PROTOTYPING PROOF CARRYING CODE

Martin Wildmoser, Tobias Nipkow
Institut für Informatik, Technische Universität München
wildmosm@in.tum.de, nipkow@in.tum.de

Gerwin Klein
National ICT Australia, Sydney
gerwin.klein@nicta.com.au

Sebastian Nanz *
Yale University, Department of Computer Science
nanz@cs.yale.edu

Abstract We introduce a generic framework for proof carrying code, developed and mechanically verified in Isabelle/HOL. The framework defines and proves sound a verification condition generator with minimal assumptions on the underlying programming language, safety policy, and safety logic. We demonstrate its usability for prototyping proof carrying code systems by instantiating it to a simple assembly language with procedures and a safety policy for arithmetic overflow.

1 Introduction

Proof Carrying Code (PCC), first proposed by Necula and Lee [11] [12], is a scheme for executing untrusted code safely. Fig. 1 shows the architecture of a PCC system. The code producer is on the left, the code receiver on the right. Both use a verification condition generator (VCG) that relies on annotations in the program to reduce the program to a logic formula. The logic used in annotations and proof is the *safety logic,* the property that is shown about the program is the *safety policy.*

It is the responsibility of the producer to generate the annotations and a proof for the formula the VCG constructs. They are then transmitted to the code receiver who again runs the VCG and uses a proof checker to verify that the proof indeed fits the formula produced by the VCG. Proof checking is much simpler and more efficient than proof searching. The framework for PCC systems we present in this paper concentrates on the safety critical receiver side. It has the following two main purposes and contributions: safety of the system and prototyping new safety logics. Proof checker, VCG,

* supported in part by NSF grant CCR-0208618.

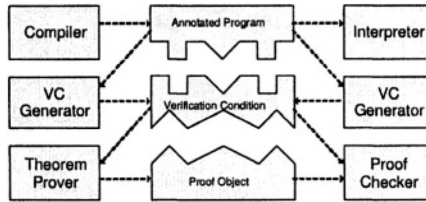

Figure 1. PCC Architecture

and safety logic constitute the trusted code base of the PCC system. Proof checkers are relatively small standard components of many logical frameworks. The VCG on the other hand is large (several thousand lines of C code in current PCC systems [6] [13]) and complex (it handles annotations, produces complex formulae, and contains parts of the safety policy). Our framework contains a VCG with a formal proof of safety, mechanically checked in the theorem prover Isabelle/HOL [16]. The VCG is not restricted to any particular machine language, safety policy, or safety logic. Additionally to the correctness of VCG and proof checker, we need the safety logic to be sound. As a recent bug [9] in the SpecialJ system [6] shows, this is not trivial. It is not even immediately clear, what exactly a safety logic must satisfy to be sound. Our framework makes the underlying assumptions on machine, policy, and logic explicit. It also makes a simple, formally clear statement what it means for a safety logic to be sound: if the formula produced by the VCG is derivable in the safety logic, the program must be safe according to the safety policy. The framework reduces the workload for showing soundness of a safety logic by giving sufficient conditions. Since the VCG is directly executable and the framework reasonably easy to instantiate, it provides a good platform for trying out, tuning, and analysing different safety policies and logics for different target platforms.

Our approach is different from other work in the formal foundation of PCC by Appel et al. [1] [2] or Hamid et al. [7] in that it works with an explicit, executable, and verified VCG and not directly on the machine semantics or a type system. The focus of the framework is on aiding logical foundations of PCC as the one started by Necula and Schneck [14] and on encouraging the analysis of safety properties other than the much researched type and memory safety. Necula and Schneck [15] also present a framework for VCGs. They work with a small, trusted core VCG that can be extended by optimised plugins. We see our work as complementary to this development: the core VCG could be proven sound within our framework, the technique of using safe, optimised extensions can then be applied to that sound core. On a broader scale, our approach is related to other techniques that impose safety policies on machine code statically: Typed Assembly Language [10], Mobile Ressource Guarantees [3] or Java Bytecode Verification [8].

There are four levels in our PCC systems. The first level, the PCC framework (§2), provides generic features and minimal assumptions. The second level is the platform (§3). Platform designers can provide a concrete instantiation of the framework with respect to a specific programming language, safety policy, and safety logic. The third level is the code producer who can now write and certify programs based on the in-

stantiated framework. We show this by certifying a concrete program in §4. Finally, also in §4, we show how code receivers can check certified code within the framework. The formalization in this paper was carried out in Isabelle/HOL, so we inherit some of Isabelle's syntax. Most of the notation is familiar from functional programming and standard mathematics, we only mention a few peculiarities. Consing an element x to a list xs is written as $x\#xs$. Infix @ is the append operator, and $xs\ !\ n$ selects the n-th element from the list xs. The type $T1 \Rightarrow T2$ is the space of total functions from $T1$ to $T2$, and we frequently use the polymorphic option type **datatype** *'a option* = *None* | *Some 'a* to simulate partiality in HOL, a logic of total functions: *None* stands for an undefined value, *Some x* for a defined value x.

2 Framework Definition

The components of a PCC system shown in Fig. 1 depend on three factors: programming language, safety policy, and safety logic. The programming language defines syntax and semantics for programs, the safety policy specifies the safety conditions programs must satisfy, and the safety logic provides a formal notation and a derivation calculus for proving these conditions. Our framework consists of skeletons and requirements for these three components and uses them to define and verify a generic VCG.

2.1 Program Semantics

Our framework expects the semantics of the underlying programming language in form of a function *effS*:: *'prog* ⇒ ((*'pos* × *'mem*) × (*'pos* × *'mem*)) *set* which relates runtime states of a program to their immediate successor states. States are tuples (p,m) of type *'pos* × *'mem*, where p denotes the current position in the control flow graph and m is the machine's memory, e.g., heap, stack and registers. Since *'prog*, *'pos* and *'mem* are type variables the representation of programs, positions and memory can be instantiated as one likes.

