Proof-Carrying-Code

Applying formal methods in a distributed world

Hans-Wolfgang Loidl

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

June 25, 2007

Motivation

Downloading software over the network is nowadays common-place.

But who says that the software does what it promises to do?

Who protects the consumer from malicious software or other undesirable side-effects?

Motivation

Downloading software over the network is nowadays common-place.

But who says that the software does what it promises to do?

Who protects the consumer from malicious software or other undesirable side-effects?

$\Longrightarrow$ **Mechanisms for ensuring that a program is "well-behaved" are needed.**

Authentication for Mobile Code

The main mechanisms used nowadays are based on authentication.
Java:

- Originally a sandbox model where all code is untrusted and executed in a secure environment (sandbox)
- In newer versions security policies can be defined to have more fine-grained control over the level of security defined.
  Managed through cryptographic signatures on the code.

Authentication for Mobile Code

Windows:

- Microsoft's Authenticode attaches cryptographic signatures to the code.
- User can distinguish code from different providers.
- Very widely used — more or less compulsory in Windows XP for drivers.

Authentication for Mobile Code

Windows:

- Microsoft's Authenticode attaches cryptographic signatures to the code.
- User can distinguish code from different providers.
- Very widely used — more or less compulsory in Windows XP for drivers.

But, all these mechanisms say nothing about the code, only about the supplier of the code!

Whom do you trust completely?

Maybe that's not such a good idea!

Microsoft Security Bulletin MS01-017

Who should read this bulletin: All customers using Microsoft® products.

Technical description: In mid-March 2001, VeriSign, Inc., advised Microsoft that on January 29 and 30, 2001, it issued two VeriSign Class 3 code-signing digital certificates to an individual who fraudulently claimed to be a Microsoft employee. …

Impact of vulnerability: Attacker could digitally sign code using the name "Microsoft Corporation".

Proof-Carrying-Code (PCC): The idea

Goal: Safe execution of untrusted code.

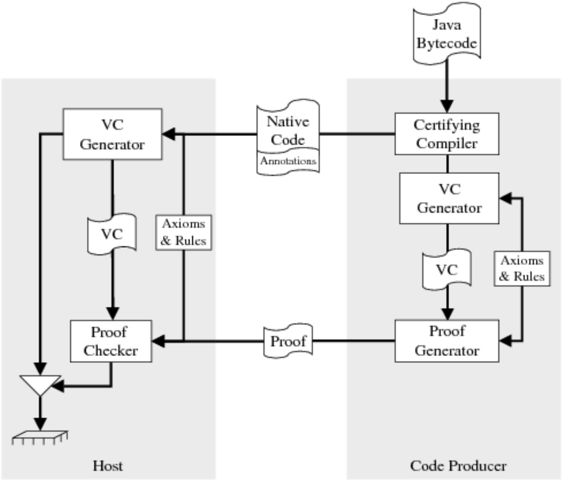
> *PCC is a software mechanism that allows a host system to determine with certainty that it is safe to execute a program supplied by an untrusted source.*

Method: Together with the code, a *certificate* describing its behaviour is sent.

This certificate is a condensed form of a formal proof of this behaviour.

Before execution, the consumer can check the behaviour, by running the proof against the program.

A PCC architecture



Program Verification Techniques

Many techniques for PCC come from the area of **program verification**. Main differences:

General program verification

- is trying to **verify good behaviour** (correctness).
- is usually interactive
- requires at least programmer annotations as invariants to the program

Program Verification Techniques

Many techniques for PCC come from the area of **program verification**. Main differences:

General program verification

- is trying to **verify good behaviour** (correctness).
- is usually interactive
- requires at least programmer annotations as invariants to the program

PCC

- is trying to **falsify bad behaviour**
- must be automatic
- may be based on inferred information from the high-level

Program Verification Techniques

Many techniques for PCC come from the area of **program verification**. Main differences:

General program verification

- is trying to **verify good behaviour** (correctness).
- is usually interactive
- requires at least programmer annotations as invariants to the program

PCC

- is trying to **falsify bad behaviour**
- must be automatic
- may be based on inferred information from the high-level

Observation: Checking a proof is much simpler than creating one

PCC: Selling Points

Advantages of PCC over present-day mechanisms:

- General mechanism for many different safety policies
- Behaviour can be checked before execution
- Certificates are tamper-proof
- Proofs may be hard to generate (producer) but are easy to check (consumer)

What does "well-behaved" mean?

PCC is a general framework and can be instantiated to many different **safety policies**.

A safety policy defines the meaning of "well-behaved".

What does "well-behaved" mean?

PCC is a general framework and can be instantiated to many different **safety policies**.

A safety policy defines the meaning of "well-behaved".

