Certifying Machine Code Safety: Shallow versus Deep Embedding

Martin Wildmoser, Tobias Nipkow

Technische Universität München, Institut für Informatik
{wildmosm|nipkow}@in.tum.de

Abstract. We formalise a simple assembly language with procedures and a safety policy for arithmetic overflow in Isabelle/HOL. To verify individual programs we use a *safety logic*. Such a logic can be realised in Isabelle/HOL either as shallow or deep embedding. In a shallow embedding logical formulas are written as HOL predicates, whereas a deep embedding models formulas as a datatype. This paper presents and discusses both variants pointing out their specific strengths and weaknesses.

1 Introduction

Proof Carrying Code (PCC), first proposed by Necula and Lee [14,15], is a scheme for executing untrusted code safely. It works without cryptography and without a trusted third party. Instead, it places the burden on showing safety on the code producer, who is obliged to annotate a program and construct a certificate that it adheres to an agreed upon safety policy. The code consumer merely has to check if the certificate—a machine-checkable proof—is correct. This check involves two steps: A verification condition generator (VCG) reduces the annotated program to a verification condition (VC), a logical formula that is provable only if the program is safe at runtime. Then a proof checker ensures that the certificate is a valid proof for the VC. If both VCG and proof checker work correctly, this scheme is tamper proof. If either the program, its annotations or the certificate are modified by an attacker, they won't fit or, if they still do, the resulting program would also be safe. Proof Checkers are relatively small standard components and well researched. The VCG is a different story. In early PCC systems it is large (23000 lines of C in [8]) and complex. The formulas it produces are usually influenced by the machine language, the safety policy and the safety logic. The machine language determines syntax and semantics of programs. These are considered safe if they satisfy the conditions the safety policy demands. The safety logic can serve multiple purposes. First, it provides a formal description language for machine states, which can be use to write annotations or to specify a safety policy. Second, it is used to express and prove verification conditions.

For some safety policies, such as checking that all instructions are used on proper arguments (type safety), a type system could play the role of the safety logic. VCG and proof checking could be replaced by automatic type inference. A typical example is Java Bytecode Verification, which is formally verified by now [12].

To handle more complex properties, for example checking that programs operate within their granted memory range (memory safety), type systems can be combined with a logic or extended to a logic like system [10,13,7]. Foundational proof carrying code tries prove safety directly in terms of the machine semantics [2,3], without a VCG or safety logic as an extra layer.
Our approach uses a VCG, but keeps it small and generic. We model this VCG as part of an Isabelle/HOL framework for PCC, which can be instantiated to various machine languages, safety policies and safety logics. The machine checked soundness proof we have for this VCG automatically carries over to the instantiations. One only has to show that the instantiation meets the requirements our framework makes explicit. None of these requirements touches the safety policy, which in turn can be replaced without disturbing any proof at all. In addition Isabelle/HOL supports the whole range of code producer and consumer activities. We can generate ML code [5] for our VCG and use Isabelle/HOL to produce and check proof objects for verification conditions [6].
By now we have instantiated various non trivial safety policies, such as constraints on runtime or memory consumption, and verified various example programs, including recursive procedures and pointer arithmetics [1]. In this paper we instantiate a simple assembly language (SAL) with a safety policy that prevents type errors and arithmetic overflows. Both are kept rather simple. This paper focuses on the safety logic, which can be embedded in Isabelle/HOL [16] either in shallow or deep style. In the first one models safety logic formulas as HOL predicates on states. The safety logic automatically inherits the infrastructure of the theorem prover such as its type system and tools for simplifying or deciding formulas. In the second one models formulas as a datatype and defines functions to evaluate or transform them. We discuss both variants and point out their specific strengths and weaknesses.

2 Execution Platform

Our simple assembly language (SAL) is a down sized version of TAL [13], which additionally has indirect jumps, multiple argument passing modes and an explicit distinction between registers and heap addresses. Since we are primarily interested in the safety logic and policy, we rather keep the programming language simple. However, with pointers and procedures SAL already includes major pitfalls of machine languages. We consider programs as safe if all instruction arguments have proper type and do not cause arithmetical overflows. Note that the latter involves reasoning about runtime values and demands an expressive annotation language. A simple type system does not suffice, because it can only express what types the results of an instruction or procedure have, not the relation between input and output values.

2.1 SAL Platform

In SAL 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 a procedure named pn.

types $loc = nat$, $pname = nat$, $pos = pname \times nat$

SAL has instructions for arithmetics, pointers, jumps and procedures.

datatype $instr = SET\ loc\ nat \mid ADD\ loc\ loc \mid SUB\ loc\ loc \mid MOV\ loc\ loc \mid JMPL\ loc\ loc\ nat \mid JMPB\ nat \mid CALL\ loc\ pname \mid RET\ loc \mid HALT$

These instructions, which we explain in §2.2, manipulate states of the form (p,m,e), where p denotes the program counter, m the memory and e the environment.

types $state = pos \times (loc \Rightarrow tval) \times env$

The program counter p is 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 distinguish three kinds of values: Uninitialised values $ILLEGAL$, natural numbers $NAT\ n$, and positions $POS\ (pn,i)$.

datatype $tval = ILLEGAL \mid NAT\ nat \mid POS\ pos$

The environment e tracks useful information about the run of a program. It is a record with two fields cs and h and equally named selector functions. To update a field x in a record r with an expression E we write $r(\!|x:=E|\!)$.

record $env = cs :: (nat \times (loc \Rightarrow tval))\ list$
$h :: pos\ list$

An environment e contains a call stack $cs\ e$, which lists the times and memory contents under which currently active procedures have been called, and a history $h\ e$, which traces the values of program counters. We use the environment like a history variable in Hoare Logic. It is not necessary for machine execution but valuable for reasoning about execution. 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.

```
OD = [
(0,[ {A0} SET B b0,
     {A1} SET C c0,
     {A2} CALL P 1,
     {A3} ADD B C,
     {A4} HALT ]),
(1,[ {A5} SET M MAX,
     {A6} SUB M C,
     {A7} JMPL B M 2,
     {A8} SET C 0,
     {A9} RET P ])]
```

Fig. 1. Sample Code

A program is a list of procedures, which consist of a name $pname$ and a list of possibly annotated instructions. With *'a option* we model partiality in Isabelle/HOL, a logic of total functions. It injects the new element $None$ into a given type *'a*.

'a option = *None* | *Some 'a*.

types
$proc = pname \times ((instr \times (form\ option))\ list)$
$prog = proc\ list$

For example Fig. 1 shows a program that safely credits the balance B of a smart card purse. A procedure checks whether $B + C$ exceeds MAX. If it does it set C to 0 thus preventing an overflow of the following ADD instruction. For better readability we write $\{A\}\ ins$ to denote an instruc-

tion/annotation pair (*ins*,*Some A*). Annotations are formulas in the safety logic and have type *form*. To access instructions we write *cmd Π p*, which gives us *Some ins* if a program *Π* has an instruction *ins* at *p*, or *None* otherwise. For example in Fig. 1 we get *cmd OD* (*0*,*2*) = *Some* (*CALL P 1*).