2.2 Safety Logic

To specify and prove properties about programs we use a safety logic.

$\llcorner True \lrcorner$ :: *'form* $\quad$ $\llcorner \bigwedge \lrcorner$ :: *'form list* ⇒ *'form*
$\llcorner False \lrcorner$ :: *'form* $\quad$ $\llcorner \Longrightarrow \lrcorner$ :: *'form* ⇒ *'form* ⇒ *'form*
$\models$ :: *'prog* ⇒ (*'pos* × *'mem*) ⇒ *'form* ⇒ *bool*
$\vdash$ :: *'prog* ⇒ *'form* ⇒ *bool*

Every structure having constants for the truth values $\llcorner True \lrcorner$ and $\llcorner False \lrcorner$, operators for conjunction $\llcorner \bigwedge \lrcorner$ and implication $\llcorner \Longrightarrow \lrcorner$, judgements for validity $\models$ and provability $\vdash$ of formulae can be employed as a safety logic as long as it respects the assumptions below. These assumptions only concern the semantics of the logical connectives. How formulae or their proofs look like and what they mean, is left open. This depends on how *'form*, $\vdash$ and $\models$ get instantiated.

assumptions
semTrueF: $\Pi,s \models \llcorner True \lrcorner$ $\qquad$ *semFalseF*: $\neg\ \Pi,s \models \llcorner False \lrcorner$
semConj: $\Pi,s \models \llcorner \bigwedge \lrcorner\ Fs\ =\ (\forall\ F \in set\ Fs.\ \Pi,s \models F)$

semImpl: $\Pi,s \models (A \Longrightarrow_s B) = (\Pi,s \models A \longrightarrow \Pi,s \models B)$

2.3 Safety Policy

Our framework expects the safety policy to be defined by means of the safety logic. We assume that for each position p in a program Π a safety formula ***safeF*** Π p expresses the conditions we want to hold whenever we reach p at runtime.

safeF:: *'prog* ⇒ *'pos* ⇒ *'form*

In addition we assume that a safety logic formula ***initF*** Π characterises all states under which a program Π can be started.

initF:: *'prog* ⇒ *'form*

Now we can give a generic notion of safety for programs: A program is safe, if all states *(p,m)* it reaches from some initial state are safe. That is *(p,m)* satisfies the safety formula ***safeF*** Πp, which the platform dedicates to position p.

$$isSafe\ \Pi = (\forall p_0\ m_0\ p\ m.\ \Pi,(p_0,m_0) \models initF\ \Pi \land ((p_0,m_0),(p,m)) \in (effS\ \Pi)^* \longrightarrow \Pi,(p,m) \models safeF\ \Pi\ p)$$

2.4 The Verification Condition Generator

The VCG is the core of our PCC framework. It takes a program Π and generates a formula ***vc*** in the safety logic. If this formula is provable, then the program is safe at runtime, i.e., ***isSafe*** Π holds. The structure of the ***vc*** is determined by the program's control flow graph, which is a directed graph. Nodes denote program positions and can be marked with annotations. Edges point to successor positions and are marked with branch conditions. Fig. 2 shows a control flow graph. It can be seen as an abstraction of the assembly program *E,* which compares two variables *X* and *Y* and eventually sets *X* to the maximum of these two.

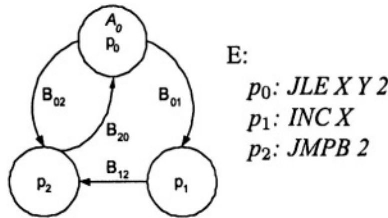

Figure 2. control flow graph

Parameters. To extract parts of the control flow graph and to express the semantics of programs by means of safety logic formulae and manipulations on these, our framework requires various parameter functions:

anF:: *'prog* ⇒ (*'pos* ⇒ *'form option*)
succsF:: *'prog* ⇒ *'pos* ⇒ (*'pos* × *'form*) *list*
wpF:: *'prog* ⇒ *'pos* ⇒ *'pos* ⇒ (*'form* ⇒ *'form*)
domC:: *'prog* ⇒ *'pos list* *ipc*:: *'prog* ⇒ *'pos*

With *anF* we access the annotations; $anF\ \Pi\ p$ returns *Some A* if position p in Π is annotated with A, otherwise *None*. Function *succsF* yields the edges of the control flow graph. Given a position p in a program Π the expression $succsF\ \Pi\ p$ yields a list of pairs (p',B) where p' is a possible successor of p and B is the branch condition for the edge from p to p'. The branch condition B is a safety logic formula that characterises the situations when p' is accessible from p. For example if Π jumps *from* p to either p' or p'' depending on a condition C, then $succsF\ \Pi\ p$ should return something like $[(p',C),(p'',\neg\ C)]$. To reflect the semantics within the safety logic we use *wpF*, a function for computing (weakest) preconditions. The formula $wpF\ \Pi\ p\ p'\ Q$ is expected to characterises those states (p,m) that have successor states (p',m') satisfying Q. The function *domC* is expected to yield the code domain of a program; this is a list of all positions with instructions. Finally *ipc* is used to determine the initial program counter.

