SIFL) is language-neutral, we have chosen to cast our work as an enhancement to the SPARK information flow framework. Indeed, this work has been inspired by challenge problems provided by our industrial collaborators at Rockwell Collins who are using SPARK on several projects.

Here we extend our framework with new functionality to generate machine-checkable proofs of the information-flow properties that it derives. Our framework is *much* more automated than tactical theorem-proving in a proof assistant. In our framework, engineers work directly with the source code using code contracts to specify/check with greater precision than in conventional language-based information flow frameworks. We believe that most units (e.g., procedures) of real embedded applications can be handled directly by our analysis—and those units that cannot may smoothly be handed off to verification in a proof assistant; the compositional nature of our system will eventually allow the whole system to be checked end-to-end in the proof assistant.

Contributions: (a 50+ pages technical report describing the details of the approach, as well as Coq proofs for evidence soundness, is available at [4].)

- We enhance our previously developed precondition-generation algorithm for SIFL assertions to emit *evidence* that program units conform to their (conditional) information flow contracts. This evidence can be viewed as an application of rules of a relational logic for information flow that encodes the algorithm's reasoning steps.
- We provide an implementation of the evidence-emitting precondition generation algorithm for SPARK.
- We encode the derived logic in Coq, and prove it sound with respect to an operational semantics for a core subset of SPARK. We thus have a foundational machine-checked proof that whenever our evidence-checker accepts evidence about a program, then that program really does conform to the given information flow policy.
- We evaluate the framework on a collection of methods from embedded applications, including applications from industrial research projects.

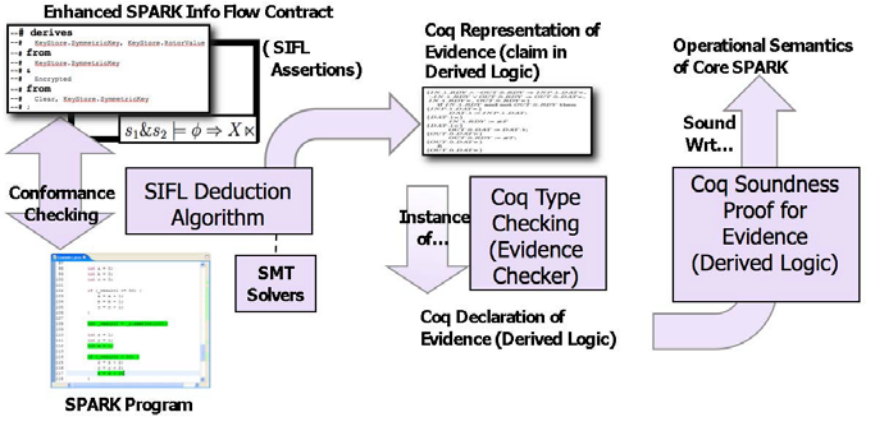
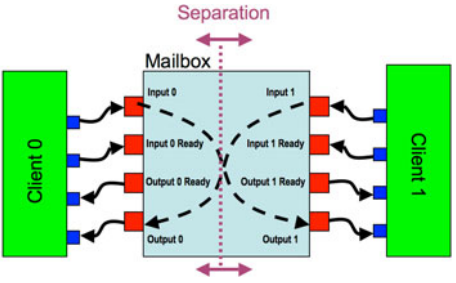


Fig. 1. Structure of Information Flow Evidence Generation and Checking

2 Background

SPARK is a safety critical subset of Ada developed by Altran Praxis and supported by AdaCore. SPARK provides (a) an annotation language for writing functional as well as information-flow software contracts, and (b) automated static analyses and semi-automated proof assistants for proving absence of run-time exceptions, and conformance of code to contracts. SPARK has been used to build a number of high-assurance systems; Altran Praxis is currently using it to implement the next generation of the UK air traffic control system. We are using SPARK due to our strong collaborative ties with Rockwell Collins, who uses SPARK to develop safety and security critical components of a number of embedded systems.

Figure 1 illustrates the structure of our SIFL contract checking and evidence generation framework. An Eclipse-based integrated development environment allows programmers to develop information-assurance applications in SPARK. Our logic-based approach allows us to extend the SPARK information-flow contract language to include support for conditional information flow and quantified flow policies that describe flows through individual components of arrays. Behind the scenes, the enhanced SPARK information flow contracts are represented using relational *agreement assertions* (explained below). Our tool framework includes a precondition generation algorithm for agreement assertions that allows us to infer SIFL contracts or check user-supplied contracts; in the latter mode, preconditions are inferred from postconditions, and then the tool checks that the user-supplied preconditions imply the inferred preconditions. The precondition generation algorithm uses a collection of SMT solvers via the Sireum Topi interface (`www.sireum.org`). As the precondition generation executes, it builds evidence, in the form of a Coq data structure, that relates postconditions to generated preconditions. This language of evidence is a relational Hoare logic derived from the basic reasoning steps in the algorithm. Coq type checking acts as an "evidence checker" confirming that the evidence emitted by the algorithm is indeed well-formed. The Coq evidence representation is proved sound in Coq wrt. an operational semantics for an imperative language representing core features of SPARK. Thus, given a SPARK program and a




```
procedure MACHINE_STEP
-- INFORMATION FLOW CONTRACT (Figure 3)
is D_0, D_1 : CHARACTER;
begin
   if IN_0_RDY and not OUT_1_RDY then
      D_0 := IN_0_DAT;   IN_0_RDY := FALSE;
      OUT_1_DAT := D_0;  OUT_1_RDY := TRUE;
   end if;
   if IN_1_RDY and not OUT_0_RDY then
      D_1 := IN_1_DAT;   IN_1_RDY := FALSE;
      OUT_0_DAT := D_1;  OUT_0_RDY := TRUE;
   end if;
end MACHINE_STEP;
```

Fig. 2. Simple MLS Guard - mailbox mediates communication between partitions

SIFL contract, if the contract checking algorithm produces evidence that type checks in Coq, we have a machine-checked proof that the program's behavior (as defined by the operational semantics) conforms to the information-flow contract.

Observe that the precondition generator is not part of the trusted code base and hence in principle might fail to produce well-typed evidence, but we are exploring (cf. the end of Sect. 4) an implementation of the precondition generator inside Coq. Once verified, this Gallina implementation cannot fail to produce valid evidence.

Figure 2 illustrates the conceptual information flows in a fragment of a simplistic MLS (Multiple Levels of Security) component, described in our earlier work [1]. Rockwell Collins engineers constructed this example to illustrate, to NSA and industry representatives, the specification and verification challenges facing the developers of MLS software. The "Mailbox" component in the center of the diagram mediates communication between two client processes – each running on its own partition in the separation kernel. *Client 0* writes data to communicate in the memory segment *Input 0* that is shared between *Client 0* and the mailbox, then it sets the *Input 0 Ready* flag. The mailbox process polls its ready flags; when it finds that, *e.g.*, *Input 0 Ready* is set and *Output 1 Ready* is cleared (indicating that *Client 1* has already consumed data deposited in the *Output 1* slot in a previous communication), then it copies the data from *Input 0* to *Output 1* and clears *Input 0 Ready* and sets *Output 1 Ready*. The communication from *Client 1* to *Client 0* follows a symmetric set of steps. The actions to be taken in each execution frame are encoded in SPARK by the `MACHINE_STEP` procedure of Fig. 2.

While upper levels of the MILS architecture require reasoning about lattices of security levels (*e.g.*, *unclassified*, *secret*, *top secret*), the policies of infrastructure components such as separation kernels and guard applications usually focus on data separation policies (reasoning about flows between components of program state), and we restrict ourselves to such reasoning in this paper.