2.2 Program Semantics

To formalise the effects of SAL instructions, we use the transition relation *effS*.

effS Π = {((*p*,*m*,*e*),(*p′*,*m′*,*e′*)) | *step Π* (*p*,*m*,*e*) = *Some* (*p′*,*m′*,*e′*) }

For this small step semantics we use *step Π* (*p*,*m*,*e*), which yields *Some* (*p′*,*m′*,*e′*) if the instruction at *p* yields the successor state (*p′*,*m′*,*e′*). For example *ADD X Y* updates *X* with (*m X*)⌜+⌝(*m Y*), which is *NAT* (*x*+*y*) if *m X*=*NAT x* and *m Y*=*NAT y* and *ILLEGAL* otherwise. Here + is addition on naturals (no overflow) and ⌜ ⌝ lifts operators from natural numbers to typed values. Like all instructions *ADD* also extends the history *h e*. We formalise this using ↦ for function update and @ for list concatenation:

cmd Π (*pn*,*i*) = *Some* (*ADD X Y*) ⟶ *step Π* ((*pn*,*i*),*m*,*e*) =
Some ((*pn*,*i*+*1*),*m*(*X*↦*m X* ⌜+⌝ *m Y*),*e*(|*h*:=*h e*@[(*pn*,*i*)]|))

The transitions of *SUB X Y*, which subtracts two numbers, and *SET X n*, which intialises *X* with *NAT n* are similar; just replace ⌜+⌝ with ⌜−⌝ or change the update to *m*(*X*↦*NAT n*). The backwards jump *JMPB t* jumps *t* instructions backwards. The conditional jump *JMPL X Y t* expects numbers at *X* and *Y*. If the first number is less than the second it jumps *t* instructions forward, otherwise just one.

cmd Π (*pn*,*i*) = *Some* (*JMPB t*) ⟶
step Π ((*pn*,*i*),*m*,*e*) = *Some* ((*pn*,*i*−*t*),*m*,*e*(|*h*:=*h e*@[(*pn*,*i*)]|))

cmd Π (*pn*,*i*) = *Some* (*JMPL X Y t*) ∧ *m X* = *NAT x* ∧ *m Y* = *NAT y* ⟶
step Π ((*pn*,*i*),*m*,*e*) = *Some* ((*pn*,*i*+*if x*<*y then t else 1*),*m*,*e*(|*h*:=*h e*@[(*pn*,*i*)]|))

The procedure call *CALL X pn* pushes the time (length of *h e*) and the current memory onto the call stack, leaves the return position in *X* and jumps into procedure *pn*. The procedure return *RET X* pops the topmost entry from the call stack and jumps to the return position it expects in *X*.

cmd Π (*pn*,*i*) = *Some* (*CALL X pn′*) ⟶ *step Π* ((*pn*,*i*),*m*,*e*) = *Some*
((*pn′*,*0*),*m*(*X*↦*POS* (*pn*,*i*+*1*)),*e*(|*cs*:=[(*length* (*h e*),*m*)]@*cs e*, *h*:=*h e*@[(*pn*,*i*)]|))

cmd Π (*pn*,*i*) = *Some* (*RET X*) ∧ *m X*=*POS r* ⟶
step Π ((*pn*,*i*),*m*,*e*) = *Some* (*r*,*m*,*e*(|*cs*:=*tl* (*cs e*); *h*:=*h e*@[(*pn*,*i*)]|))

The move operation *MOV X Y* interprets the values at *X* and *Y* as locations *x* and *y*; it copies the value at *x* to *y*.

cmd Π (*pn*,*i*) = *MOV X Y* ∧ *m X*=*NAT x* ∧ *m Y*=*NAT y* ⟶
step ((*pn*,*i*),*m*,*e*) = ((*pn*,*i*+*1*), *m*(*y*↦(*m x*)),*e*(|*h*:=(*h e*)@(*pn*,*i*)|))

Finally, for *HALT* or in case the premises above do not hold *step* returns *None*, i.e. *cmd Π p* = *Some HALT* ⟶ *step Π* (*p*,*m*,*e*) = *None*.

2.3 Safety Logic and Policy

To notate and prove safety properties of programs formally we use a so called safety logic. The essential constituents of this logic are connectives for implication $\llcorner\Longrightarrow\lrcorner$ and conjunction $\llcorner\bigwedge\lrcorner$ and judgements for provability $\vdash$ and validity $\models$.

$$\begin{array}{ll} \llcorner\bigwedge\lrcorner :: 'form\ list \Rightarrow 'form & \vdash :: prog \Rightarrow 'form \Rightarrow bool \\ \llcorner\Longrightarrow\lrcorner :: 'form \Rightarrow 'form \Rightarrow 'form & \models :: state \Rightarrow 'form \Rightarrow bool \end{array}$$

At this point we do not specify the syntax of formulas. This will be done later by instantiating *'form* in a deep and shallow style. However, we assume that the formula language is expressive enough to characterise initial and safe states of programs. That is, we assume that one can define functions $initF :: prog \Rightarrow 'form$, which specifies initial states, and $safeF :: prog \Rightarrow pos \Rightarrow 'form$, which yields local safety formulas. Together *initF* and *safeF* comprise a so called safety policy. A program is safe if all states (p,m,e) we can reach from an initial state (p_0,m_0,e_0) are safe.

$$\begin{array}{l} isSafe\ \Pi = (\forall\, p_0\ m_0\ e_0\ p\ m\ e.\ (p_0,m_0,e_0) \models initF\ \Pi \wedge \\ \qquad ((p_0,m_0,e_0),(p,m,e)) \in (effS\ \Pi)^* \longrightarrow (p,m,e) \models safeF\ \Pi\ p) \end{array}$$

In this paper we instantiate *initF* such that it only holds for states (p,m,e) where the program counter p is $(0,0)$, the memory is uninitialised $\forall\, x.\ m\ x = ILLEGAL$, the history is empty $h\ e = []$ and the call stack has one entry $cs\ e = [(0,m)]$ containing a copy of the initial memory m. The safety formula for a position p, i.e. $safeF\ \Pi\ p$, will be constructed such that it guarantees safe execution of the instruction at p. In our case this means all arguments have proper types and numerical results this instruction yields are equal or below some maximum number *MAX*. In other words: the instruction is type safe and does not cause an overflow. For example if we have $ADD\ X\ Y$ at program position p, the formula $safeF\ \Pi\ p$ demands that variables X and Y have values $NAT\ x$ and $NAT\ y$ such that $x + y \leq MAX$.