Definition. The *vcg* constructs the verification condition out of so called inductive safety formulae $isafeF\ \Pi\ p$, which we generate individually for each position p in a program Π. We call a state (p,m) *inductively safe* if it satisfies the inductive safety formula for p, i.e., $\Pi,(p,m) \models isafeF\ \Pi\ p$. Fig. 3 defines $isafeF\ \Pi\ p$. The wellformedness

$$
\begin{array}{l}
wf\ \Pi \longrightarrow \\
isafeF\ \Pi\ p = if\ p \in set\ (domC\ \Pi) \\
then\ \llcorner\bigwedge\lrcorner [safeF\ \Pi\ p]\ @ \\
\qquad (case\ (anF\ \Pi\ p) \\
\qquad of\ None \Rightarrow (map(\lambda(p',B).\ B \llcorner\Longrightarrow\lrcorner\ wpF\ \Pi\ p\ p'\ (isafeF\ \Pi\ p')) \\
\qquad\qquad\qquad (succsF\ \Pi\ p)) \\
\qquad \mid Some\ A \Rightarrow [A]) \\
else\ \llcorner False\lrcorner
\end{array}
$$

Figure 3. **Construction of inductive safety formulae**

constraint $wf\ \Pi$ ensures that every loop in Π has at least one annotation; otherwise the recursion of *isafeF* would not terminate.
When p lies outside the code domain $domC\ \Pi$ we must never reach it at runtime. We express this formally by returning the unsatisfiable formula $\llcorner False\lrcorner$ in this case. For positions p within the code domain the inductive safety formula guarantees the safety formula $safeF\ \Pi\ p$. In addition, if there is an annotation A at p, we conjoin the safety formula with A. For example in program E from Fig. 2, we have the annotation A_0 at p_0. Hence, we obtain $\llcorner\bigwedge\lrcorner [safeF\ E\ p_0, A_0]$ for $isafeF\ E\ p_0$.
If p is not annotated, we take an successor positions p' together with their branch conditions B and recursively compute the inductive safety formulae $isafeF\ \Pi\ p'$. Using the *wpF* operator we construct a precondition $wpF\ \Pi\ p\ p'\ (isafeF\ \Pi\ p')$. If this precondition holds for a state (p,m) with some successor (p',m'), then $isafeF\ \Pi\ p'$ holds for (p',m'). By constructing implications of the form $B \llcorner\Longrightarrow\lrcorner\ (wpF\ \Pi\ p\ p'\ (isafeF\ \Pi\ p'))$, we design the inductive safety formula $isafeF\ \Pi\ p$ such that all states satisfying the branch condition B for a particular successor p' also have to satisfy the precondition above. These implications are constructed for all pairs (p',B) we get from *succsF* $\Pi\ p$. For example the positions p_1 and p_2 are not annotated in E. Below are their in-

ductive safety formulae, where *safeF, wpF,* branch conditions and annotations are not expanded.

$$isafeF\ E\ p_1 = \bigwedge[\ safeF\ E\ p_1,\ B_{12} \Longrightarrow wpF\ E\ p_1\ p_2\ (\bigwedge[\ safeF\ E\ p_2, B_{20} \Longrightarrow wpF\ E\ p_2\ p_0\ (\bigwedge[safeF\ E\ p_0, A_0])])]$$

$$isafeF\ E\ p_2 = \bigwedge[safeF\ E\ p_2, B_{20} \Longrightarrow wpF\ E\ p_2\ p_0\ (\bigwedge[safeF\ E\ p_0, A_0])]$$

Executing a program Π with an inductively safe state *(p,m)* produces a trace of inductively safe states until we reach an annotated position p'. The state (p',m') under which we reach this position, is safe and satisfies the annotation. After this state the execution could become unsafe. However, this does not happen if all successor states of (p',m') are again inductively safe. This observation guides the construction of the verification condition $vcg\ \Pi$, which we show in Fig. 4. The verification condition *vcg*

$$\begin{aligned} vcg\ \Pi = \bigwedge(&[initF\ \Pi \Longrightarrow (isafeF\ \Pi\ (ipc\ \Pi))] @ \\ &map(\lambda p_a.\ \bigwedge(map(\lambda(p',B).\ \bigwedge[isafeF\ \Pi\ p_a, B] \Longrightarrow \\ &\qquad\qquad wpF\ \Pi\ p_a\ p'\ (isafeF\ \Pi\ p')) \\ &\qquad (succsF\ \Pi\ p_a))) \\ &[p_a \in domC\ \Pi.\ anF\ \Pi\ p_a \neq None]) \end{aligned}$$

Figure 4. Verification Condition Generator

Π demands two things: First, all initial states must satisfy the first inductive safety formula $isafeF\ \Pi\ (ipc\ \Pi)$. Second, for every annotated position p_a the inductive safety formula $isafeF\ \Pi\ p_a$ and the branch condition B for all successors p' of p_a must guarantee the precondition $wpF\ \Pi\ p_a\ p'\ (isafeF\ \Pi\ p')$. This ensures that the transitions out of annotated positions leads to inductively safe successor states. As discussed above, this proves the safety of Π. For example *vcg E* would have the following form:

$$\begin{aligned} \bigwedge[&initF\ E \Longrightarrow isafeF\ E\ p_0, \\ &\bigwedge[isafeF\ E\ p_0, B_{01}] \Longrightarrow wpF\ E\ p_0\ p_1\ (isafeF\ E\ p_1), \\ &\bigwedge[isafeF\ E\ p_0, B_{02}] \Longrightarrow wpF\ E\ p_0\ p_2\ (isafeF\ E\ p_2)] \end{aligned}$$

The first conjunct expresses that initial states are inductively safe. Note that $ipc\ E = p_0$. Since p_0 has two successors p_1 and p_2, which are accessible if B_{01} resp. B_{02} hold, we have two further conjuncts. One requires us to show that all states satisfying the inductive safety formula for p_0 and the branch condition B_{01} can only have successor states that satisfy the inductive safety formula for p_1. The other is analogous for p_2.

Soundness. The VCG is sound if for every well formed program Π a provable verification condition $\Pi \vdash vcg\ \Pi$ guarantees program safety, i.e., $isSafe\ \Pi$.

theorem $wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \longrightarrow isSafe\ \Pi$

We have proven this theorem in Isabelle based on the requirements our PCC framework has on its parameter functions. In these assumptions, which we discuss in detail in the appendix, we require that *succsF* approximates the control flow, that *wpF* yields proper preconditions and that the safety logic is correct.