Figure 3(a) displays a fragment of an information flow contract for the mailbox example written in our contract language that enhances the original SPARK contract language with the ability to specify conditional information flows. This specification states from which input values (and under which conditions) the final values of `OUT_0_DAT` and `OUT_1_DAT` are *derived*. For example, `OUT_0_DAT` always derives from `IN_1_RDY` and `OUT_0_RDY` because these "guarding variables" determine whether or not the body of the conditional that assigns to `OUT_0_DAT` is executed, i.e., `OUT_0_DAT` is control dependent on `IN_1_RDY` and `OUT_0_RDY`. In addition, the final value of `OUT_0_DAT` depends

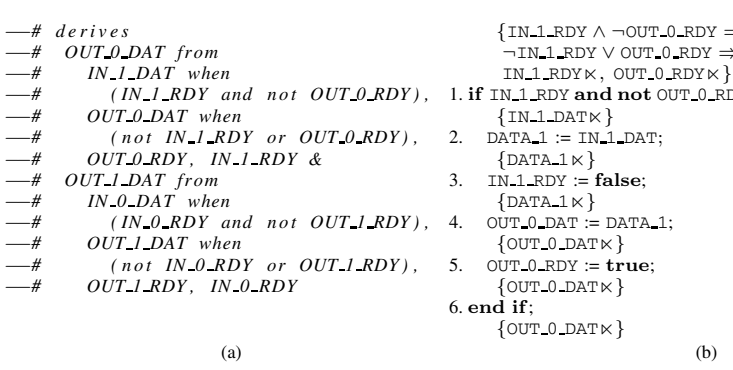


Fig. 3. (a) Fragment of conditional information flow contract. (b) Corresponding derivation with SIFL assertions.

on the initial value of IN_1_DAT when the flag IN_1_RDY is set and the flag OUT_0_RDY is cleared; otherwise, it depends on the initial value of OUT_0_DAT.

The derives clauses in SPARK, like most formal specification mechanisms for information flow, are unconditional (e.g., they do not include the when clauses illustrated in Figure 3). Thus, they cannot distinguish the flag variables as guards nor phrase the conditions under which the guards allow information to pass or be blocked. This means that guarding logic, which is central to many security applications including those developed at Rockwell Collins, *is completely absent from the checkable specifications* in SPARK. In general, the lack of ability to express *conditional* information flow not only inhibits automatic verification of guarding logic specifications, but also results in imprecision which cascades and builds throughout the specifications in the application.

To capture conditional information flow as well as other forms of information that cannot be specified in SPARK, we have been building [1,2] on a reasoning framework based on *conditional* agreement assertions, also called *2-assertions*, originally introduced by Amtoft and Banerjee [5]. These SIFL assertions are of the form $\phi \Rightarrow E{\ltimes}$, where ϕ is a boolean expression and E is any kind of expression (to be defined in the next section), which is satisfied by a pair of stores if either at least one of them does not satisfy ϕ, or they agree on the value of E:

Definition 1. $s_1 \& s_2 \models \phi \Rightarrow E{\ltimes}$ *iff* $[\![E]\!]_{s_1} = [\![E]\!]_{s_2}$ *whenever* $s_1 \models \phi$ *and* $s_2 \models \phi$.

We use $\theta \in \mathbf{twoAssn}$ to range over 2-assertions. For $\theta = (\phi \Rightarrow E{\ltimes})$, we call ϕ the antecedent of θ and write $\phi = ant(\theta)$, and we call E the consequent of θ and write $E = con(\theta)$. We use $\Theta \in \mathcal{P}(\mathbf{twoAssn})$ to range over sets of 2-assertions, with conjunction implicit. Thus, $s \& s_1 \models \Theta$ iff $\forall \theta \in \Theta : s \& s_1 \models \theta$. We often write $E{\ltimes}$ for $\mathit{true} \Rightarrow E{\ltimes}$, and often write θ for the singleton set $\{\theta\}$.

Fig. 3(b) illustrates a simple derivation using SIFL assertions that answers the question: what is the source of information flowing into variable OUT_0_DAT? The natural way to read the derivation is from the bottom up (since our algorithm works "backwards"). Thus, for OUT_0_DAT⋉ to hold after execution of P, we must have DATA_1⋉ before line 4 (since data flows from DATA_1 to OUT_0_DAT), IN_1_DAT⋉ before line 2 (since data flows from IN_1_DAT to DATA_1), and before line 1 IN_1_RDY⋉ and

OUT_0_RDY⋉ (since they *control* which branch of the condition is taken), along with conditional assertions. The precondition shows, just as we would expect, that the value of OUT_0_DAT depends *unconditionally* on IN_1_RDY and OUT_0_RDY, and *conditionally* on IN_1_DAT and OUT_0_DAT.

3 Evidence Representations

One of our primary goals in this paper is to design *evidence* terms η whose types correspond to triples: if we can establish that η has type $\{\Theta\}\ C\ \{\Theta'\}$ then the command C has information flow property given by precondition Θ and postcondition Θ' where both are sets of *2-assertions*. Intuitively, evidence represents the primary reasoning steps taken in the precondition generation algorithm when constructing a derivation such as the one displayed in Fig. 3(b). We shall need several auxiliary kinds of evidence, described later but summarized below:

$\vdash \eta : \{\Theta\}\ C\ \{\Theta'\}$	η shows C has information flow pre/post-condition Θ/Θ'
$\vdash \nu : \phi \overset{C}{\Leftarrow} \phi'$	ν shows ϕ is NPC for ϕ' wrt. C
$\vdash \iota : \phi \Rightarrow_1 \phi'$	ι shows ϕ logically implies ϕ'
$\vdash \tau : \Theta \Rightarrow_2 \Theta'$	τ shows the 2-assertions in Θ logically imply Θ'
$\vdash \mu : C$ mods_only X	μ shows C modifies at most X

3.1 Preliminaries

We shall now describe our language, sufficient to represent the primary features of SPARK. We shall consider only one-dimensional[2] arrays, and model such an array as a total mapping from integers into integers that is zero except for a finite number of places; we do thus not try to model array bounds which is an orthogonal issue and we assume that SPARK development tools have been applied to prove that there are no index range nor arithmetic overflow violations.

Basic Syntax. Commands C are given by the abstract syntax

$$C ::= \mathbf{skip} \mid C\,;C \mid \mathbf{assert}(B) \mid x := A \mid h := H \mid \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \\ \mid\ \mathbf{while}\ B\ \mathbf{do}\ C \mid \mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ C$$

In $\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ C$ we require q and m to be different identifiers, neither modified by C. We use x (and y, z) to range over scalar identifiers, h to range over array identifiers, and z, w to range over either kind of identifier; we use A to range over arithmetic expressions, B and ϕ to range over boolean expressions which are also called *1-assertions*, H to range over array expressions, and E to range over any kind of expression. Those are given by the syntax

$$A ::= c \mid x \mid A\ \mathsf{op}\ A \mid H[A] \\ B ::= A\ \mathsf{bop}\ A \mid \mathit{true} \mid \mathit{false} \mid B \wedge B \mid B \vee B \mid \neg B \\ H ::= h \mid Z \mid H\{A : A\}$$

[2] Multi-dimensional arrays are supported in the complete SPARK language, but they are not yet supported in our theory nor in our tool implementation.

where c ranges over constants, op ranges over binary arithmetic operators, bop ranges over binary comparison operators, and Z denotes the array which is zero everywhere. Note that an array identifier can be assigned an arbitrary array expression but we will typically only do a one-place update: we use the standard notation $h[A_0] := A$ as a shorthand for $h := h\{A_0 : A\}$.

Semantics. A value is just an integer in Int. Thus a store s is a (partial) mapping from scalar identifiers into values, and from array identifiers into total functions in $\mathsf{Int} \to \mathsf{Int}$ where we shall use a to range over members of that function space. Then $[\![A]\!]_s$ denotes the value resulting from evaluating A in store s, $[\![H]\!]_s$ denotes the function resulting from evaluating H in store s, and $[\![B]\!]_s$ denotes the boolean resulting from evaluating B in store s. We we say that s satisfies ϕ, written $s \models \phi$, iff $[\![\phi]\!]_s = \mathsf{True}$. Below, we shall list the semantic clauses that deal with arrays (the other clauses are straightforward):

$$[\![H[A]]\!]_s = [\![H]\!]_s([\![A]\!]_s) \qquad [\![Z]\!]_s = \lambda n.0$$
$$[\![H\{A_0 : A\}]\!]_s = [[\![H]\!]_s \mid [\![A_0]\!]_s \mapsto [\![A]\!]_s]$$

We write $s\ [\![C]\!]\ s'$ if the command C transforms the store s into store s'. For example, $s\ [\![h := H]\!]\ s'$ iff for some a we have $a = [\![H]\!]_s$ and $s' = [s \mid h \mapsto a]$.

Given C and s, there exists at most one s' such that $s\ [\![C]\!]\ s'$ holds; if C is a **while** loop that loops on s or an **assert** command that fails then no such s' will exist.

In **for** loops, we allow zero iterations and let the final value of the counter q be one above the bound; then one can prove that a **for** loop can be expressed as a **while** loop: for all s and s', $s\ [\![\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ C]\!]\ s'$ iff $s\ [\![q := 1\,; C_w]\!]\ s'$ where C_w is given by $\mathbf{while}\ q \leq m\ \mathbf{do}\ (C\,; q := q + 1)$.