2.4 Verification condition generation

Equipped with $\llcorner\bigwedge\lrcorner$, $\llcorner\Longrightarrow\lrcorner$, *initF* and *safeF* we can define a generic VCG, which transforms a given well formed program into a formula, the verification condition *VC*, that is provable only if the program is safe. The VCG soundness theorem below expresses this formally:

$vcg :: prog \Rightarrow 'form$ **theorem**: $wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \longrightarrow isSafe\ \Pi$

The wellformedness judgement *wf* demands that every instruction is annotated and that the main procedure has no *RET* instructions. In our project [1] we usually work with a VCG that also accepts programs where only targets of backward jumps, entry and exit positions of procedures are annotated. However, in this paper we focus on the safety logic and rather keep the VCG simple.

$$
\begin{array}{l}
vcg\ \Pi = \bigwedge [initF\ \Pi \Longrightarrow \bigwedge [safeF\ \Pi\ (ipc\ \Pi),\ anF\ \Pi\ (ipc\ \Pi)]]@ \\
(map\ (\lambda\ p.\ (map(\lambda\ (p',B).\ \bigwedge [safeF\ \Pi\ p,\ anF\ \Pi\ p,\ B] \Longrightarrow \\
\qquad\qquad\qquad\qquad (wpF\ \Pi\ p\ p'\ (\bigwedge [safeF\ \Pi\ p',\ anF\ \Pi\ p'])))) \\
\qquad\qquad (succsF\ \Pi\ p)) \\
\qquad (domC\ \Pi)))
\end{array}
$$

In addition to the instruction and annotation fetch operations *cmd* and *anF* this VCG uses various other auxiliary functions. With $succsF\ \Pi\ p$ it computes the list of all immediate successors p' of a position p paired with a branch condition B. This branch condition is expected to hold whenever p' is accessible from p at runtime. For example assume Π has at position $p{=}(pn,i)$ an instruction that jumps t instructions forward if some condition C holds, or *1* otherwise. Then we expect $succsF\ \Pi\ p$ to yield two successor positions $(pn,i{+}t)$ and $(pn,i{+}1)$ with C or its negation as branch conditions, i.e. $succsF\ \Pi\ (pn,i) = [((pn,i{+}1),C),((pn,i{+}1),\neg C)]$. The function *wpF* is named after Dijkstra's operator for weakest preconditions. It takes a postcondition Q and constructs a formula $wpF\ \Pi\ p\ p'\ Q$, that covers exactly those states where the program Π can make a transition from p to p' such that Q holds when we reach p'.

The verification condition is a big conjunction. There is one initial conjunct and one conjunct for each position in the code domain $domC\ \Pi$, which lists the positions of all instructions in Π. Hence, the overall size of the VC is linear to the program size. The initial conjunct demands that initial states are safe and satisfy the initial annotation $anF\ \Pi\ (ipc\ \Pi)$ where $ipc\ \Pi$ denotes the initial program counter ($(0,0)$ in our case). The conjunct we get for each position p in the code domain demands that a state (p,m,e) that is safe and satisfies the annotation at p only has successor states (p',m',e') that satisfy the safety formula and annotation at p'. For example if a position p annotated with A only has one successor p' with branch condition B and annotation A' we get this conjunct inside the VC:

$$\bigwedge [safeF\ \Pi\ p,\ A,\ B] \Longrightarrow wpF\ \Pi\ p\ p'\ (\bigwedge [safeF\ \Pi\ p',\ A'])$$

So far we have not defined any of the auxiliary functions nor the safety logic and policy. The VCG above is generic. By instantiating the parameter functions one can use it for various PCC platforms. We have proven that the soundness theorem above holds, if these parameter functions meet some basic requirements. The *succsF* function has to approximate the control flow graph of a program. It can yield spurious successors, but must not forget some or yield invalid branch conditions.

assumption *succsFcomplete*:
$$wf\ \Pi \wedge (p,m,e) \in safe_{\Box}\ \Pi \wedge ((p,m,e),(p',m',e')) \in effS\ \Pi \longrightarrow \exists\, B.\ (p',B) \in set\ (succsF\ \Pi\ p) \wedge (p,m,e) \models B$$

This must only hold for all states in the safety closure $safe_{\Box}\ \Pi$, the set of states that can can occur in a safe execution of Π. These are the initial states and states that are reachable from these by only traversing states that are safe and satisfy their annotation.

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

$(p,m,e) \in safe_{\Box}\ \Pi \wedge ((p,m,e),(p',m',e')) \in effS\ \Pi \wedge$
$(p,m,e) \models safeF\ \Pi\ p \wedge (p,m,e) \models anF\ \Pi\ p \wedge$
$(p',m',e') \models safeF\ \Pi\ p' \wedge (p',m',e') \models anF\ \Pi\ p' \longrightarrow (p',m',e') \in safe_{\Box}\ \Pi$

The *wpF* operator has to be compatible with the semantics of SAL. That is, the formula it yields must guarantee the postcondition in the successor state.

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

Another requirement is the correctness of the safety logic. That is, provable formulas must be valid for all states in $safe_{\Box}\ \Pi$.

assumption *correctSafetyLogic*:
$wf\ \Pi \wedge \Pi \vdash F \longrightarrow \forall s \in safe_{\Box}\ \Pi.\ s \models F$

Finally, we require that the logical connectives have their ordinary semantics and that *initF* is consistent with *ipc*.

assumptions
$s \models (A \Longrightarrow_{\sqcup} B) \longrightarrow s \models A \longrightarrow s \models B$
$s \models \bigwedge_{\sqcup} Fs = \forall F \in set\ Fs.\ s \models F$
$(p,m,e) \models initF\ \Pi \longrightarrow ipc\ \Pi = p$

Note that these requirements are kept very weak in order to allow for a wide range of instantiations. With $safe_{\Box}\ \Pi$ in the premisses verifying these requirements becomes simpler; one only has to consider states originating from a safe execution. A lot of properties, for example the wellformedness of the call stack, can be deduced from this fact.