3 Framework Instantiation

In this section we instantiate the framework with a simple assembly language (SAL). We show how HOL can be instantiated as safety logic and demonstrate it on a safety policy that prohibits type errors and arithmetical overflows.

3.1 A Simple Assembly Language

SAL provides instructions for arithmetics, pointers, jumps, and procedures. We distinguish two kinds of addresses. Locations, which we model as natural numbers, identify memory cells, whereas positions identify places in a program. We denote positions as pairs *(pn,i),* where *i* is the relative position inside the procedure with name *pn*.

types ***loc = nat, pname = nat, pos = pname × nat***

datatype ***instr =SET loc nat | ADD loc loc | SUB loc loc | MOV loc loc |***
JMPL loc loc nat | JMPB nat | CALL loc pname | RET loc | HALT

The instructions manipulate states of the form *(p,(m,e)),* where *p* denotes the program counter and *(m,e)* the system memory. Since pairs associate to the right in Isabelle/HOL we often leave out the inner brackets and write *(p,m,e)* to denote a state with program counter *p*, main memory *m* and environment *e*.

types ***SALstate = pos × (loc ⇒ tval) × env***

The program counter stores the position of the instruction that is executed next. The main memory *m,* which maps locations to typed values, stores all the data a program works on. We have three kinds of values: Uninitialised values having type *ILLEGAL,* natural numbers *NAT n,* and positions *POS (pn,i).*

datatype ***tval = ILLEGAL | NAT nat | POS pos***

The environment *e* tracks information about the run of a program. It contains a call stack *cs e,* which lists the memory contents and times under which currently active procedures have been called, and a history *h e,* which traces the values of program counters.

record ***env = cs :: (nat × (loc ⇒ tval)) list***
h :: pos list

To update a field *x* in a record *r* with an expression *E* we write ***r(|x:=E|),*** to access it we write *x r*. We use the environment like a history variable in Hoare Logic; it provides valuable information for annotations written as predicates on states. We can describe states by relating them to former states or refer to system resources,e.g., the length of *h e* is a time measure.
A SAL program is a list of procedures, which consist of a name *pname* and a list of possibly annotated instructions. Annotations are predicates on states.

types ***SALform = SALstate ⇒ bool***
SALprocedure = pname × ((instr × (SALform option)) list)
SALprogram = SALprocedure list

To access instructions we write *cmd* Π ***p***, which gives us *Some ins* if Π has an instruction *ins* at *p*, or *None* otherwise.

3.2 SAL Semantics

SAL Instructions do the following: *SET X n* initialises *X* with *NAT n*. *ADD X Y* and *SUB X Y* add and subtract the values at *X* and *Y* storing the result in *X*. *MOV X Y* interprets the values of *X* and *Y* as addresses *a* and *b;* it copies the value at *a* to *b*. *JMPL X Y t* jumps t positions forward if the value at *X* is less than the value at *Y;* otherwise just one. *JMPB t* jumps *t* positions backwards. *CALL X pn* jumps into procedure *pn* leaving the return address in *X*. *RET X* leaves a procedure and returns to the address expected in *X*. Finally, *HALT* stops execution. In the instantiation of *effS* we formalise these effects.

$$\textit{effS}\ \Pi = \{(s,s') \mid \textit{step}\ \Pi\ s = \textit{Some}\ s'\}$$

We do this with an auxiliary expression $\textit{step}\ \Pi\ (p,m,e)$, which yields *Some* (p',m',e') if the instruction $\textit{cmd}\ \Pi\ p$ exists and yields the successor state (p',m',e'). For example *ADD X Y* updates *X* with $(m\ X) \oplus (m\ Y)$, which is *ILLEGAL* if either *X* or *Y* contains no number or *NAT (a+b)* if *m X = NAT a* and *m Y = NAT b*. In addition the history is augmented with the current program counter. $\textit{cmd}\ \Pi\ (pn,i) = \textit{Some ADD X Y} \longrightarrow$
$\textit{step}\ \Pi\ ((pn,i),m,e) =$
$= \textit{Some}\ ((pn,i+1),m[X \mapsto (m\ X) \oplus (m\ Y)],e(\!|h := (h\ e) @ (pn,i)|\!))$

The other instructions can be handled in a similar fashion.

3.3 SAL Safety Policy

In initial states the program counter is *(0,0),* the main memory only contains uninitialised values and the environment *e* has an empty history and a copy of the initial memory on its call stack.

$$\textit{initF}\ \Pi = \lambda(p,m,e).\ p=(0,0) \land \forall X.\ m\ X=\textit{ILLEGAL} \land h\ e=[] \land cs\ e=[(0,m)]$$

States are safe if the current instruction respects type safety and does not produce an arithmetic overflow, that is numerical results are less than *MAX*. Example:

$$\textit{cmd}\ \Pi\ p = \textit{Some}\ (\textit{ADD X Y}) \longrightarrow$$
$$\textit{safeF}\ \Pi\ p = \lambda(p,m,e).\ (\exists n.\ (m\ X) \oplus (m\ Y) = \textit{NAT}\ n \land n \leq \textit{MAX})$$

For the sake of brevity we skip the remaining instructions.