To analyze **for** loops, we could thus rely on an analysis for **while** loops, but we shall present (Sect. 3.6) a specialized analysis of **for** loops that often gives more precise information than analyzing the equivalent **while** loop would have done.

3.2 Evidence

We shall provide rules, numbered below from (1) to (12), for inferring judgements of the form $\vdash \eta : \{\Theta\}\ C\ \{\Theta'\}$. Each rule corresponds to an evidence construct and is designed so as to enable the following soundness property:

Theorem 2. *Assume that* $\vdash \eta : \{\Theta\}\ C\ \{\Theta'\}$. *Then* $\models \{\Theta\}\ C\ \{\Theta'\}$.

Here $\models \{\Theta\}\ C\ \{\Theta'\}$ denotes the desired semantic soundness result: if $s_1 \& s_2 \models \Theta$, and $s_i\ [\![C]\!]\ s_i'$ for $i = 1, 2$, then $s_1' \& s_2' \models \Theta'$. Also, evidence has unique type: if $\vdash \eta : \{\Theta_i\}\ C_i\ \{\Theta_i'\}$ for $i = 1, 2$ then $\Theta_1 = \Theta_2$, $C_1 = C_2$, and $\Theta_1' = \Theta_2'$.

Syntax-Directed Evidence. For each syntactic construct there is a corresponding piece of evidence. For the most basic constructs, the inference rules are listed in Fig. 4. (For space reasons, we omit the evidence for **while** loops, and we postpone **for** loops until Sect. 3.6.) We let $\mathsf{add}^{\wedge}_B(\Theta)$ denote the result of conjoining B to the antecedent of each assertion in Θ: $\mathsf{add}^{\wedge}_B(\Theta) = \{\mathsf{add}^{\wedge}_B(\theta) \mid \theta \in \Theta\}$ where $\mathsf{add}^{\wedge}_B((\phi \Rightarrow E\ltimes)) = (\phi \wedge B) \Rightarrow E\ltimes$.

$$\frac{}{\vdash \mathsf{SkipE}(\Theta) \,:\, \{\Theta\}\ \mathbf{skip}\ \{\Theta\}} \tag{1}$$

$$\frac{\vdash \eta_1 \,:\, \{\Theta_1\}\ C_1\ \{\Theta\} \qquad \vdash \eta_2 \,:\, \{\Theta\}\ C_2\ \{\Theta_2\}}{\vdash \mathsf{SeqE}(\eta_1,\eta_2) \,:\, \{\Theta_1\}\ C_1\,;C_2\ \{\Theta_2\}} \tag{2}$$

$$\frac{}{\vdash \mathsf{AssignE}(\Theta,x,A) \,:\, \{\Theta[A/x]\}\ x := A\ \{\Theta\}} \tag{3}$$

$$\frac{}{\vdash \mathsf{HAssignE}(\Theta,h,H) \,:\, \{\Theta[H/h]\}\ h := H\ \{\Theta\}} \tag{4}$$

$$\frac{}{\vdash \mathsf{AssertE}(\Theta,B) \,:\, \{\mathsf{add}^{\wedge}_{B}(\Theta)\}\ \mathbf{assert}(B)\ \{\Theta\}} \tag{5}$$

Fig. 4. Simple rules for syntax-directed evidence

Before presenting the rule for conditionals, we need to introduce the notion of "necessary precondition" (NPC). We say that ϕ is a NPC for ϕ' wrt. C if whenever $s\ [\![C]\!]\ s'$ and $s' \models \phi'$ then $s \models \phi$. It is easy to see that the set of NPCs for given C and ϕ' forms a *Moore family* (closed under arbitrary conjunction) and hence there exists a smallest (strongest) NPC, which is equal to $wp(C,\phi')$ with wp denoting "weakest precondition" (satisfied by a store s if there exists s' with $s\ [\![C]\!]\ s'$ and $s' \models \phi'$). It may be infeasible to compute $wp(C,\phi')$ exactly but then any *weaker* assertion (and trivially *true*) can be used as NPC. We use ν to range over evidence for NPC, described in Sect. 3.3.

The general rule for a conditional $C = \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$ is

$$\frac{\Theta = \{\phi \Rightarrow E{\ltimes}\} \qquad \vdash \eta_i \,:\, \{\Theta_i\}\ C_i\ \{\Theta\}\ (i=1,2) \qquad \vdash \nu \,:\, \phi_0 \overset{C}{\Leftarrow} \phi}{\vdash \mathsf{CondE}(\eta_1,\eta_2,\nu,B) \,:\, \{\mathsf{add}^{\wedge}_{B}(\Theta_1) \cup \mathsf{add}^{\wedge}_{\neg B}(\Theta_2) \cup \{\phi_0 \Rightarrow B{\ltimes}\}\}\ C\ \{\Theta\}} \tag{6}$$

which demands the postcondition to be a singleton; if not, we decompose it and then recombine using the UnionE evidence construct (9). The assertion $\phi_0 \Rightarrow B{\ltimes}$ occurring in the precondition expresses that if two runs must agree on the consequent E then they must also agree on the test B; this may be too restrictive if E is not modified by either branch in which case we can instead use the ConseqNotModE evidence construct (11).

We now look back at the derivation in Fig. 3(b). The analysis of the command in line 4 was done using rule (3), with $x = \text{OUT\_0\_DAT}$ and $A = \text{DATA\_1}$ and $\Theta = \text{OUT\_0\_DAT}{\ltimes}$, giving the precondition $\Theta[A/x] = \text{DATA\_1}{\ltimes}$. The analysis of the conditional in line 1 was done using rule (6), with $B = (\text{IN\_1\_RDY} \wedge \neg\text{OUT\_0\_RDY})$ and using evidence η_1 for the analysis of C_1 (the lines 2–5) and evidence η_2 for the analysis of C_2 (**skip**), both with postcondition $\Theta = \text{OUT\_0\_DAT}{\ltimes}$. From $\Theta_1 = \text{IN\_1\_DAT}{\ltimes}$ and $\Theta_2 = \text{OUT\_0\_DAT}{\ltimes}$, and from *true* being a NPC for *true*, we get the precondition

$$\Theta_0 = \{\ (\mathit{true} \wedge (\text{IN\_1\_RDY} \wedge \neg\text{OUT\_0\_RDY})) \Rightarrow \text{IN\_1\_DAT}{\ltimes},\ (\mathit{true} \wedge \neg(\text{IN\_1\_RDY} \wedge \neg\text{OUT\_0\_RDY})) \Rightarrow \text{OUT\_0\_DAT}{\ltimes},\ \mathit{true} \Rightarrow (\text{IN\_1\_RDY} \wedge \neg\text{OUT\_0\_RDY}){\ltimes}\}$$

Non-Syntax Directed Evidence. Some additional evidence constructs are given by the inference rules listed in Fig. 5 which we shall now explain and motivate.

The rules (7,8) allow us to strengthen the precondition or weaken the postcondition; here τ is evidence (described in Sect. 3.3) for *2-implication*: if $\vdash \tau \,:\, \Theta \Rightarrow_2 \Theta'$ then Θ logically implies Θ' (that is, for all stores s_1, s_2, if $s_1 \& s_2 \models \Theta$ then $s_1 \& s_2 \models \Theta'$).