Examples:

- (functional) correctness
- type correctness ([1])
- array bounds and memory access (CCured)
- resource-consumption (MRG)

An Example: CCured

CCured is a system for checking **pointer-safety** of C programs, developed by the group of George Necula at Berkeley.

Uses a hybrid mechanism of static type checking and run-time checks.

Goal: Prove pointer safety statically, where possible, and minimise required run-time checks.

A Roadmap to a PCC infrastructure

Task of the infrastructure: **Certify** that the **execution** of the program is **well-behaved**.

Several steps to build the infrastructure:

- Formalise **execution** as an **operational semantics** of the language.
- Formalise **well-behaved** as a **security policy** (type-system)
- **Certify** safety by producing a proof-term (or similar).

The core language

Mini-C language:

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid e_1 \ \mathtt{op}\ e_2 \mid (\tau)e \mid e_1 \oplus e_2 \mid \,!e \\
c & ::= & \mathtt{skip} \mid c_1 ; c_2 \mid e_1 := e_2
\end{array}
$$

Types: standard C types with extension for **pointers into arrays and dynamic types**.

Efficient type inference is possible and demonstrated for this type system.

The CCured type system: pointers

C contains 2 evil pointer operations: arithmetic and casts.

The type system distinguishes between 3 kinds of pointers:

- **Safe pointers**: no arithmetic or casts; represented as an address
- **Sequence pointers**: arithmetic but **no casts**; represented as a region
- **Dynamic pointers**: **casts**, all bets are off! represented as a region

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                      // ptr arithm
    e = *p;                         // read elem
    while ((int)e % 2 == 0) {       // check tag
        e = *(int **)e;             // unbox
    }
    acc += ((int)e >> 1);           // strip tag
}
```

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                        // ptr arithm
    e = *p;                           // read elem
    while ((int)e % 2 == 0) {         // check tag
        e = *(int **)e;               // unbox
    }
    acc += ((int)e >> 1);             // strip tag
}
```

a and p point into an array with elems of type int *

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                      // ptr arithm
    e = *p;                         // read elem
    while ((int)e % 2 == 0) {       // check tag
        e = *(int **)e;             // unbox
    }
    acc += ((int)e >> 1);           // strip tag
}
```

a is subject to pointer arithm ("sequence pointer")

$\Longrightarrow$ check for out of bounds

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                      // ptr arithm
    e = *p;                         // read elem
    while ((int)e % 2 == 0) {       // check tag
        e = *(int **)e;             // unbox
    }
    acc += ((int)e >> 1);           // strip tag
}
```

p has no arithmetic (“safe pointer”)
$\Longrightarrow$ no bounds check needed

Example program

Sum over an array of boxed integers:

```
int acc; /* accumulator */ int **p; // elem ptr
int **a; /* array */       int i;   // index
int *e;   /* unboxer */
acc = 0;
for (i=0; i<100; i++) {
    p = a + i;                     // ptr arithm
    e = *p;                        // read elem
    while ((int)e % 2 == 0) {      // check tag
        e = *(int **)e;            // unbox
    }
    acc += ((int)e >> 1);          // strip tag
}
```

e is subject to a type cast ("dynamic pointer")

$\Longrightarrow$ nothing known about underlying type

Safe pointers

Invariant for **SAFE** pointers of type τ:

- Maybe 0 or
- points to a valid area of memory containing an object of type τ.
- All other pointers to the same area are also of SAFE and of type τ.
- Safe pointers are represented using one word.

Run-time check: null-pointer reference.

Sequence pointers

Invariants for **SEQUENCE** pointers:

- Cannot be cast (passing actual arguments and returning are implicit casts).
- Can be subject to pointer arithmetic (adding or subtracting an integer from it).
- Can be set to any integer value.
- Can be cast to an integer and can be subtracted from another pointer (useful for comparisons).
- Sequence pointers are represented using three words.

Run-time checks: null-pointer check and bounds check.

Dynamic pointers

Invariants for **DYNAMIC** pointers:

- Can be cast from and to dynamic pointers;
- Can be cast from and to integers;
- Can perform pointer arithmetic;
- Target memory maintains tags of types at run-time;
- Aliases are dynamic pointers.

Operational Semantics

The value of an integer, or a safe pointer is an integer n; the value of a sequence or dynamic pointer is a **home**, modelled as a pair $\mathbb{N} \times \mathbb{N}$ of start address and offset.

$$v ::= n \ \mid \langle h, n \rangle$$

Operational Semantics

The value of an integer, or a safe pointer is an integer n; the value of a sequence or dynamic pointer is a **home**, modelled as a pair $\mathbb{N} \times \mathbb{N}$ of start address and offset.