3 Shallow Embedding

3.1 Syntax

In a shallow embedding logical formulas are written directly in the logic of the theorem prover. In our case this means SAL formulas become Isabelle/HOL predicates on states.

type *form* = *state* ⇒ *bool*

We can write arbitrary Isabelle functions from *state* to *bool* and use them to describe machine states. Typically we do this using λ notation. For example $\lambda(p,m,e).\ m\ X = NAT\ 1$ covers all states having the value *NAT 1* at memory location X. Since we have machine states with enviroments, we can also describe states by relating them to former states. For example $\lambda(p,m,e).\ m\ X = (\overleftarrow{m}\ e)\ X$ holds for states, where location X contains the same value as it did at call time of the current procedure. Here we use the shortcut $\overleftarrow{m}\ e$ for the memory at calltime, which we can retrieve from the environment e, i.e. $\overleftarrow{m}\ e = snd\ (hd\ (cs\ e))$. In a similar fashion we can reconstruct the program counter or the environment

at call time, i.e $\overleftarrow{p}\ e = (h\ e)!k$ and $\overleftarrow{e}\ e = e(\!|cs{:=}tl\ (cs\ e);\ h{:=}take\ k\ (h\ e)|\!)$ where $k = fst\ (hd\ (cs\ e))$. The following formulas give some flavour on the style of a shallow embedded formula language. They could be used to annotate the example program *OD*. The function *incA* increments the offset of a position, i.e. $incA\ (pn,i) = (pn,i+1)$.

$A_0 = \lambda(p,m,e).\ True, \quad A_1 = \lambda\ (p,m,e).\ m\ B = NAT\ b_0,$
$A_2 = \bigwedge [A_1, \lambda(p,m,e).\ m\ C = NAT\ c_0], \quad A_3 = \bigwedge [A_1,$
$\lambda(p,m,e).\ \exists\, c.\ m\ C = NAT\ c \wedge (c < 0 \longrightarrow (c = c_0 \wedge b_0 + c_0 \leq MAX))]$
$A_4 = \lambda(p,m,e).\ True$
$A_{5a} = \lambda(p,m,e).\ \ m\ P{=}POS\ (incA\ \overleftarrow{p}\ e) \wedge \exists\, b.\ m\ B{=}NAT\ b \wedge \exists\, c.\ m\ C{=}NAT\ c$
$A_5 = \bigwedge [A_{5a}, \lambda(p,m,e).\ \forall\, x.\ x{\neq}P \longrightarrow m\ x{=}\overleftarrow{m}\ e\ x]$
$A_{6a} = \lambda(p,m,e).\ \forall\, x.\ x{\neq}P \wedge x{\neq}M \longrightarrow m\ x{=}\overleftarrow{m}\ e\ x$
$A_6 = \bigwedge [A_{5a}, A_{6a}, \lambda(p,m,e).\ m\ M = NAT\ MAX]$
$A_7 = \bigwedge [A_{5a}, A_{6a}, \lambda(p,m,e).\ \exists\, c.\ m\ C = NAT\ c \wedge m\ M = NAT\ (MAX - c)]$
$A_8 = \bigwedge [A_7, \lambda(p,m,e).\ \exists\, b\ n.\ m\ B = NAT\ b \wedge m\ M = NAT\ n \wedge n{\leq}b]$
$A_9 = \lambda(p,m,e).\ (\forall\, x.\ x{\neq}C \wedge x{\neq}M \wedge x{\neq}P \longrightarrow m\ x = \overleftarrow{m}\ e\ x) \wedge$
$\qquad (\exists\, b\ c\ c'.\ m\ B{=}NAT\ b\ \wedge m\ C{=}NAT\ c \wedge \overleftarrow{m}\ e\ C{=}NAT\ c' \wedge$
$\qquad (c \neq 0 \longrightarrow (c = c' \wedge b + c' \leq MAX)))\}$

3.2 Validity

In the shallow embedding we define validity of formulas simply by application.

$(p,m,e) \models Q\ \equiv\ Q\ (p,m,e)$

3.3 Provability

The provability judgement $\vdash$ of a logic is usally defined with derivation rules. However, since we write formulas as HOL predicates, we can use Isabelle/HOL's built in derivation rules as proof calculus. We consider a formula F provable if it is valid for all states in $safe_\Box$: $\Pi \vdash F\ \equiv\ \forall\, s.\ s \in safe_\Box\ \Pi \longrightarrow s \models F$

3.4 Weakest Precondition

The predicate, which $wpF\ \Pi\ p\ p'\ Q$ yields, computes the successor state for the transition from p to p' in program Π and applies the postcondition Q to it. For *ADD*, *CALL* and *MOV*, we define *wpF* as follows. The remaining instructions are analogous.

$wpF\ \Pi\ p\ p'\ Q = (case\ cmd\ \Pi\ p\ of\ None \Rightarrow \lambda\ (p,m,e).\ False$
$|\ Some\ a \Rightarrow case\ a\ of\ \ldots$
$|\ ADD\ X\ Y \Rightarrow \lambda(p,m,e).\ Q\ (p',m(X \mapsto (m\ X)\ulcorner+\urcorner(m\ Y)),e(\!|\ h{:=}(h\ e)@[p]|\!))$
$|\ CALL\ X\ pn \Rightarrow \lambda(p,m,e).\ Q\ (p',m(X{\mapsto}POS\ (pn,0)),e(\!|h{:=}(h\ e)@[p]|\!))$
$|\ MOV\ X\ Y \Rightarrow \lambda(p,m,e).\ (case\ m\ X\ of\ ILLEGAL \Rightarrow False\ |\ POS\ r \Rightarrow False\ |$
$\qquad NAT\ x \Rightarrow (case\ m\ Y\ of\ ILLEGAL \Rightarrow False\ |\ POS\ r' \Rightarrow False\ |$
$\qquad NAT\ y \Rightarrow Q(p',m(y \mapsto m\ x),e(\!|h{:=}(h\ e)@[p]|\!)))) \ldots$

3.5 Code generation

Isabelle can generate ML programs out of executable Isabelle/HOL definitions [4]. However, for *wpF* this code generator does not produce the kind of ML program we want. Due to our shallow embedding the code generator also turns safety logic formulas into ML programs. Instead we would like them to be handled as terms of type *state* ⇒ *bool*. A way out is to enhance the code generator by a quotation/antiquotation mechanism. We can introduce functions *term* and *toterm*:: *'a* ⇒ *'a* that are identities for Isabelle's inference system. For the code generator these functions serve as markers: When it generates code for an Isabelle term and steps into a *term* quotation it treats the following input term as output of the currently generated ML program. If inside this mode a *toterm* antiquotation appears, it switches back to normal mode. For example, consider the following two Isabelle definitions:

$f = \lambda\ n.\ n + n + n, \qquad g = \lambda\ n.\ term\ (toterm\ n + (toterm\ (n + n)))$

When applied to operand *5* the ML program we get for *f* would return the integer *15*, whereas the one for *g* would return the term *5 + 10*. Using this mechanism we are able to generate an executable VCG from the definitions in Isabelle.