3.4 SAL Safety Logic

By identifying assertions with HOL predicates, we instantiate a shallow embedded safety logic in Fig. 5. The valididy judgment $\models$ is directly defined by applying a predicate to a state. The argument Π is only there to be compatible with the generic signature of the framework. We define the provability judgment $\vdash$ directly by means of the semantics. This enables us to prove verification conditions with Isabelle/HOL's inference rules using various tactics and decision procedures as tools. Alternatively we could also use a deep embedding and define $\vdash$ with an explicit proof calculus, possibly tailored to the programming language and its safety policy. This means more effort, but could pay off in form of shorter proofs or higher degree of automation in

proof search. However, this paper focuses on the framework and we rather keep the instantiation simple. According to $\vdash$ a formula F is provable iff it holds for all states

$$\lfloor True \rfloor = \lambda s.\ True \qquad \bigwedge fs = \lambda s.\ \forall F \in set\ fs.\ F\ s$$
$$\lfloor False \rfloor = \lambda s.\ False \qquad A \Longrightarrow B = \lambda s.\ A\ s \longrightarrow B\ s$$
$$\Pi,s \models F = F\ s \qquad \Pi \vdash F = \forall s.\ s \in isafe_{\Box}\ \Pi \longrightarrow \Pi,s \models F$$

Figure 5. **Safety Logic for SAL.**

in $isafe_{\Box}\ \Pi$. The inductively defined set $isafe_{\Box}\ \Pi$ contains all initial states and states that originate from a computation where all states are inductively safe.

$$\Pi,(p,m) \models initF\ \Pi \longrightarrow (p,m) \in isafe_{\Box}\ \Pi$$

$$(p,m) \in isafe_{\Box}\ \Pi \wedge \Pi,(p,m) \models isafeF\ \Pi\ p \wedge \Pi,(p',m') \models isafeF\ \Pi\ p' \wedge$$
$$((p,m),(p',m')) \in effS\ \Pi \longrightarrow (p',m') \in isafe_{\Box}\ \Pi$$

This constraint on states simplifies proofs and shortens annotations, because one can derive properties of a state from the fact that this state can be reached at runtime by only traversing inductively safe intermediate states.

3.5 Instantiating VCG helper functions

The instantiations of *anF, domC* and *ipc* are straightforward. More interesting are *wpF* and *succsF*. For the instantiation of *wpF* we use **λ-abstraction** to postpone substitution of formulae to the verification stage. Example:

$$cmd\ \Pi\ p = Some\ (ADD\ X\ Y) \longrightarrow wpF\ \Pi\ p\ p'\ Q =$$
$$\lambda(p,m,e).\ let\ m'=m[X \mapsto (m\ X) \oplus (m\ Y)];\ e'=e(|h:=(h\ e)@p|)\ in\ Q\ (p',m',e')$$

We compute the effect of *ADD X Y* on some symbolic state *(p,m,e)* and demand that *Q* holds for the resulting state. Finally, we have a glimpse of the *succsF* instantiation. Here, we chose *JMPL* as example:

$$cmd\ \Pi\ (pn,i) = Some\ (JMPL\ X\ Y\ t) \longrightarrow succsF\ \Pi\ (pn,i) =$$
$$[((pn,i+t),\lambda(p,m,e).\ \exists n\ n'.\ m\ X=NAT\ n \wedge m\ Y=NAT\ n' \wedge n<n' \wedge p=(pn,i)),$$
$$((pn,i+1),\lambda(p,m,e).\ \exists n\ n'.\ m\ X=NAT\ n \wedge m\ Y=NAT\ n' \wedge \neg n<n' \wedge p=(pn,i))]$$

The constraint on the program counter *p=(pn,i)* in the branch conditions helps to apply system invariants. These are properties that hold for all states in $isafe_{\Box}\ \Pi$ irrespective of Π. For example $\lambda((pn,i),m,e).\ cmd\ \Pi\ (pn,i) = Some\ (RET\ X) \longrightarrow (\exists k\ m'\ css.\ cs\ e = (k,m')\#css \wedge cmd\ \Pi\ (h\ e)!k = (CALL\ pn\ X))$ is a system invariant. It says that for the call time *k* of the current procedure the history *h e* records the position of a *CALL* instruction.

3.6 Verifying Procedures

Procedure proofs should be modular. Code with procedure calls should only depend on these procedure's specifications (the annotations at entry and exit positions) and not on their code. For example $\lambda(p,m,e).\ m\ X = (\overleftarrow{m}\ e)\ X \oplus (NAT\ 1)$ might be the postcondition of a procedure that increments a location *X*. Here we use $\overleftarrow{m}\ e = snd\ (hd\ (cs\ e))$ to reconstruct the memory at call time.This procedure could be called from a position where *X* is *NAT 5*. The programmer expects that after the procedure *X* is *NAT 6* and could write this into the annotation at the return point. In the verification

condition we would have to prove that this follows from the procedure's postcondition. However $\lambda(p,m,e).\ m\ X = (\overleftarrow{m}\ e)\ X \oplus (NAT\ 1) \Longrightarrow (\lambda(p,m,e).\ m\ X = NAT\ 6)$ is not provable. The information that *X* has been *NAT 5* at the procedures entry point is missing. We cannot add this information into the postcondition, otherwise we loose modularity. A way out is to pack call context dependent information into branch conditions, which *succsF* computes individually for each successor. If a procedure returns to $(pn',i'+1)$ and (pn',i') is annotated with *Ac* we can construct the branch condition $\lambda(p,m,e).\ Ac\ (\overleftarrow{pc}\ e, \overleftarrow{m}\ e, \overleftarrow{e}\ e)$, which claims that $(\overleftarrow{pc}\ e, \overleftarrow{m}\ e, \overleftarrow{e}\ e)$, the state at call time, satisfies the annotation *Ac*. Note that $\overleftarrow{pc}$ and $\overleftarrow{e}$, the position and environment at call time, can be defined analogously to $\overleftarrow{m}$. Since branch conditions are added to inductive safety formulas, we now obtain a provable formula: $(\bigwedge [\lambda(p,m,e).\ m\ X = (\overleftarrow{m}\ e)\ X \oplus (NAT\ 1)\ ,\ \lambda(p,m,e).\ (\overleftarrow{m}\ e)\ X = NAT\ 5]) \Longrightarrow (\lambda\ (p,m,e).\ m\ X = NAT\ 6)$. Call context dependent branch conditions involve some technicalities for the definition and verification of *succsF*. However, they fit neatly into our concept of a generic VCG. We achieve modular procedure proofs although our VCG has no notion of procedures at all.