$$v ::= n \mid \langle h, n \rangle$$

Each home is tagged as being an integer or a pointer, and has an associated **kind** and **size** functions. The semantic domain for pointers:

$$
\begin{aligned}
\| \texttt{int} \|_H &= \mathbb{N} \\
\| \texttt{DYNAMIC} \|_H &= \{\langle h, n \rangle \mid \; h \in H \land (h = 0 \lor \texttt{kind}(h) = \mathit{untyped}\} \\
\| \tau \texttt{ ref SEQ} \|_H &= \{\langle h, n \rangle \mid \; h \in H \land (h = 0 \lor \texttt{kind}(h) = \mathit{typed}(\tau)\} \\
\| \tau \texttt{ ref SAFE} \|_H &= \{h + i \mid \; h \in H \land 0 \leq i \leq \mathit{size}(h) \land \\
&\qquad (h = 0 \lor \texttt{kind}(h) = \mathit{typed}(\tau)\}
\end{aligned}
$$

Operational Semantics (pointers)

$$\frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n_1 \rangle \quad \Sigma, M \vdash e_2 \Downarrow n_2}{\Sigma, M \vdash e_1 \oplus e_2 \Downarrow \langle h_1, n_1 + n_2 \rangle} \quad \text{(Pointer Artihm)}$$

$$\frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle}{\Sigma, M \vdash (\texttt{int})e \Downarrow h + n} \quad \text{(CastToInt)}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\tau \ \texttt{ref SEQ})e \Downarrow \langle 0, n \rangle} \quad \text{(CastToSeq)}$$

$$\frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle \quad \mathbf{0} \le \mathbf{n} \le \texttt{size}(\mathbf{h})}{\Sigma, M \vdash (\tau \ \texttt{ref SAFE})e \Downarrow h + n} \quad \text{(CastToSafe)}$$

Operational Semantics (read operations)

Two kinds of reads, with different obligations for run-time checks:

$$\frac{\Sigma, M \vdash e \Downarrow n \quad \mathbf{n \neq 0}}{\Sigma, M \vdash !e \Downarrow M(n)} \quad (\textsc{SafeRd})$$

$$\frac{\Sigma, M \vdash e \Downarrow \langle h, n\rangle \quad \mathbf{h \neq 0} \quad \mathbf{0 \leq n \leq \texttt{size}(h)}}{\Sigma, M \vdash !e \Downarrow M(h+n)} \quad (\textsc{DynRd})$$

$$\frac{\Sigma, M \vdash e_1 \Downarrow n \quad \mathbf{n \neq 0} \quad \Sigma, M \vdash e_2 \Downarrow v}{\Sigma, M \vdash e_1 := e_2 \Downarrow M(n \mapsto v)} \quad (\textsc{SafeWr})$$

$$\frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n\rangle \quad \mathbf{h \neq 0} \quad \mathbf{0 \leq n \leq \texttt{size}(h)} \quad \Sigma, M \vdash e_2 \Downarrow v}{\Sigma, M \vdash e_1 := e_2 \Downarrow M(h+n \mapsto v)} \quad (\textsc{DynWr})$$

The CCured type system: rules

The type system keeps track of the kind of pointers.
Rules for converting pointers:

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\tau \leq \texttt{int}} \qquad \frac{}{\texttt{int} \leq \tau\ \texttt{ref SEQ}}$$

$$\frac{}{\texttt{int} \leq \texttt{DYNAMIC}}$$

$$\frac{}{\tau\ \texttt{ref SEQ} \leq \tau\ \texttt{ref SAFE}}$$

Typing rules for commands

$\Gamma \vdash c$ means, command c is well-typed.
$\Gamma \vdash e : \tau$ means, expression e has type τ.

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \qquad \frac{\Gamma \vdash e : \tau \ \texttt{ref SAFE} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e'}$$

$$\frac{\Gamma \vdash e : \texttt{DYNAMIC} \quad \Gamma \vdash e' : \texttt{DYNAMIC}}{\Gamma \vdash e := e'}$$

Typing rules for expressions

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \ \texttt{op}\ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash (\tau)e : \tau}$$

$$\frac{}{\Gamma \vdash (\tau \ \texttt{ref SAFE})0 : \tau \ \texttt{ref SAFE}} \qquad \frac{\Gamma \vdash e_1 : \tau \ \texttt{ref SEQ} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \tau \ \texttt{ref SEQ}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{DYNAMIC} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{DYNAMIC}} \qquad \frac{\Gamma \vdash e : \tau \ \texttt{ref SAFE}}{\Gamma \vdash !e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{DYNAMIC}}{\Gamma \vdash !e : \texttt{DYNAMIC}}$$

Safety Policy

The **safety policy** states, that at all times in the execution, the contents of each memory address must correspond to the typing constraints of the home to which it belongs.