4 Deep Embedding

4.1 Syntax

In a deep embedding we represent logical formulas as a datatype. At leaf positions we have expressions.

datatype *expr* = *V nat* | *Lv nat* | ⌞*tval*⌟ | *Pc* | *Rp* | *Tm* |
expr ⌞+⌟ *expr* | *expr* ⌞−⌟ *expr* | *expr* ⌞*⌟ *expr* |
⌞*if* *expr* ⌞=⌟ *expr* ⌞*then*⌟ *expr* ⌞*else*⌟ *expr* | *Deref expr* | *Old expr*

Following Winskel [17] we distinguish two kinds of variables. Program variables *V 1*, *V 2*, ... denote values we find at specific locations in memory. For example *V 1* stands for the value we find at location *1* in memory. Apart from these we have logical variables *Lv 1*, *Lv 2*, ..., which stand for arbitrary values that do not depend on the state of a program. Quantification will be defined later on only for logical variables. Since these are not affected by machine instructions we will not have to bother about them when we define the *wpF* Operator later on. Apart from variables we have constants ⌞*NAT 2*⌟, ⌞*POS (0,1)*⌟, ⌞*ILLEGAL*⌟, ... and special identifiers for the current program counter *Pc*, the return position of the current procedure *Rp* and the system time *Tm*. These primitives can be combined via arithmetical operators and conditionals. To support pointers, we have the *Deref E* expression. It yields the value we find at address *a*, provided *E* evaluates to *NAT a*. Finally, we have a call state expression *Old E*, which interprets an expression *E* in the call state of the current procedure. This enables one to describe states by relating them to former states.

datatype $form = True \mid False \mid \bigwedge form\ list \mid form \Longrightarrow form \mid \neg form \mid expr = expr \mid expr \leqslant expr \mid expr < expr \mid expr :: vtype \mid \forall nat\ form$

Formulas are either the boolean constants $True$ and $False$, conjunctions $\bigwedge [A,B,\ldots]$, implications $A \Longrightarrow B$ or negations $\neg A$. In addition we have relational formulas $E = E'$, $E \leqslant E'$ or $E < E'$ and a type checking formula $E :: T$, where T can either be Pos for POS or Nat for NAT.

datatype $vtype = Pos \mid Nat$

Finally, we can quantify over logical variables. In $\forall v\ F$ all free occurences of $Lv\ v$ in F are bound and F is expected . Below we have the annotations for our example program, written in this new style.

$A_0 = True, \quad A_1 = V\ B = NAT\ b_0, \quad A_2 = \bigwedge [A_1,\ V\ C = NAT\ c_0]$
$A_3 = \bigwedge [\ A_1,\ V\ C :: Nat,\ NAT\ 0 < V\ C \Longrightarrow$
$\bigwedge [V\ B + V\ C \leqslant NAT\ MAX,\ V\ C = NAT\ c_0]$
$A_4 = True, \quad A_{5a} = \bigwedge [V\ P = Rp,\ V\ B :: Nat,\ V\ C :: Nat]$
$A_{5b} = \forall x \neg (Lv\ x = NAT\ P) \Longrightarrow Deref\ (Lv\ x) = Old\ (Deref\ (Lv\ x))$
$A_5 = \bigwedge [A_{5a}, A_{5b}], \quad A_{6a} = \forall x \bigwedge [\neg (Lv\ x = NAT\ P),$
$\neg (Lv\ x = NAT\ M)] \Longrightarrow Deref\ (Lv\ x) = Old\ (Deref\ (Lv\ x))$
$A_6 = \bigwedge [A_{5a}, A_{6a}, V\ M = NAT\ MAX]$
$A_7 = \bigwedge [A_{5a}, A_{6a}, V\ M = NAT\ MAX \neg V\ C], A_8 = A_7$
$A_9 = \bigwedge [A_{5a}, \forall x \bigwedge [\neg (Lv\ x = NAT\ P), \neg (Lv\ x = NAT\ M,$
$\neg (Lv\ x) = NAT\ C)] \Longrightarrow Deref\ (Lv\ x) = Old\ (Deref\ (Lv\ x)),$
$\neg (V\ C = NAT\ 0) \Longrightarrow \bigwedge [V\ C = Old\ (V\ C),\ V\ B + V\ C \leqslant NAT\ MAX]]$

4.2 Validity

We use $eval::(nat \Rightarrow tval) \Rightarrow state \Rightarrow expr \Rightarrow tval$ to evaluate expressions on a given state and interpretation for logical variables. Program variables stand for memory locations, logical variables are interpreted via L and constants are directly converted to values.

$eval\ L\ (p,m,e)\ (V\ v) = m\ v, \qquad eval\ L\ s\ (Lv\ v) = L\ v, \qquad eval\ L\ s\ ty = tv$

The identifer Pc stands for the program counter, Tm for the system time (number of executed instructions), and Rp for the return position of the current procedure. It evaluates to $POS\ (0,0)$ if we are in the main procedure or to $ILLEGAL$ in case of a malformed call stack.

$eval\ (p,m,e)\ Pc = POS\ p \qquad eval\ (p,m,e)\ Tm = NAT\ (length\ (h\ e))$
$eval\ (p,m,e)\ Rp = (case\ length\ (cs\ e)\ of\ 0 \Rightarrow ILLEGAL$
$\mid Suc\ n \Rightarrow (case\ n\ of\ 0 \Rightarrow POS\ (0,0) \mid Suc\ n' \Rightarrow POS\ (incA\ (\overleftarrow{p}\ e))))$

Arithmetical expressions and conditionals are evaluated recursively.

$eval\ L\ s\ (E + E') = (eval\ L\ s\ E) \dot{+} (eval\ L\ s\ E')$

The cases for $\neg$ and $*$ are analogous.

$eval\ L\ s\ (if\ E_0 = E_1\ then\ E_2\ else\ E_3) =$

if $(eval\ L\ s\ E_0 = eval\ L\ s\ E_1)$ *then* $eval\ L\ s\ E_2$ *else* $eval\ L\ s\ E_3)$

With *Deref E* we fetch the value at position a, provided E evaluates to $NAT\ a$.

$eval\ L\ (p,m,e)\ (Deref\ E) = (case\ (eval\ L\ (p,m,e)\ E)$
$of\ ILLEGAL \Rightarrow ILLEGAL \mid POS\ r \Rightarrow ILLEGAL \mid NAT\ a \Rightarrow m\ a)$

Finally, we evaluate *Old E* by retrieving the call state from the environment.

$eval\ L\ (p,m,e)\ (Old\ E) = eval\ L\ (\overleftarrow{p}\ e,\overleftarrow{m}\ e,\overleftarrow{e}\ e)\ E$

Next, we define the validity of formulas relative to states and interpretations.