$$\frac{\vdash \eta : \{\Theta''\}\ C\ \{\Theta'\} \qquad \vdash \tau : \Theta \Rightarrow_2 \Theta''}{\vdash \mathsf{PreImplyE}(\tau,\eta) : \{\Theta\}\ C\ \{\Theta'\}} \tag{7}$$

$$\frac{\vdash \eta : \{\Theta\}\ C\ \{\Theta''\} \qquad \vdash \tau : \Theta'' \Rightarrow_2 \Theta'}{\vdash \mathsf{PostImplyE}(\eta,\tau) : \{\Theta\}\ C\ \{\Theta'\}} \tag{8}$$

$$\frac{\vdash \eta_1 : \{\Theta_1\}\ C\ \{\Theta_1'\} \qquad \vdash \eta_2 : \{\Theta_2\}\ C\ \{\Theta_2'\}}{\vdash \mathsf{UnionE}(\eta_1,\eta_2) : \{\Theta_1 \cup \Theta_2\}\ C\ \{\Theta_1' \cup \Theta_2'\}} \tag{9}$$

$$\frac{\vdash \mu : C\ \mathsf{mods\_only}\ X \qquad \mathrm{fv}(\Theta) \cap X = \emptyset}{\vdash \mathsf{NotModE}(\mu,\Theta) : \{\Theta\}\ C\ \{\Theta\}} \tag{10}$$

$$\frac{\vdash \mu : C\ \mathsf{mods\_only}\ X \qquad \vdash \nu : \phi \overset{C}{\Leftarrow} \phi' \qquad \mathrm{fv}(E) \cap X = \emptyset}{\vdash \mathsf{ConseqNotModE}(\mu,\nu,E) : \{\phi \Rightarrow E\ltimes\}\ C\ \{\phi' \Rightarrow E\ltimes\}} \tag{11}$$

$$\frac{\vdash \eta : \{\Theta\}\ C\ \{\Theta'\} \qquad \vdash \nu : \phi \overset{C}{\Leftarrow} \phi'}{\vdash \mathsf{AntecStrongerE}(\eta,\nu) : \{\mathsf{add}^{\wedge}_{\phi}(\Theta)\}\ C\ \{\mathsf{add}^{\wedge}_{\phi'}(\Theta')\}} \tag{12}$$

Fig. 5. Rules for non-syntax-directed evidence

Two derivations may be combined using (9) which can trivially be generalized to an evidence construct combining an arbitrary number of elements: $\mathsf{UnionE}(\eta_1 \ldots \eta_n)$.

The rule (10) allows a simple treatment of 2-assertions when no identifier is modified; it uses evidence μ for not-modification: if $\vdash \mu : C\ \mathsf{mods\_only}\ X$ then all identifiers[3] possibly modified by C are included in X. We shall not represent such evidence explicitly, since not-modification is a syntactic property which can be checked easily by a simple Gallina function in Coq. Again looking back at the derivation in Fig. 3(b), we observe that line 5 could have been analyzed using rule (3) but can also be analyzed using rule (10) which is particularly powerful if applied to a whole block of code. For example, for the program in Figure 2, the precondition (shown in Figure 3) of the second conditional does not contain any identifiers that are modified by the first conditional, and hence by a single application of (10) can be shown to be also the precondition of the whole program.

Another rule (11) addresses the more general case where antecedents, but not consequents, may be modified. We then need to ensure that whenever two post-states are required to agree on the consequent, also the two pre-states are required to agree on the consequent. This is expressed using the notion of NPC, which is also used in (12) to allow us to make pre-and postconditions "more conditional", by strengthening the antecedents.

3.3 Auxiliary Evidence

Evidence for 2-Implication. In order to justify the simplication of assertions, or showing that a user-supplied precondition is correct in that it implies the precondition generated

[3] If C modifies just one entry of h then h has to be included in X. This may seem very imprecise, but we shall present (in Sect. 3.6) an analysis that in many cases does allow us to get precise information about how individual array elements are affected by **for** loops.

$$\frac{\Theta \supseteq \Theta_0}{\vdash \mathsf{Superset2I}(\Theta, \Theta_0) \ : \ \Theta \ \Rightarrow_2 \ \Theta_0} \tag{13}$$

$$\frac{\text{for all } \phi \Rightarrow E\ltimes \in \Theta, \text{ there exists no } s \text{ with } s \models \phi}{\vdash \mathsf{Vacuous2I}(\Theta) \ : \ \emptyset \ \Rightarrow_2 \ \Theta} \tag{14}$$

$$\frac{\mathrm{fv}(E) = \emptyset}{\vdash \mathsf{Const2I}(E, \phi) \ : \ \emptyset \ \Rightarrow_2 \ (\phi \Rightarrow E\ltimes)} \tag{15}$$

$$\frac{\vdash \tau_1 \ : \ \Theta_1 \ \Rightarrow_2 \ \Theta_1' \qquad \vdash \tau_2 \ : \ \Theta_2 \ \Rightarrow_2 \ \Theta_2'}{\vdash \mathsf{Union2I}(\tau_1, \tau_2) \ : \ \Theta_1 \cup \Theta_2 \ \Rightarrow_2 \ \Theta_1' \cup \Theta_2'} \tag{16}$$

$$\frac{\vdash \tau_1 \ : \ \Theta_1 \ \Rightarrow_2 \ \Theta_1' \qquad \vdash \tau_2 \ : \ \Theta_2 \ \Rightarrow_2 \ \Theta' \qquad \Theta = (\Theta_2 \setminus \Theta_1') \cup \Theta_1}{\vdash \mathsf{Trans2I}(\tau_1, \tau_2) \ : \ \Theta \ \Rightarrow_2 \ \Theta'} \tag{17}$$

$$\frac{\vdash \iota \ : \ \phi \ \Rightarrow_1 \ \phi'}{\vdash \mathsf{Contravar2I}(\iota, E) \ : \ \{\phi' \Rightarrow E\ltimes\} \ \Rightarrow_2 \ \{\phi \Rightarrow E\ltimes\}} \tag{18}$$

$$\frac{E \text{ is of the form } E_1 \ \mathsf{op} \ E_2 \text{ or } E_1 \ \mathsf{bop} \ E_2 \text{ or } E_1 \wedge E_2 \text{ or } E_1 \vee E_2 \text{ or } \neg E_1}{\vdash \mathsf{BinOp2I}(E, \phi) \ : \ \{\phi \Rightarrow E_1\ltimes, \phi \Rightarrow E_2\ltimes\} \ \Rightarrow_2 \ \{\phi \Rightarrow E\ltimes\}} \tag{19}$$

Fig. 6. Rules for 2-implication evidence

by our inference algorithm, we need evidence that a given assertion set logically implies another assertion set. Such evidence can be built using a number of constructs (more might be added) whose inference rules are listed in Fig. 6.

Here (13) says that any set of 2-assertions logically implies a smaller set, while (14) allows us to discard (replace by the empty set) 2-assertions that are vacuously true, and (15) allows us to discard 2-assertions whose consequent are constants.

Derivations can be combined "horizontally" by (16) which easily can be generalized to take an arbitrary number of arguments, and "vertically" by (17) which as a special case (when $\Theta_1' = \Theta_2$) has the "standard" transitivity rule.

The rule (18) allows us to lift simplications on antecedents to simplifications on 2-assertions, and expresses that 2-implication is contravariant in the antecedent; here ι ranges over evidence (described later) for logical implication: if $\vdash \iota \ : \ \phi \ \Rightarrow_1 \ \phi'$ then ϕ logically implies ϕ' (that is, whenever $s \models \phi$ then also $s \models \phi'$). For example, for the precondition Θ_0 computed above, one can easily verify that

$$\begin{array}{l} true \wedge \big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \Rightarrow \texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY} \\ true \wedge \neg\big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \Rightarrow \neg\big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \end{array}$$

and hence rule (18), together with rule (16), allows us to simplify Θ_0 to

$$\begin{array}{l} \Theta_0' = \big\{ \ \big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \Rightarrow \texttt{IN\_1\_DAT}\ltimes, \neg\big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \Rightarrow \texttt{OUT\_0\_DAT}\ltimes, \\ \qquad \big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big)\ltimes \big\}. \end{array}$$

Complex consequents can be decomposed using (19). For example, we can split the last consequent of Θ_0' to reach the final precondition of the code segment:

$$\begin{array}{l} \Theta_0'' = \big\{ \ \big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \Rightarrow \texttt{IN\_1\_DAT}\ltimes, \neg\big(\texttt{IN\_1\_RDY} \ \wedge \neg \texttt{OUT\_0\_RDY}\big) \Rightarrow \texttt{OUT\_0\_DAT}\ltimes, \\ \qquad \texttt{IN\_1\_RDY}\ltimes, \texttt{OUT\_0\_RDY}\ltimes \big\}. \end{array}$$

Evidence for Necessary Precondition. Recall that we need, for the rules (6) and (11) and (12), evidence ν such that if $\vdash\ \nu\ :\ \phi\ \overset{C}{\Leftarrow}\ \phi'$ then whenever $s\ [\![C]\!]\ s'$ and $s' \models \phi'$ then also $s \models \phi$. To build such evidence, we use one construct for each language construct, with one extra to make a shortcut when the command does not modify the 1-assertion. Two typical rules are listed below:

$$\frac{\vdash\ \nu_1\ :\ \phi_1\ \overset{C_1}{\Leftarrow}\ \phi \qquad \vdash\ \nu_2\ :\ \phi_2\ \overset{C_2}{\Leftarrow}\ \phi}{\vdash\ \mathsf{CondNPC}(\nu_1, \nu_2, B)\ :\ (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)\ \overset{\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2}{\Leftarrow}\ \phi} \quad (20)$$

$$\frac{\vdash\ \nu\ :\ \phi_0\ \overset{C}{\Leftarrow}\ \phi \qquad \vdash\ \iota_0\ :\ \phi_0 \wedge B\ \Rightarrow_1\ \phi \qquad \vdash\ \iota_1\ :\ \phi' \wedge \neg B\ \Rightarrow_1\ \phi}{\vdash\ \mathsf{WhileNPC}(\nu, \iota_0, \iota_1)\ :\ \phi\ \overset{\textbf{while } B \textbf{ do } C}{\Leftarrow}\ \phi'} \quad (21)$$

Evidence for Logical Implication. We have two kinds of evidence:

1. rules that resemble axiomatization of propositional logic;
2. the evidence $\mathsf{CheckLI}(\phi, \phi')$ which says that "a decision procedure has verified that ϕ logically implies ϕ'".