4 Case Study: Overflow Detection

4.1 Motivating Example for Overflow Detection

The exemplary safety policy expressed the definition of *safeF* in §3.3 has two aspects: First, type safety is needed as a general property to ensure that SAL programs never get stuck. Second, the safety formula demands that the result of arithmetic operations does not exceed *MAX*, thus preventing overflows. Consider the following program fragment: [*CALL P CHECK, ADD B C*]
It might be part of an application that tries to add a credit stored as a natural number in memory location *C* to a balance in *B*—for example as part of a load transaction of a smart card purse. Before executing the addition, a procedure *CHECK* is called to ensure that the new balance in *B* is less than *MAX;* if it does, the credit in *C* will be set to zero and thus the balance remains the same as before. Special care has to be taken in the implementation of *CHECK:*

[*SET M MAX*, *SET H 0*, *ADD H B*, *ADD H C*, *JMPL H M 2*, *SET C 0*, *RET P*]

M represents the maximum balance considered for the application. *H* should contain $B + C$ after the second *ADD* statement. If the check $B + C < M$ fails, the credit is set to zero; otherwise it is left unchanged. Even this simple example contains an implementation flaw: there could be an overflow in *H*. And the flaw is not merely theoretical: in the case of a silent overflow as in Java it would lead to debiting the purse instead of crediting.

4.2 Annotated SAL Program

Fig. 6 shows the corrected and annotated version of our example. The main procedure and *CHECK* are now identified with 0 and 1. For better readability we write instruction/annotation pairs of the form *(ins, None)* as just *ins* and *(ins, Some A)* as *{A} ins.*

$$
\begin{aligned}
OD = [&(0,[SET\ B\ b_0,\ SET\ C\ c_0,\\
&\quad \{\lambda(p,m,e).\ m\ B = NAT\ b_0 \wedge m\ C = NAT\ c_0\}\\
&\quad CALL\ P\ 1,\\
&\quad \{\lambda(p,m,e).\ m\ B = NAT\ b_0 \wedge (\exists c.\ m\ C = NAT\ c \wedge\\
&\qquad\qquad c = (if\ b_0 + c_0 < MAX\ then\ c_0\ else\ 0))\}\\
&\quad ADD\ B\ C,\ HALT\ ])\\
&(1,[\ \{\lambda(p,m,e).\ m\ P = POS\ (incA\ (\overleftarrow{pc}\ e)) \wedge (\exists b.\ m\ B = NAT\ b) \wedge\\
&\qquad\qquad (\exists c.\ m\ C = NAT\ c) \wedge (\forall X.\ X \neq P \longrightarrow m\ X = \overleftarrow{m}\ e\ X)\}\\
&\quad SET\ M\ MAX,\ SUB\ M\ C,\ JMPL\ B\ M\ 2,\ SET\ C\ 0,\\
&\quad \{\lambda(p,m,e).\ (\forall X.\ X \neq C \wedge X \neq M \wedge X \neq P \longrightarrow m\ X = \overleftarrow{m}\ e\ X) \wedge\\
&\qquad\qquad (\exists b\ c\ c'.\ m\ B = NAT\ b\ \wedge m\ C = NAT\ c \wedge \overleftarrow{m}\ e\ C = NAT\ c' \wedge\\
&\qquad\qquad c = (if\ b + c' < MAX\ then\ c'\ else\ 0))\}\\
&\quad RET\ P\ ])]
\end{aligned}
$$

Figure 6. **Corrected and annotated program OD.**

Before execution of *CALL P 1*, the memory positions *B* and *C* contain the numbers b_0 and c_0. The annotation for *ADD B C* states that the value of *C* may have changed according to the condition $b_0 + c_0 < MAX$.
Inside the *CHECK* procedure we first set the memory location *M* to the maximum balance. The annotation states that location *P* stores the proper return address for the procedure: *incA* $(\overleftarrow{pc}\ e)$ represents the program counter of the calling procedure incremented by one. Furthermore the annotation states that there are natural numbers in both *B* and *C,* and that all memory locations except *P* are the same as in the caller. The following statements require no annotations, only the exit point of the procedure *RET P* does: it states that all values except for those in *C*, *M,* and *P* are unchanged, that there are natural numbers in both *B* and *C,* and that the new value of *C* will be changed to zero if the new balance exceeds the maximum balance.

4.3 Verification Condition

In Fig. 7 we show the part of the verification condition that is generated for the return from procedure *CHECK.* In general we get as many parts (conjuncts) as there are paths between annotated positions. That means the size of verification conditions is linear to the number of positions if all branch positions are annotated. The example demonstrates again how the VCG works. On the top-level the conditions for the annotated program positions are conjoined; the fragment refers to position *p=(1,4)* of our program, *1* stands for the procedure *CHECK* and *4* for the line number with the statement *RET P.* There is only one successor p'*=(0,2),* which is the statement *ADD B C.* Therefore the conjunction over the list of all successors collapses to one element. The verification condition fragment shown in Fig. 7 results from the expression $\llcorner\bigwedge\lrcorner$ [*isafeF OD (1,4), B*] $\llcorner\Longrightarrow\lrcorner$ ***wpF*** *OD (1,4) (0,2) (isafeF OD (0,2))* where *B* is the branch condition of *succsF OD (1,4)*. Numbers 1–4 in Fig. 7 correspond to the assumption of the implication, numbers 5–6 to the conclusion. *isafeF OD (1,4)* results in $\llcorner\bigwedge\lrcorner$[*safeF OD (1,4), Ae*] (compare Fig. 3), where *safeF OD (1,4)* corresponds to 1 and the annotation *Ae,* e.g., *anF OD (1,4) = Some Ae,* corresponds to 2. The branch