$L,s \models \lfloor True \rfloor \quad \neg\ L,s \models \lfloor False \rfloor \quad L,s \models \lfloor \bigwedge \rfloor\ Fs = (\forall\ F \in set\ Fs.\ L,s \models F)$
$L,s \models F \lfloor \Longrightarrow \rfloor\ F' = (L,s \models F \longrightarrow L,s \models F') \quad L,s \models \lfloor \neg \rfloor F = \neg\ (L,s \models F)$
$L,s \models E \lfloor = \rfloor E' = (eval\ L\ s\ E\ = eval\ L\ s\ E')$
$L,s \models E \lfloor :: \rfloor\ T = (case\ (eval\ L\ s\ E)$
$of\ ILLEGAL \Rightarrow False \mid POS\ p \Rightarrow T{=}Pos \mid NAT\ n \Rightarrow T{=}Nat)$
$L,s \models E \lfloor \leqslant \rfloor E' = (case\ (eval\ L\ s\ E)\ of\ ILLEGAL \Rightarrow False \mid POS\ r \Rightarrow False \mid$
$NAT\ x \Rightarrow (case\ (eval\ L\ s\ E')\ of\ ILLEGAL \Rightarrow False \mid POS\ r' \Rightarrow False \mid$
$NAT\ y \Rightarrow x \leq y))$

The case for $\lfloor < \rfloor$ is analogous to $\lfloor \leqslant \rfloor$; just replace $\leq$ with $<$. The meaning of $\lfloor \forall \rfloor\ v\ F$ is that F holds irrespective of the interpretation of $Lv\ v$.

$L,s \models \lfloor \forall \rfloor\ v\ F = (\forall\ tv.\ L(v \mapsto tv),s \models F)$

4.3 Provability

Provability is defined in a similar manner as for the shallow embedding. A formula is considered provable if it holds for all interpretations and states in $safe_\square$.

$\Pi \vdash F \equiv \forall\ L.\ \forall\ s \in safe_\square\ \Pi.\ L,s \models F$

To show provability of a formula, there are two alternatives. One can either expand the definition of $\vdash$ and work directly with the inference rules of HOL. This makes sense if the code consumer's logic is HOL (something that the shallow embedding requires). On the other hand, if the code consumer's safety logic is more specialised, the deep embedding can still model the precise inference system involved. For example, we have derived suitable introduction and elimination rules for our language of formulas that do not rely on λ calculus and HOL. However, proving with deep embedded inference rules inside Isabelle/HOL turned out to be inconvenient. The proof tools are designed to prove HOL formulae not elements of a datatype. In addition the $\lfloor \forall \rfloor$ elimination rule causes trouble. It says that from $\lfloor \forall \rfloor\ v\ F$ we can deduce $F[t/v]$, which is F with all free occurrences of $Lv\ v$ replaced by some term t. We need a form of substitution that renames bounded logical variables in F when they occur as free variables in t. This renaming complicates the correctness proof and turned out to double the size of our deep embedding theories. Nevertheless defining and verifying deep embedded inference rules inside Isabelle/HOL pays off if one wants to use specialised tools for proof search and checking. Since Isabelle is generic one can also think about instantiating the safety logic as a new object logic.

4.4 Weakest Precondition

A big difference between our shallow and deep embedding lies in the definition of the *wpF* operator. In the deep embedding we express the effects of instructions at the level of formulas with substitutions. For these we use finite maps, which we internally represent as lists of pairs, e.g. $fm = [(1,1),(2,4),(3,5),(3,6)]$. Finite maps enable us to generate executable ML code for map operations like lookup -↓-:: $'a \triangleright 'b \Rightarrow 'a \Rightarrow 'b\ option$, domain dom:: $('a \triangleright 'b) \Rightarrow 'a\ list$ or range ran:: $('a \triangleright 'b) \Rightarrow 'b\ list$. A few examples demonstrate how these operators work $fm{\downarrow}0 = None$, $fm{\downarrow}3 = Some\ 5$, $dom\ fm = [1,2,3]$ and $ran\ fm = [1,4,5]$. Note that a pair (x,y) is overwritten by a pair (x,y') to the left of it. For *ADD*, *CALL* and *MOV*, we define *wpF* as follows. The remaining instructions are analogous.

$$
\begin{aligned}
&wpF\ \Pi\ p\ p'\ Q = (case\ cmd\ \Pi\ p)\ of\ None \Rightarrow \lfloor False \rfloor \mid Some\ a \Rightarrow case\ a\ of\ \ldots \\
&\mid ADD\ X\ Y \Rightarrow substF\ [(Tm,\ Tm \lfloor{+}\rfloor \lfloor NAT\ 1 \rfloor),(Pc,\lfloor POS\ p' \rfloor), \\
&\qquad\qquad (V\ X,\ V\ X \lfloor{+}\rfloor V\ Y)]\ Q \\
&\mid CALL\ X\ pn \Rightarrow popCs\ (substF\ [(Tm,Tm \lfloor{+}\rfloor \lfloor NAT\ 1 \rfloor),(Rp,\lfloor POS\ (pn,i{+}1) \rfloor), \\
&\qquad\qquad (Pc,\lfloor POS\ p' \rfloor),(V\ X,\lfloor POS\ (pn,i{+}1) \rfloor)]\ Q) \\
&\mid MOV\ X\ Y \Rightarrow substPtF\ X\ Y\ [(Tm,Tm \lfloor{+}\rfloor \lfloor NAT\ 1 \rfloor),(Pc,\lfloor POS\ p' \rfloor)]\ Q\ \ \ldots
\end{aligned}
$$

The substitution function $substF$:: $(expr \triangleright expr) \Rightarrow form \Rightarrow form$ is the main workhorse for the deep embedding. With $substF\ em\ F$ we simultaneously substitute expressions of the form $V\ v$, Tm, Pc or Rp in a formula F according to a finite map em. It traverses F and applies $substE\ em\ E$ on all expressions it finds.

$$
\begin{aligned}
&substF\ em\ \lfloor True \rfloor = \lfloor True \rfloor \qquad substF\ em\ \lfloor False \rfloor = \lfloor False \rfloor \\
&substF\ em\ (\lfloor \bigwedge \rfloor\ Fs) = \lfloor \bigwedge \rfloor\ (map\ (substF\ em)\ Fs) \\
&substF\ em\ (F_1 \lfloor \Longrightarrow \rfloor\ F_2) = (substF\ em\ F_1) \lfloor \Longrightarrow \rfloor\ (substF\ em\ F_2) \\
&substF\ em\ (\lfloor \neg \rfloor\ F) = \lfloor \neg \rfloor\ (substF\ em\ F) \\
&substF\ em\ (E \lfloor :: \rfloor\ T) = (substE\ em\ E) \lfloor :: \rfloor\ T \\
&substF\ em\ (\lfloor \forall \rfloor\ v\ F) = \lfloor \forall \rfloor\ v\ (substF\ em\ F)
\end{aligned}
$$