3.4 Manipulating 2-Assertions

It is often useful to transform a set of 2-assertions into a set which is "simpler" and which satisfies certain properties; for example, to analyze a **while** loop (or a method call), all (modified) consequents in the postcondition must be identifiers. Ideally, we would like the result to be equivalent to the original (like Θ_0' to Θ_0 in the previous example), but often it will be strictly stronger (as is Θ_0'' when $(E_1 \wedge \neg E_2)\ltimes$ is decomposed into $E_1\ltimes$ and $E_2\ltimes$). Hence[4] our overall approach does in general *not* calculate the *weakest* precondition.

We have written an algorithm that transforms a set of 2-assertions Θ' into a more manageable form, as may be required by the command C for which Θ' is the postcondition, while producing evidence that the result is at least as strong as the original. The algorithm as input also takes a set X, to be thought of as the identifiers that are modified by C. The algorithm returns τ, and also Θ_u (assertions whose *consequents* are unmodified) and Θ_n, such that $\vdash\ \tau\ :\ \Theta_n \cup \Theta_u\ \Rightarrow_2\ \Theta'$ and

- all array expressions inside Θ_n and Θ_u are identifiers; thus there are no occurrences of Z or $H\{A_0\ :\ A\}$ which may be introduced by rule (4) but hamper readability;
- all assertions in Θ_n are of the form $\phi \Rightarrow w\ltimes$ or of the form $\phi \Rightarrow h[A]\ltimes$, as is required if C is a **while** or **for** loop;
- if $\phi \Rightarrow E\ltimes \in \Theta_u$ then $\mathrm{fv}(E) \cap X = \emptyset$;
- if $\phi \Rightarrow h[A]\ltimes \in \Theta_n$ then $\mathrm{fv}(A) \cap X = \emptyset$;
- if $\phi_1 \Rightarrow E\ltimes \in \Theta_n$ and $\phi_2 \Rightarrow E\ltimes \in \Theta_n$ then $\phi_1 = \phi_2$.

3.5 Generating Evidence

For any command and postcondition, it is possible to compute a precondition, together with evidence that the resulting triple is indeed semantically sound. To help with that,

[4] Another reason is the approximation needed to efficiently handle loops.

we need an algorithm $\mathbf{NpcEvdGen}$ generating evidence for necessary precondition. For the nonlooping constructs, such an algorithm is straightforward to write, but for a **while** loop a *precise* analysis involves guessing an invariant; we expect that we might be able to use some of the emerging tools for finding loop invariants even though our perspective is dual.

We shall first present a *nondeterministic* algorithm $\mathbf{EvdGen}$ such that for all commands C (without **while** and **for**) and postconditions Θ, a call $\mathbf{EvdGen}(C,\Theta)$ returns evidence η such that $\vdash\ \eta\ :\ \{\Theta_0\}\ C\ \{\Theta\}$ for some Θ_0. Below we list the possible actions of $\mathbf{EvdGen}$, the applicability of which depend on the form of C and/or Θ.

Decompose Postcondition. The enabling condition is that Θ has at least two elements. Let Θ'_1,Θ'_2 be nonempty disjoint sets such that $\Theta = \Theta'_1 \cup \Theta'_2$. For each $i \in \{1..2\}$, recursively call $\mathbf{EvdGen}(C,\Theta'_i)$ to produce η_i such that $\vdash\ \eta_i\ :\ \{\Theta_i\}\ C\ \{\Theta'_i\}$ for some Θ_i. Define $\eta = \mathsf{UnionE}(\eta_1,\eta_2)$; we thus have $\vdash\ \eta\ :\ \{\Theta_1 \cup \Theta_2\}\ C\ \{\Theta\}$.

Push Through Postcondition. The enabling condition is that $\mathrm{fv}(\Theta) \cap X = \emptyset$ where X is such that $\vdash\ \mu\ :\ C\ \mathsf{mods\_only}\ X$ for some μ. Then we can define $\eta = \mathsf{NotModE}(\mu,\Theta)$ and achieve $\vdash\ \eta\ :\ \{\Theta\}\ C\ \{\Theta\}$.

Push Through Consequent of Postcondition. The enabling condition is that Θ is a singleton $\{\phi' \Rightarrow E\ltimes\}$, and that $\mathrm{fv}(E) \cap X = \emptyset$ with X such that $\vdash\ \mu\ :\ C\ \mathsf{mods\_only}\ X$ for some μ. Let $\nu = \mathbf{NpcEvdGen}(C,\phi')$. There thus exists ϕ with $\vdash\ \nu\ :\ \phi\ \stackrel{C}{\Leftarrow}\ \phi'$. Now define $\eta = \mathsf{ConseqNotModE}(\mu,\nu,E)$ and get $\vdash\ \eta\ :\ \{\phi \Rightarrow E\ltimes\}\ C\ \{\Theta\}$.

Syntax-Directed Actions. Two typical cases are as follows:

If $C = x := A$ then $\mathbf{EvdGen}(C,\Theta)$ returns $\mathsf{AssignE}(\Theta,x,A)$.

If $C =$ **if** B **then** C_1 **else** C_2, and Θ is a singleton $\{\phi' \Rightarrow E\ltimes\}$, we for each $i \in \{1,2\}$ recursively call $\mathbf{EvdGen}(C_i,\Theta)$ to produce η_i such that $\vdash\ \eta_i\ :\ \{\Theta_i\}\ C_i\ \{\Theta\}$ for some Θ_i, and call $\mathbf{NpcEvdGen}(C,\phi')$ to compute ν such that for some ϕ we have $\vdash\ \nu\ :\ \phi\ \stackrel{C}{\Leftarrow}\ \phi'$. We then return $\eta = \mathsf{CondE}(\eta_1,\eta_2,\nu,B)$ which by the typing rules satisfies the desired $\vdash\ \eta\ :\ \{\Theta_0\}\ C\ \{\Theta\}$ for some Θ_0.

Properties of $\mathbf{EvdGen}$. The precondition Θ_0 for an assignment statement $x := A$ depends only on the postcondition Θ, but not on the kind of evidence that was chosen; we will always have $\Theta_0 = \Theta[A/x]$. However, we do *not* have a similar result for conditionals $C =$ **if** B **then** C_1 **else** C_2: for example, if E is not modified by C then $\mathbf{EvdGen}(C,\Theta)$ may either produce evidence of the form $\mathsf{CondE}(\eta_1,\eta_2,\nu,B)$ whose type has a precondition containing an assertion with B as consequent, or evidence of the form $\mathsf{ConseqNotModE}(\mu,\nu,E)$ whose type does not have that property.

We thus need to restrict the non-determinism present in the definition of $\mathbf{EvdGen}$. For conditionals, syntax-directed evidence CondE should only be generated when the postcondition is a singleton whose consequent has been modified. For other constructs, we may be free either to split the postcondition or to apply the syntax-directed rules directly. The advantage of the former is that then the evidence provides fine-grained information about which preconditions come from which postconditions. The advantage of the latter is that then the evidence becomes more compact.

3.6 For Loops

We shall introduce 3 extra evidence constructs:

$$\eta ::= \ldots \mid \mathsf{ForAsWhileE}(\eta, \mu) \mid \mathsf{ForPolyE}(\ldots) \mid \mathsf{InstantiateE}(\eta, \mu, A)$$

Here ForAsWhileE just analyzes a **for** loop as a **while** loop; this involves splitting an assertion $\phi \Rightarrow h[A]\ltimes$ into $\phi \Rightarrow h\ltimes$ and $\phi \Rightarrow A\ltimes$ and thus we lose any information about individual array elements.