$$
\begin{array}{ll}
 & \bigwedge [\\
1 & \quad \bigwedge [\ \lambda(p,m,e).\ \exists pn'\ i'.\ m\ P = POS\ (pn',i'+1) \wedge \\
 & \qquad (\exists k\ m'\ cl\ css.\ cs\ e = (k, m')\#cl\#css \wedge (pn',i') = (h\ e)!k), \\
2 & \qquad \lambda(p,m,e).\ (\forall X.\ X \neq C \wedge X \neq M \wedge X \neq P \longrightarrow m\ X = \overleftarrow{m}\ e\ X) \wedge \\
 & \qquad (\exists b\ c\ c'.\ m\ B = NAT\ b \wedge m\ C = NAT\ c \wedge \overleftarrow{m}\ e\ C = NAT\ c' \wedge \\
 & \qquad c = if\ b + c' < MAX\ then\ c'\ else\ 0], \\
3 & \quad \bigwedge [\ \lambda(p,m,e).\ m\ P = POS\ (0,2) \wedge p=(1,4), \\
4 & \qquad \lambda(p,m,e).((\lambda(p,m,e).\ m\ B = NAT\ b_0 \wedge m\ C = NAT\ c_0)\ (\overleftarrow{pc}\ e,\overleftarrow{m}\ e,\overleftarrow{e}\ e))] \\
 &] \\
5 & \Longrightarrow \bigwedge [\ \lambda(p,m,e).\ \exists n.\ (m\ B)\oplus(m\ C) = NAT\ n \wedge n \leq MAX, \\
6 & \qquad \lambda(p,m,e).\ m\ B = NAT\ b_0 \wedge \exists c.\ m\ C = NAT\ c \wedge \\
 & \qquad c = if\ b + c_0 < MAX\ then\ c_0\ else\ 0]
\end{array}
$$

Figure 7. **Fragment of the verification condition.**

condition B for $RET\ P$ appears in 3 and 4, and consists of $\bigwedge [\lambda\ (p,m,e).\ m\ P = POS\ (0,2) \wedge p=(1,4), \lambda(p,m,e).\ Ac\ (\overleftarrow{pc}\ e, \overleftarrow{m}\ e, \overleftarrow{e}\ e)]$ where $\lambda(p,m,e).\ Ac\ (\overleftarrow{pc}\ e, \overleftarrow{m}\ e, \overleftarrow{e}\ e)$ is the annotation of the call instruction, e.g., *anF OD (0,1) = Some Ac,* applied to the reconstructed state at the moment of the call, and P is the memory location of the return address. This shows again how the environment e enables us to reconstruct the call state $(\overleftarrow{pc}\ e, \overleftarrow{m}\ e, \overleftarrow{e}\ e)$ and how to transfer the information Ac of the call point to the return point. Note that this context-specific information is encoded into the branch condition B, which *succsF* computes individually for each successor. The annotation at the procedure's return point does not refer to a particular call point. Hence, the procedure and its verification are modular. The conclusion of the verification condition consists of the safety condition for *ADD* in 5 and its annotation in 6; together they form *isafeF OD (0,2).*

4.4 Code Producer and Consumer

The code producer can write annotated programs in Isabelle. To obtain the verification condition one can generate and execute ML code for the VCG [5] or use the simplifier to evaluate *vcg* Π. Proving the verification condition is supported by powerful proof tools and a rich collection of HOL theorems. For the example in Fig. 6 the simplifier and a decision procedure for presburger arithmetic suffice to prove the verification condition. For the client side Isabelle provides (compressed) proof terms and a proof checker [4]. Proofs are encoded as λ terms having a type that corresponds to the theorem they prove (Curry Howard Isomorphism). Proof Checking becomes a type checking problem, which can be handled by a small trusted program.

5 Conclusion

Our framework can be instantiated to various programming languages, safety policies, and safety logics. As long as the requirements of the framework are satisfied, one can directly apply our generic VCG and rely on its machine checked soundness proof. In our instantiation to SAL we show how HOL can be embedded as safety logic and how

this can be used to verify the absence of arithmetic overflows. Since HOL is very expressive, formulating complex assertions or safety policies is possible. Isabelle's code generator gives us an executable version of the VCG. Using the built in tools for proof search, proof terms and proof checking we can simulate producer and client activities. Before one embarks on a particular PCC implementation, one can build a prototype in our framework and prove the soundness of the safety logic. On our web page [19] we present more complex examples and instantiations of our framework. These include programs with pointer arithmetic or recursive procedures and safety policies about time and memory consumption of programs. Moreover we have instantiated a safety logic based on first order arithmetic in form of a deep embedding [18]. There, formulae are modelled as HOL datatype and can by analysed by other HOL functions. This enables us to optimise verification conditions after/during their construction. By now, we also have instantiated the PCC framework to a (downsized) version of the Java Virtual Machine [17]. For this we did not have to change the framework, thus we believe that our framework's formalisation and its requirements are reasonable, even for real life platforms.

References

[1] Appel, A. W. (2001). Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01),* pages 247–258.

[2] Appel, A. W. and Felty, A. P. (2000). A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00),* pages 243–253.