Expressions of the form $Lv\ v$ or $\lfloor ty \rfloor$ are ignored by *substE*, because they are not affected by instructions. Here Winskel's [17] distinction of program and logical variables pays off. In *wpF* only program variables appear in the expressions we substitute in. Hence, we do not have to rename bound (=logical) variables. However, a substitution with renaming is useful when one wants to define deep embedded inference rules (see §4.3). For the remaining primitive expressions *substE* looks up the expression map and replaces them with their substitute in case there is some. Otherwise *substE* just recurses down the expression structure.

$$E{=}Lv\ v \vee E{=}\lfloor ty \rfloor \longrightarrow substE\ em\ E = E$$

$$
\begin{aligned}
&E{=}V\ v \vee E{\in}\{Pc,Rp,Tm\} \longrightarrow \\
&substE\ em\ E = (case\ em{\downarrow}E\ of\ None \Rightarrow E \mid Some\ \ E' \Rightarrow E')
\end{aligned}
$$

$$o{\in}\{\lfloor{+}\rfloor,\lfloor{-}\rfloor,\lfloor{*}\rfloor\} \longrightarrow substE\ em\ (E_1\ o\ E_2) = (substE\ em\ E_1)\ o\ (substE\ em\ E_2)$$

$$
\begin{aligned}
&substE\ em\ (\lfloor if \rfloor\ E_1 \lfloor = \rfloor\ E_2\ \lfloor then \rfloor\ E_3\ \lfloor else \rfloor\ E_4) = \lfloor if \rfloor\ (substE\ em\ \ E_1) \lfloor = \rfloor \\
&(substE\ em\ \ E_2)\ \lfloor then \rfloor\ (substE\ em\ E_3)\ \lfloor else \rfloor\ (substE\ em\ \ E_4)
\end{aligned}
$$

In case of *Deref E*, we have to check, whether E evaluates to $NAT\ v$ where v is the location of a variable $V\ v$ that is substituted by em to some expression E'. In this case the *Deref E* expression needs to be substituted as well. Since the evaluation of E depends on the state we cannot do this statically. A way out is to replace *Deref E* by another expression that incorporates this check, e.g. $\llbracket if \rrbracket\ E'' = \llbracket NAT\ v \rrbracket\ \llbracket then \rrbracket\ E'\ \llbracket else \rrbracket\ E''$ where $E'' = substE\ em\ E$ and $em{\downarrow}E = Some\ E'$. This check needs to be done for all variables in the domain of em; we fetch them with the auxiliary function *changedvars*.

$v \in set\ (changedvars\ em) = (\exists E'.\ em{\downarrow}(V\ v) = Some\ E')$

$substE\ em\ (Deref\ E) = (let\ E'' = substE\ em\ E;$
$res = (foldl\ (\lambda E'.\ (v,E').\ (\llbracket if \rrbracket\ E'' = \llbracket NAT\ v \rrbracket\ \llbracket then \rrbracket\ E'\ \llbracket else \rrbracket\ E''))$
$(Deref\ E'')\ (changedvars\ em))\ in\ res)$

In this definition we use the HOL function *foldl* wich calls its input function recursively over a list of arguments.

$foldl\ f\ a\ [] = a \qquad foldl\ f\ a\ (x\#xs) = foldl\ f\ (f\ a\ x)\ xs$

Since we use *substE* only to express the effects of instructions on the current state, it ceases to play a role when we come to an expression that refers to another state. Hence, *substE* terminates when it reaches an *Old* expression.

$substE\ em\ (Old\ E) = Old\ E$

To express the effect of pointer instructions, e.g. $MOV\ X\ Y$, we use the special substitution function $substPtF :: nat \Rightarrow nat \Rightarrow (expr \triangleright expr) \Rightarrow form \Rightarrow form$. It works exactly like *substF* except that it calls $substPtE :: nat \Rightarrow nat \Rightarrow (expr \triangleright expr) \Rightarrow expr \Rightarrow expr$, when it encounters an expression. The function *substPtE* is a variant of *substE* that does additional transformations for variables and *Deref* expressions. When we execute $MOV\ X\ Y$ it could be that the target location $NAT\ v$ stored in Y coincides with a variable $V\ v$ in an expression. In this case the value of $V\ v$ after the $MOV\ X\ Y$ instruction becomes the value at the location we find in X. To express this effect we can replace $V\ v$ with a conditional expression.

$substPtE\ X\ Y\ em\ (V\ v) =$
$\llbracket if \rrbracket\ V\ Y = \llbracket NAT\ v \rrbracket\ \llbracket then \rrbracket\ Deref\ (V\ X)\ \llbracket else \rrbracket\ (substE\ em\ (V\ v))$.

For *Deref E* expressions the same technique can be applied.

$substPtE\ X\ Y\ em\ (Deref\ E) = let\ E'' = substPtE\ X\ Y\ em\ E;\ res = \ldots$
$in\ (\llbracket if \rrbracket\ V\ Y = E''\ \llbracket then \rrbracket\ Deref\ (V\ X)\ \llbracket else \rrbracket\ res)$

Finally, we need special formula manipulations for procedure calls and returns. Remember that *CALL* pushes the current state (call state) onto the call stack. With *popCs* the *wpF* function reverses this effect. After the call a new procedure is active and expressions of the form *Old E* have a different meaning. The expression *Old E* evaluated after the call yields the same value as E does before (because $(\overleftarrow{p}\ e', \overleftarrow{m}\ e', \overleftarrow{e}\ e') = (p,m,e)$ when $e' = (cs := ((length\ (h\ e),m)\#(cs\ e));\ h := (h\ e)@[p])$. To ensure validity of some formula Q after the call *popCs*

Q replaces occurrences of *Old E* in Q with E. For *RET* we do the opposite; we replace *Old* (*Old E*) with *Old E*.

5 Comparison

Expressiveness
In our shallow embedding we use predicates in HOL as assertion language. These are more expressive than the deep embedded first order formulas. One can quantify over functions, use expressions of any type and has direct and unrestricted access to the state. For example we verified list reversal [1] in our shallow embedding, but we found it difficult to do this example in the deep embedding, which does not offer expressions for lists. However, this shortcoming could be overcome by utilizing a richer assertion language.