We shall now present a method, first given in [2] which contains further motivation and examples, that in certain cases allows us to reason about individual array elements. To do so in a finite way, we need the concept of *polymorphic identifiers*. Those may occur in pre/post conditions but never in commands; we shall use u to range over them. We shall extend $\models \{\Theta\}\ C\ \{\Theta'\}$ to cover the case where Θ and/or Θ' contains a polymorphic identifier u: then $\models \{\Theta\}\ C\ \{\Theta'\}$ holds iff $\models \{\Theta[c/u]\}\ C\ \{\Theta'[c/u]\}$ holds for all constants c. We have the inference rule

$$\frac{\vdash \eta : \{\Theta\}\ C\ \{\Theta'\} \qquad \vdash \mu : C\ \mathsf{mods\_only}\ X \qquad \mathrm{fv}(A) \cap X = \emptyset}{\vdash \mathsf{InstantiateE}(\eta, \mu, A) : \{\{A\ltimes\} \cup \Theta[A/u]\}\ C\ \{\Theta'[A/u]\}}$$

which is applicable not just when C is a **for**-loop. We shall design ForPolyE such that

$$\vdash \mathsf{ForPolyE}(\ldots) : \{\Theta\}\ \mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ C\ \{h[u]\}$$

if certain requirements, to be motivated and detailed below, are fulfilled. Let μ with $\vdash \mu : C\ \mathsf{mods\_only}\ X$ where $q, m \notin X$, and a polymorphic identifier u, be given.

Index Sets. There must exist a set of arithmetic expressions, $\{A_j \mid j \in J\}$ with $\mathrm{fv}(A_j) \cap X = \emptyset$, such that for all array assignments $h := H$ in C there exists j and A such that $H = h\{A_j : A\}$ (and thus the assignment may be written $h[A_j] := A$).

Linearity. We shall assume, as is very often the case for practical applications, that each A_j is a linear function in q. That is, there exists integer constants (or identifiers not in X) $b_j \neq 0$ and k_j such that A_j is given by $b_j q + k_j$. Then for each $j \in J$, we define

$$A'_j = \frac{u - k_j}{b_j}$$
$$\phi_j = (u - k_j) \bmod b_j = 0 \ \wedge\ u - k_j \geq b_j \ \wedge\ u - k_j \leq m b_j$$

with the intention that whereas A_j computes an index value from the iteration number, A'_j computes the iteration number from the index value while ϕ_j denotes the set of index values, as formalized by the following properties (to be suitably quantified):

1. if $c = [\![A_j]\!]_{[s|q \rightarrow i]}$ for $i \in \{1 \ldots s(m)\}$ then $[\![A'_j[c/u]]\!]_s = i$.
2. $s \models \phi_j[c/u]$ iff $c \in \{[\![A_j]\!]_{[s|q \rightarrow i]} \mid i \in \{1 \ldots s(m)\}\}$.

Local Preconditions. For all $j \in J$, there must exists η_j and Θ_j such that

1. $\vdash \eta_j : \{\Theta_j\}\ C\ \{h[A_j]\ltimes\}$ with $u \neq \mathrm{fv}(\Theta_j)$
2. $\mathrm{fv}(\Theta_j) \cap X \subseteq \{h\}$
3. if h occurs in Θ_j it is in a context of the form $h[A]$ where for all $j' \in J$, all $i, i' \in \{1 \ldots s(m)\}$, all stores s: if $[\![A]\!]_{[s|q \mapsto i]} = [\![A_{j'}]\!]_{[s|q \mapsto i']}$ then $i \leq i'$.

Requirement 2 excludes loop-carried dependencies in the body such as $h[q] := x; x := y$ in which case $h[1]$ depends on the initial value of x while $h[2]$, $h[3]$,... depends on the initial value of y. Requirement 3 is designed to exclude loop-carried dependencies *within* the array h; it is possible to list some cases that are easily checkable and which each is a *sufficient condition* for this requirement to hold:

1. when J is a singleton $\{j\}$, and the only occurrence of h in Θ_j is in the context $h[bq + k]$ where $b \geq b_j$ and $b \geq 1$ and $k \geq k_j$;
2. when $b_j = 1$ for all $j \in J$, and if h occurs in some Θ_j it is in a context of the form $h[A]$ with A of the form $q + c$ where c satisfies:

$$\forall j \in J :\ c \geq k_j \text{ or } c \leq k_j - m.$$

Both conditions will accept a loop body containing (only) $h[q] := h[q + 1]$ and reject a loop body containing $h[q] := h[q - 1]$.

We are now ready to construct the precondition Θ, as the union of

BOUND $\{\mathit{true} \Rightarrow m\ltimes\}$
INDEX $\{\mathit{true} \Rightarrow w\ltimes \mid w \in \cup_{j\in J}\mathrm{fv}(A_j) \setminus \{q\}\}$
OUTSIDE $\{\bigwedge_{j\in J} \neg\phi_j \Rightarrow h[u]\ltimes\}$
UPDATED for each $j \in J$, the set $\mathsf{add}^{\wedge}_{\phi_j}(\Theta_j[A'_j/q])$.

Here BOUND ensures that the two runs agree on the number of iterations, while INDEX ensures that the two runs agree on which indices are updated. For an index that might not be updated, the two runs must agree on the original value, as expressed by OUTSIDE. But for an index that may be updated, we apply the computed preconditions, as expressed by UPDATED.

Example. As in [2], we can analyze a **for** loop whose body swaps[5] $h[q]$ and $h[q + m]$ and where we therefore have $J = \{1, 2\}$, $A_1 = q$, $A_2 = q + m$, $b_1 = b_2 = 1$, $k_1 = 0$, and $k_2 = m$. We compute $\phi_1 = u \geq 1 \wedge u \leq m$, $\phi_2 = u - m \geq 1 \wedge u - m \leq m$, $A'_1 = u$, and $A'_2 = u - m$; we also get $\Theta_1 = \{h[q + m]\ltimes\}$ and $\Theta_2 = \{h[q]\ltimes\}$. The abovementioned sufficient condition 2 amounts to the 4 claims

$$0 \geq 0 \text{ or } 0 \leq 0 - m \qquad 0 \geq m \text{ or } 0 \leq m - m$$
$$m \geq 0 \text{ or } m \leq 0 - m \qquad m \geq m \text{ or } m \leq m - m$$

which are all easily verified. Hence we may generate the expected precondition

$$\{\ m\ltimes,\ (u < 1 \vee u > 2m) \Rightarrow h[u]\ltimes,$$
$$1 \leq u \leq m \Rightarrow h[u + m]\ltimes,\ m + 1 \leq u \leq 2m \Rightarrow h[u - m]\ltimes\ \}.$$

[5] Since each position participates in at most one swap there is no loop-carried dependency.

4 Machine-Checked Evidence and Soundness Overview

We now discuss how the evidence constructors of the previous section are represented and proven sound in Coq. Our technical report [4] provides detailed correctness proofs for all the evidence constructors. At the time of writing, we have completed the corresponding formalization of the proofs in Coq for assignments, conditionals, arrays, polymorphic tuples and almost all of the **for** loop, and we do not anticipate problems completing the remaining soundness proofs (remainder of **for** plus **while**).

4.1 Representation of Evidence

Our representation of evidence is based on deep embeddings of the language and the logic in Coq, using type-respecting categories of variables (e.g. SkalVar), expressions Expr (separated into arithmetic, boolean and array expressions AExpr, BExpr, HExpr), and Commands. Based on these definitions, we define expression evaluation and the operational semantics in direct correspondence to the definitions in Section 3; for example, Opsem s C t means that command C transforms state s into state t.

A 2-assertion is made up of a BExpr and an Expr:

Definition TwoAssn :=prod BExpr Expr.

Abbreviating the type of lists of 2-assertions as TwoAssns, we introduce the inductive type of pre/postconditions as

Inductive assns :=
| Assns : TwoAssns → TwoAssns → assns
| APoly : (AExpr → assns)→ assns.

Here, the first constructor carries a precondition/postcondition pair and will be used for standard triples. The second constructor allows the assertions to be parametrized by a shared variable, and will be required for implementing for-loops.

Evidence takes the form of an inductive proposition with constructors corresponding to the rules in Figures 4 and 5.