[3] Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W. (2003) A Resource-aware Program Logic for a JVM-like Language In Trends in Functional Programming, editor: S. Gilmore, Edinburgh

[4] Berghofer, S. and Nipkow, T. (2000). Proof terms for simply typed higher order logic. In *Theorem Proving in Higher Order Logics,* Springer LNCS vol. 1869, editors: J. Harrison, M. Aagaard

[5] Berghofer (2003). Program Extraction in simply-typed Higher Order Logic. In *Types for Proofs and Programs, International Workshop, (TYPES 2002),* Springer LNCS, editors: H. Geuvers, F. Wiedijk

[6] Colby, C., Lee, P., Necula, G. C., Blau, F., Plesko, M., and Cline, K. (2000). A certifying compiler for Java. In *Proc. ACM SIGPLAN conf. Programming Language Design and Implementation,* pages 95–107.

[7] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z. (2002). A syntactic approach to foundational proof-carrying code. In *Proc. 17th IEEE Symp. Logic in Computer Science,* pages 89–100.

[8] Klein, G. (2003). *Verified Java Bytecode Verification.* PhD thesis, Institut fur Informatik, Technische Universität München.

[9] League, C., Shao, Z., and Trifonov, V. (2002). Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Department of Computer Science, Yale University.

[10] Morrisett, G., Walker, D., Crary, K., and Glew, N. (1998). From system F to typed assembly language. In *Proc. 25th ACM Symp. Principles of Programming Languages,* pages 85–97. ACM Press.

[11] Necula, G. C. (1997). Proof-carrying code. In *Proc. 24th ACM Symp. Principles of Programming Languages,* pages 106–119. ACM Press.

[12] Necula, G. C. (1998). *Compiling with Proofs.* PhD thesis, Carnegie Mellon University.

[13] Necula, G. C. and Lee, P. (2000). Proof generation in the touchstone theorem prover. In McAllester, D., editor, *Automated Deduction — CADE-17,* volume 1831 of *Lect. Notes in Comp. Sci.,* pages 25–44. Springer-Verlag.

[14] Necula, G. C. and Schneck, R. R. (2002). A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In Voronkov, A., editor, *Proc.CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark,* volume 2392 of *Lect. Notes in Comp. Sci.,* pages 47–62. Springer-Verlag.

[15] Necula, G. C. and Schneck, R. R. (2003). A sound framework for untrustred verification-condition generators. In *Proc. IEEE Symposium on Logic in Computer Science (LICS03),* pages 248–260.

[16] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL – A Proof Assistant for Higher-Order Logic,* volume 2283 of *Lect. Notes in Comp. Sci.* Springer.

[17] Klein, G. and Nipkow, T. (2004) A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler *Technical Report,* National ICT Australia, Sydney

[18] Wildmoser, M. and Nipkow, T. (2004) Certifying machine code safety: shallow versus deep embedding. *TPHOLs 2004*

[19] VeryPCC website in Munich (2004),`http://isabelle.in.tum.de/verypcc/`.

Appendix: Requirements

Our PCC framework makes some assumptions on the functions it takes as parameters (cf. p.4). Based on these assumptions we prove the generic VCG correct. It is the task of the framework instantiator to make sure that the implementations of the parameter functions satisfy the requirements listed below. We have proven in Isabelle that these requirements hold for our instantiation to SAL. Hence, we have a PCC system for SAL with a mechanically verified trusted code base. Note that none of the requirements involves the safety policy *safeF.* Hence it is very easy to instantiate our framework to different safety policies.

Assumption *correctWpF* ensures that *wpF* computes proper preconditions. That is for every state *(p,m)* having a successor state (p',m'), we require that Q holds for (p',m') whenever $wpF\ \Pi\ p\ p'\ Q$ holds for *(p,m).* We require this property only for wellformed programs Π and for states in $isafe_{\Box}\ \Pi$, a set of states we introduce in §3.4.

assumption *correctWpF:*
$wf\ \Pi \wedge\ (p,m) \in isafe_{\Box}\ \Pi \wedge ((p,m),(p',m')) \in (effS\ \Pi) \wedge$
$\Pi,(p,m) \models (wpF\ \Pi\ p\ p'\ Q) \longrightarrow \Pi,(p',m') \models Q$

Although the set $isafe_{\Box}\ \Pi$ seems to complicate matters at a first sight, it simplifies the instantiator's job of proving the requirements. Only initial states and safe states originating from a safe execution must be considered. We can conclude information about these states from inductive safety formulae of previous states.

Assumption *correctIpc* demands that *ipc* and *initF* fit together:

assumption *correctIpc*: $\Pi,(p,m) \models \mathit{initF}\ \Pi \longrightarrow p = \mathit{ipc}\ \Pi$

In *succsF–complete* we assume that *succsF* covers all transitions of *effS* and yields branch conditions that hold whenever a particular transition is accessible. Again, this is only required for wellformed programs and states in $\mathit{isafe}_{\Box}\ \Pi$.

assumption *succsF-complete*:
$\mathit{wf}\ \Pi \wedge (p,m) \in \mathit{isafe}_{\Box}\ \Pi \wedge ((p,m),(p',m')) \in \mathit{effS}\ \Pi$
$\longrightarrow (\exists B.\ (p',B) \in \mathit{set}\ (\mathit{succsF}\ \Pi\ p) \wedge \Pi,(p,m) \models B)$

In *correctSafetyLogic* the safety logic's provability judgement is constrained such that provable formulae are guaranteed to hold for states in $\mathit{isafe}_{\Box}$.

assumption *correctSafetyLogic*:
$\Pi \vdash f \wedge (p,m) \in \mathit{isafe}_{\Box}\ \Pi \longrightarrow \Pi,(p,m) \models f$

Based on these assumptions we can prove that our VCG is sound. A provable verification condition gurantess safety of a program at runtime.

theorem *vcg-soundness*:
$[\![\ \mathit{wf}\ \Pi;\ \Pi \vdash \mathit{vcg}\ \Pi\ ]\!] \Longrightarrow \mathit{isSafe}\ \Pi$