Proof Size
Shallow and deep embedding differ in the definition of *wpF*; one uses λ abstraction, the other substitutions. This difference affects verification conditions and their proofs. For example for the transition from (*0*,*2*) to (*1*,*0*) in our example program we get these formulas:

$VC_s = \bigwedge[\lambda(p,m,e).\mathit{True}, \lambda(p,m,e).m\ B{=}\mathit{NAT}\ b_0 \wedge m\ C{=}\mathit{NAT}\ c_0, \lambda(p,m,e).\mathit{True}]$
$\Longrightarrow\ (\lambda(p,m,e).\ \bigwedge[\lambda(p,m,e).\mathit{MAX}{\leq}\mathit{MAX}, \lambda(p,m,e).m\ P{=}\mathit{POS}\ (\mathit{incA}\ \overleftarrow{p}\ e) \wedge$
$\exists b.m\ B{=}\mathit{NAT}\ b \wedge \exists c.m\ C{=}\mathit{NAT}\ c \wedge \forall x.\ x{\neq}P \longrightarrow m\ x{=}\overleftarrow{m}\ e\ x]$
$((0,2), m[P{\mapsto}\mathit{POS}\ (0,3)], e(\!|h{:=}(h\ e)@[(0,2)]; cs{:=}(\mathit{length}\ (h\ e), m)\#(cs\ e)|\!)))$

$VC_d = \bigwedge[\mathit{True}, \bigwedge[V\ B{=}\mathit{NAT}\ b_0, V\ C{=}\mathit{NAT}\ c_0], \mathit{True}]$
$\Longrightarrow\ \bigwedge[\mathit{NAT}\ \mathit{MAX}{\leq}\mathit{NAT}\ \mathit{MAX}, \mathit{POS}\ (0,3){=}\mathit{POS}\ (0,3), V\ B :: \mathit{Nat},$
$V\ C :: \mathit{Nat}, \forall x.(\mathit{Lv}\ x{=}\mathit{NAT}\ P) \Longrightarrow \mathit{Deref}\ (\mathit{Lv}\ x){=}\mathit{Deref}\ (\mathit{Lv}\ x)]$

In the formula VC_s, which results from the shallow embedding, we have various uncontracted λ terms. This is because the VCG does not simplify; it just plugs annotations and safety formulas into a skeleton of conjunctions and implications determined by the control flow graph. The contraction of these λ terms is done when we prove them in Isabelle. In VC_d, which results from the deep embedding, these simplifications are carried out by the substitution function, which is executed when we run the VCG. Hence, the proof of VC_s involves more simplification steps than the one of VC_d. For example in VC_s we find after β contraction this subformula:

$m[P{\mapsto}\mathit{POS}\ (0,3)]\ x = \overleftarrow{m}\ e(\!|h{:=}(h\ e)@[(0,2)];\ cs{:=}(\mathit{length}\ (h\ e), m)\#(cs\ e)|\!)\ x$

Knowing that $x \neq P$ and the definitions of $\mapsto$, $\overleftarrow{m}$ and record updates, we can simplify this to the triviality $m\ x = m\ x$. In VC_d this triviality is already exposed by the *wpF* operator, which yields $\mathit{Deref}\ (\mathit{Lv}\ x) = \mathit{Deref}\ (\mathit{Lv}\ x)$ in this situation. The VC we get for our example program can be proven automatically in Isabelle using built in decision procedures for Presburger Arithmetics. The latter is required for the formula we get for the transition from (*1*,*4*) to (*0*,*4*).

There we have to show that the Addition operation at (*0*,*3*) cannot overflow. The resulting proof object for the shallow embedding is about twice the size as the deep embedding. Other experiments [1] confirm this fact.

Formula Optimisations
Another advantage we get from the deep embedding is that we can write Isabelle/HOL functions that operate on the structure of formulas. This enables us to optimise VCs after/during their construction. Elsewhere [1] we present an optimizer for VCs in the deep embedding. It evaluates constant formulas and subexpressions, for example $\lfloor NAT\ MAX \rfloor \leqslant \lfloor NAT\ MAX \rfloor$ can be reduced to $\lfloor True \rfloor$. In addition it simplifies implications, for example $A \Longrightarrow \lfloor True \rfloor$ or $\bigwedge [\ldots, A, \ldots] \Longrightarrow A$, and conjunctions, for example $\bigwedge [\ldots, \lfloor True \rfloor, \ldots]$ or $\bigwedge [A, \bigwedge [B, C], D]$. It can also do some trivial deductions, for example $V\ b = \lfloor NAT\ b_0 \rfloor$ implies $V\ b :: Nat$. These transformations, which can be done in time quadratic to the formula size, suffice to reduce the size of VCs and their proofs considerably. For example VC_d can be reduced to $\lfloor True \rfloor$. Although these optimisations do not always trivialise VCs, experiments [1] show that leads to proof objects that are about 3 times smaller than they are in the shallow embedding. More could be gained by coupling the optimizer to a proof procedure that performs introduction and elimination rules on our first order formula language.

Annotation Analysis
In the shallow embedding we cannot analyse annotations in the VCG or its helper functions. This is because HOL predicates cannot be structurally analysed by other HOL functions (Isabelle does not support reflection). In the deep embedding the structure of formulas is accessible and can be used to handle more complex machine instructions like computed gotos. We simply demand that the possible targets of such jumps, which are runtime values and therefore hard to determine statically, must be annotated. Then we can define a *succsF* function that reads off the possible successors from the annotation. Since annotations must be verified in the resulting VC this approach is sound.

6 Conclusion

As we expected the deep embedded safety logic was harder to instantiate within our PCC framework than the shallow one. One has to define explicit evaluation and substitution functions and prove them correct. This becomes a non trivial task when variable renamings are involved. In addition one has to deal with subtle effects pointer instructions or procedure calls have on formulas. However, once the deep embedding is proven correct it buys us a lot. We can specify and prove correct an optimiser or pre-prover for VCs and handle more instructions (computed gotos). Homeier [11] also works with a deep embedded assertion language and points out similar advantages. Based on these experiences we instantiated our PCC framework to a down-sized version of the Java Virtual Machine [9] using an extended version of our deep embedded assertion language.

References

1. VeryPCC project:. `http://isabelle.in.tum.de/verypcc/`, 2003.
2. A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, June 2001.
3. A. W. Appel and A. P. Felty. 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, January 2000.
4. S. Berghofer. Program extraction in simply-typed higher order logic. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop, (TYPES 2002)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2003.
5. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, International Workshop, (TYPES 2000)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2000.
6. S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lect. Notes in Comp. Sci.*, pages 38–52. Springer-Verlag, 2000.
7. J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound tal for back-end optimization. In *Programming Languages Design and Implementation (PLDI)*. ACM Sigplan, 2003.
8. C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. ACM SIGPLAN conf. Programming Language Design and Implementation*, pages 95–107, 2000.
9. K. Gerwin and N. Tobias. A machine-checked model for a Java-like language, virtual machine and compiler. Research report, National ICT Australia, Sydney, 2004.
10. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th IEEE Symp. Logic in Computer Science*, pages 89–100, July 2002.
11. P. V. Homeier and D. F. Martin. Secure mechanical verification of mutually recursive procedures. pages 1–19, 2003.
12. G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.
13. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 85–97. ACM Press, 1998.
14. G. C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
15. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer, 2002.
17. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.