Inductive TEvid (X: list SkalVar) : Command → assns → Prop :=
| TSkipE ... | TAAssignE ... | TCondE ...
...

For example, the constructor for TAAssignE,

| TAAssignE : ∀ Θ x A, TEvid X (Assign x (AExp A)) (Assns(TwoAssnsSubstA Θ x A) Θ)

is a direct translation of rule (3), where TwoAssnsSubstA Θ x A represents the substitution (code omitted) of arithmetic expression A for x in the 2-assertion Θ.

Evidence for conditionals (rule 6) is translated similarly; the rule takes three explicit arguments of evidence type, one for each possible outcome of the branch, and one for the necessary precondition.

| TCondEN: ∀ {Θ_1 ϕ E Θ_2 Θ' C1 C2 ϕ_0}
(η_1: TEvid X C1 (Assns Θ_1 [(ϕ, E)])) (η_2: TEvid X C2 (Assns Θ_2 [(ϕ, E)]))
B (ν : NPCEvid X ϕ_0 (Cond B C1 C2) ϕ),
andIntoTheta Θ_1 B ++ andIntoTheta Θ_2 (NotExpr B) ++ [(ϕ_0, BExp B)] = Θ' →
allVarsIn (BFv ϕ_0) X =true → TEvid X (Cond B C1 C2) (Assns Θ' [(ϕ, E)])

We have evidence forms declared for all syntax but while and for loops, and we have also declared constructors for all the rules mentioned in Fig. 5. There are similar inductive types and accompanying soundness proofs for NPC, 1-implication, 2-implication and expression equivalence. Rules representing decision-procedure-validated evidence such as CheckLI are currently axiomatically admitted, although future work will aim to verify the output of decision procedures using methods similar to [6].

4.2 Soundness

The soundness of constructed evidence terms rests on the interpretation of 2-assertions

```
Definition twoSatisfies(s1 s2:State) (asn: TwoAssn) := let (φ, E) = asn in
  BEval φ s1 = Some true → BEval φ s2 = Some true → (Eval E s1 = Eval E s2)
```

which corresponds to the informal definition given earlier in the paper.[6] The interpretation is naturally extended to lists of 2-assertions (where Forall is the universal quantification over list elements, taken from the Coq library):

```
Definition twoAssnsInterpretation (s1 s2: State) (a:TwoAssns) : Prop :=
    Forall (twoSatisfies s1 s2) a.
```

We are now ready to model the definition (Sect. 3.2) of the predicate $\models \{\Theta\}\ C\ \{\Theta'\}$.

```
Definition validHoareTriple (X: list SkalVar) C (asns: assns): Prop :=
match asns with
| Assns pre post ⇒ ∀ s s' t t',
        (∀ x, In x X → ∃ v1, lookup s x = Some v1 ∧ ∃ v2, lookup s' x = Some v2) →
        Opsem s C t → Opsem s' C t' →
        twoAssnsInterpretation s s' pre → twoAssnsInterpretation t t' post
| ...
end.
```

Our soundness statement (Theorem 2 in Sect. 3) is formulated as follows.

```
Theorem soundness : ∀ X C asns, TEvid X C asns → (validHoareTriple X C asns).
```

Whenever we apply this soundness result to a defined concrete piece of evidence, the type of the resulting construction explicitly witnesses the validity of the triple. For example, applying the soundness result to a piece of evidence named assignEvid

```
Definition evidSound := soundness _ _ _ assignEvid.
```

guarantees our intended security property in that it yields a term of type

```
validHoareTriple X cmd (Assns pre post).
```

Performing the proof of the soundness theorem constitutes a major engineering task even once all the definitions are set up correctly. As an indication of the effort, the Coq code has around 2300 lines of soundness proof and only 590 lines of trusted definitions; it formalizes a manual correctness proof in [4] which is about 5.5 pages from about 610 lines of LaTeX source.

[6] This direct correspondence between informal and formal definitions is crucial, as the formal definitions introduced here form part of the trusted code base of our system.

In order to address the concern that certificates may be unacceptably large, we have also explored an implementation of the precondition generator inside Coq, in the style of proof-by-reflection. Our precondition generator consists of a Gallina function

```
Fixpoint generatePrecondition (c:Command) (post:TwoAssns)
    (X: list SkalVar): (TwoAssns * list SkalVar) := ...
```

with defining clauses that closely resemble Figures 4 and 5. The soundness theorem

```
Theorem generatePreconditionEvidence: ∀ C Y X Z post pre, allVarsIn X Y = true→
(pre, X) = generatePrecondition C post Z → TEvid Y C (Assns pre post).
```

expresses that any result (pre, X) from a call to generatePrecondition yields evidence for the triple made up from the inferred precondition and the supplied command and post-condition. Thus, evidence terms need not be explicitly constructed, as valid evidence can be constructed automatically for any command and postcondition.

5 Evaluation

We summarize our initial experience in applying our SIFL deduction engine for evidence generation. The SIFL precondition generation algorithm supports assignments, conditionals, arrays, **for** and **while** loops, polymorphic flow contracts, and procedure calls. We tested this implementation on procedures from a collection of embedded applications (an Autopilot, a Minepump, a Water Boiler monitor, and a Missile Guidance system – all developed outside of our research group), and a collection of small programs that we developed ourselves to highlight common array idioms that we discovered in information assurance applications. Approximately 6-15 procedures from each of these examples were selected due to having the richest control flow and loop structures. The security-critical sections to be certified from code bases in this domain are often relatively small, *e.g.*, roughly 1000 LOC (non-comment lines of code) for a Rockwell Collins high assurance guard and 3000 LOC for an (undisclosed) device certified by Naval Research Labs researchers [7]. The average LOC per procedure in our examples is 22. In this evaluation, we focused on running the tool in a mode that infers information flow contracts. For each procedure P, and each output variable w, the algorithm analyzes the body wrt. post-condition $w \ltimes$. All experiments were run under JDK 1.6 on a 2x2.6 GHz Quad Core Intel Xeon Mac Pro with 32 GB of RAM.

We were most interested in evaluating (a) the size of the generated evidence, (b) the number and structure of assertions in the inferred precondition (for the purpose of minimizing its size), (c) the time required for the algorithm to infer a contract and generate evidence, and (d) the time required for Coq to type-check (i.e., establish the correctness) of the evidence. In the subsequent paragraphs, we shall summarize the outcomes for the first 3 measures (see [4] for detailed data and evidence outputs for all examples we considered); for (d), no procedure required more than 3 seconds.

In contrast to other proof-carrying code applications such as mobile code, contract size is not as significant an issue in our context since contracts are not being transmitted or checked at run-time. Instead, the focus is on leveraging contracts for greater assurance in the certification process. We consider three different metrics for the size of evidence: the number of evidence constructors, the total number of Coq abstract syntax

tree (AST) nodes (which captures the size of assertion expressions and program ASTs in the evidence), and the number of bytes in the text file holding the evidence. For the full version of the mailbox example of Section 2 which has 24 LOC, the generated evidence includes 318 evidence constructors, 3729 Coq AST nodes, and 89 KB of text. For a slightly longer example (30 LOC) from the autopilot codebase that has one of the longest analysis run-times, the presence of four conditions leads to larger generated evidence due to more conditional preconditions: 326 evidence constructors, 19461 Coq AST nodes, and 211 KB of text. Our current contract representation includes the syntax tree of the program as well as the 2-assertions generated between each command. Thus, many program expressions are repeated numerous times across the evidence structure for a procedure. There is significant opportunity to optimize the size by, e.g., common subexpression elimination.

Initial examination of the generated preconditions identified a number of minimization opportunities (e.g., simplifying assertions of the form $\mathit{true} \wedge \phi \Rightarrow E\ltimes$ to $\phi \Rightarrow E\ltimes$ or removing a 2-assertion when it is implied by another within the precondition). In the mailbox example, there are 52 AST nodes in the precondition for `OUT_0_DAT` without minimization and 28 when minimization applied. Our current strategy for expressing minimization includes many fine-grained reasoning steps using the rules of Figure 6. Thus, the number of evidence constructors in the `OUT_0_DAT` derivation actually increases from 25 to 83. Potentially, this can be reduced by making the Coq checker "smarter" by having it do more manipulation of logical expressions without direct instruction from the evidence generator.

The time required for processing a procedure ranged from 3 to 140 secs. As previously discussed, the correctness of minimization of assertions was validated with calls to SMT solvers. The repeated calls to the SMT solvers were the dominating factor in the time required to infer contracts, and we have not yet devoted any effort to optimize this. Many of the minimization steps can be implemented using simple syntactic checks, and we are in the process of implementing and proving correct a minimizer in Coq that will allow us to dramatically reduce the number of SMT solver calls. Experimental results from our earlier work [1] in which we used only syntactic scans to minimize showed that inference for almost all the procedures could be completed in less than a second. Our approach is compositional which greatly aids scalability when considering the overall time requirements for a complete application.

6 Related Work

Bergeretti and Carré [8] present a compositional method for inferring and checking dependencies among variables in SPARK programs. That approach is flow-sensitive, unlike most security type systems [9] that rely on assigning a security level ("high" or "low") to each variable. Chapman and Hilton [10] present an approach, now implemented in the latest SPARK release, for extending SPARK information flow contracts with lattices of security levels and enhancing the SPARK Examiner accordingly.

Agreement assertions (inherently flow-sensitive) were introduced in [11] and later extended in [5] to introduce conditional agreement assertions (for a heap-manipulating language). In [1] that approach was applied to the (heap-free) SPARK setting and

worked out extensively, with an algorithm for computing loop invariants and with reports from an implementation; then arrays were handled in subsequent work [2].

Our evidence-checker is an example of the proof-carrying code paradigm [12]; it is *foundational* [13] in that the rules used by the evidence checker are themselves proved sound with a machine-checked proof. Although the original PCC generated proofs mainly via type-checking, more recently the PCC paradigm has been extended to policies of mobile code concerning resource consumption and information flow [14,15]. Certificate generation for such systems was obtained by formalizing static analyses (refined type systems or abstract interpretation frameworks) either directly at the level of virtual machine code, or by providing compiler-mediated interpretations of appropriate high-level analyses [16,17,18]. Wildmoser and Nipkow developed verified VCGens for bytecode for a deeply embedded assertion language for bytecode [19]. In the context of abstract-interpretation-based PCC, Besson et al. [20] employed certificates in the form of strategies for (re-)recomputing fixed points at the consumer side.

Techniques for reducing the size of evidence representations using oracles [21] and small witnesses [22] developed into *reflective* PCC [15] where the evidence checker (or even a partial evidence reconstruction algorithm) is implemented in the tactic language of the proof assistant, and proved sound by the principle of reflection. We have found (cf. Section 4) that our current evidence checker permits this approach.

In addition to direct justification of static analyses wrt. operational semantics, several of the above-mentioned formalizations employ program logics and/or VCGen's as intermediate representations. In order to employ these for the verification of information flow, the relational nature of information flow security must be taken into account, either by direct use of relational program logics [23], or by suitable encodings [24,25] in nonrelational logics based on the idea of self-composition [26,27].

In contrast to (typically not foundationally validated) efforts to relax baseline security policies to more permissive notions (e.g. declassification), our conditional information flow analysis aims to improve the precision and trustworthiness of static analysis results for the baseline policy, in the setting of an existing domain-specific tool flow methodology. Dufay, Felty, and Matwin [28] and Terauchi and Aiken [29] provide tool support for the verification of noninterference based on self-composition. In [28], the Krakatoa/Why verification framework is extended by variable-agreement assertions and corresponding loop annotations, and emits verification conditions in Coq that are typically interactively discharged by the user. In [29], information inherent in type systems for noninterference is exploited to limit the application of the program-duplication to smaller subphrases, obtaining self-composed programs that are better amenable to fully automated state-space-exploring techniques. Neither system produces foundationally validated and independently checkable artefacts of evidence relating the source program to user-level specifications, and it is at present unclear whether either could be extended to support conditional information flow policies.

7 Conclusions and Future Work

By implementing an evidence emitting algorithm and an associated evidence checking framework in Coq, we have provided a solution that allows developers to work at source level to specify/check rich information flow contracts while still enabling

machine-checked proofs that source code implementations conform to contracts. This work puts in place a crucial element of our larger vision for end-to-end security assurance – namely, the ability eventually to leverage our other work on formally verified compilers [30] to provide a tool chain that enables us to prove that *deployed executable code* conforms to complex information flow policies stated as source-level contracts. Our next steps include adding a higher-level information flow policy specification language on top of our framework, enlarging the subset of SPARK that our tools can handle, and engineering a connection to the CompCert verified compiler stack [31]. We are also working with our industrial partners to evaluate our tools on additional examples.

Acknowledgements. This work is funded in part by the Air Force Office of Scientific Research (FA9550-09-1-0138), and by Rockwell Collins whose engineers David Hardin, David Greve, Ray Richards, and Matt Wilding we would like to thank for feedback on earlier versions of this work. We would also like to thank the anonymous referees for useful and detailed comments.

References

1. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.A.: Specification and Checking of Software Contracts for Conditional Information Flow. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Heidelberg (2008)
2. Amtoft, T., Hatcliff, J., Rodríguez, E.: Precise and Automated Contract-Based Reasoning for Verification and Certification of Information Flow Properties of Programs with Arrays. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 43–63. Springer, Heidelberg (2010)
3. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006). IEEE Press (2006)
4. Amtoft, T., Dodds, J., Zhang, Z., Appel, A., Beringer, L., Hatcliff, J., Ou, X., Cousino, A.: A certificate infrastructure for machine-checked proofs of conditional information flow (2012), `http://santos.cis.ksu.edu/papers/Amtoft-al-POST12/`
5. Amtoft, T., Banerjee, A.: Verification condition generation for conditional information flow. In: 5th ACM Workshop on Formal Methods in Security Engineering (FMSE 2007), pp. 2–11. George Mason University, ACM (2007)
6. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011)
7. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: 13th ACM Conference on Computer and Communications Security (CCS 2006), pp. 346–355 (2006)
8. Bergeretti, J.F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Transactions on Programming Languages and Systems 7, 37–61 (1985)
9. Volpano, D.M., Smith, G.: A Type-Based Approach to Program Security. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 607–621. Springer, Heidelberg (1997)
10. Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. ACM SIGAda Ada Letters XXIV, 39–46 (2004)
11. Amtoft, T., Banerjee, A.: Information Flow Analysis in Logical Form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004)

12. Necula, G.C.: Proof-carrying code. In: POPL 1997, pp. 106–119. ACM Press (1997)
13. Appel, A.W.: Foundational proof-carrying code. In: LICS 2001. IEEE Computer Society (2001)
14. Sannella, D., Hofmann, M., Aspinall, D., Gilmore, S., Stark, I., Beringer, L., Loidl, H.W., MacKenzie, K., Momigliano, A., Shkaravska, O.: Mobile resource guarantees. In: van Eekelen, M.C.J.D. (ed.) Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming (TFP 2005), Intellect, pp. 211–226 (2007)
15. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS Proof Carrying Code Infrastructure. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 1–24. Springer, Heidelberg (2008)
16. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic Certification of Heap Consumption. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 347–362. Springer, Heidelberg (2005)
17. Albert, E., Puebla, G., Hermenegildo, M.V.: Abstraction-Carrying Code. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 380–397. Springer, Heidelberg (2005)
18. Barthe, G., Pichardie, D., Rezk, T.: A Certified Lightweight Non-interference Java Bytecode Verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
19. Wildmoser, M., Nipkow, T.: Asserting Bytecode Safety. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 326–341. Springer, Heidelberg (2005)
20. Besson, F., Jensen, T.P., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. Theor. Comput. Sci. 364, 273–291 (2006)
21. Necula, G.C., Rahul, S.P.: Oracle-based checking of untrusted software. In: POPL 2001, pp. 142–154 (2001)
22. Wu, D., Appel, A.W., Stump, A.: Foundational proof checkers with small witnesses. In: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2003), pp. 264–274. ACM (2003)
23. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) POPL 2004, pp. 14–25. ACM (2004)
24. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: CSF 2007, pp. 233–248. IEEE Computer Society (2007)
25. Beringer, L.: Relational Decomposition. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 39–54. Springer, Heidelberg (2011)
26. Darvas, Á., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
27. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004), pp. 100–114. IEEE Computer Society (2004)
28. Dufay, G., Felty, A.P., Matwin, S.: Privacy-Sensitive Information Flow with JML. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 116–130. Springer, Heidelberg (2005)
29. Terauchi, T., Aiken, A.: Secure Information Flow as a Safety Problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
30. Appel, A.W.: Verified Software Toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011)
31. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54 (2006)