Formally, the following predicate must be fulfilled at all times

$$
\begin{aligned}
WF(M_H) \quad \equiv \quad & \forall h \in H^*.\ \forall i \in \mathbb{N}.0 \leq i < \texttt{size}(h) \Rightarrow \\
& (\texttt{kind}(h) = \mathit{untyped} \Rightarrow M(h+i) \in \| \ \texttt{DYNAMIC} \ \|_H \wedge \\
& \ \texttt{kind}(h) = \mathit{typed}(\tau) \Rightarrow M(h+i) \in \| \ \tau \ \|_H
\end{aligned}
$$

We can prove that this property is preserved by all rules in the type system.

Theorems

We separate run-time failure from rightful termination like this: $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of expression e.

Theorems

We separate run-time failure from rightful termination like this: $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of expression e.

Theorem

(Progress and type preservation) If $\Gamma \vdash e : \tau$ and $\Sigma \in \| \Gamma \|_H$ and $WF(M_H)$, then either $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ or $\Sigma, M_H \vdash e \Downarrow v$ and $v \in \| \tau \|_H$.

Theorems

$\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of command c.

Theorems

$\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ means a run-time check failed during the execution of command c.

Theorem

(Progress for commands) If $\Gamma \vdash c$ and $\Sigma \in \parallel \Gamma \parallel_h$ and $WF(M_H)$ then either $\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ or $\Sigma, M_H \vdash c \Longrightarrow M'_H$ and M'_H is well-formed.

Main results

- An efficient inference algorithm attaches `ref SEQ`, `ref SAFE`, `DYNAMIC` annotations to plain C code.
- Most of the checks can be done statically.
- The performance overhead of the remaining run-time checks is moderate: 0–150%
- Purely dynamic checks would incure a performance overhead of factors 6–20
- Several array bounds bugs discovered in SPECINT95

Further Reading

- *CCured: Type-Safe Retrofitting of Legacy Code*, in POPL'02 — ACM Symposium on Principles of Programming Languages, 2002. Online Demo at `http://manju.cs.berkeley.edu/ccured/web/index.html`.

Further Reading

- George Necula, *Proof-carrying code* in POPL'97 — Symposium on Principles of Programming Languages, Paris, France, 1997.
  `http://raw.cs.berkeley.edu/Papers/pcc_popl97.ps`
- George Necula, *Proof-Carrying Code: Design and Implementation* in Proof and System Reliability, Springer-Verlag, 2002.
  `http://raw.cs.berkeley.edu/Papers/marktoberdorf.pdf`
- *CCured Demo*,
  `http://manju.cs.berkeley.edu/ccured/web/index.html`

Main Challenges of PCC

PCC is a very powerful mechanism. Coming up with an efficient implementation of such a mechanism is a challenging task.

The main problems are

- Certificate size
- Size of the trusted code base (TCB)
- Performance of validation
- Certificate generation

Certificate Size

A certificate is a formal proof, and can be encoded as e.g. LF Term.

BUT: such proof terms include a lot of repetition
$\Longrightarrow$ huge certificates

Approaches to reduce certificate size:

- Compress the general proof term and do reconstruction on the consumer side
- Transmit only hints in the certificate (oracle strings)
- Embed the proving infrastructure into a theorem prover and use its tactic language

Size of the Trusted Code Base (TCB)

The PCC architecture relies on the correctness of components such as VC-generation and validation.

But these components are complex and implementation is error-prone.

Approaches for reducing size of TCB:

- Use proven/established software
- Build everything up from basics **foundational PCC** (Appel)

Performance

Even though validation is fast compared to proof generation, it is on the critical path of using remote code
$\Longrightarrow$ performance of the validation is crucial for the acceptance of PCC.

Approaches:

- Write your own specialised proof-checker (for a specific domain)
- Use hooks of a general proof-checker, but replace components with more efficient routines, e.g. arithmetic

LF Terms

The Logical Framework (LF) is a generic description of logics.

- Entities on three levels: objects, families of types, and kinds.
- Signatures: mappings of constants to types and kinds
- Contexts: mappings of variables to types
- Judgements:

$$\Gamma \vdash_{\Sigma} A : K$$

meaning A has kind K in context Γ and signature Σ.

$$\Gamma \vdash_{\Sigma} M : A$$

meaning M has type A in context Γ and signature Σ.

Styles of Program Logics

Two styles of program logics have been proposed.

Styles of Program Logics

Two styles of program logics have been proposed.

- Hoare-style logics: $\{P\}\mathrm{e}\{Q\}$
  Assertions are parameterised over the "current" state.
  Example: Specification of an exponential function

$$\{0 \leq y \ \wedge \ x = X \wedge \ y = Y\} \ \mathtt{exp}(x, y) \ \{r = X^{Y}\}$$

  Note: X, Y are **auxiliary variables** and must not appear in e

Styles of Program Logics

Two styles of program logics have been proposed.

- Hoare-style logics: $\{P\}\mathtt{e}\{Q\}$
  Assertions are parameterised over the "current" state.
  Example: Specification of an exponential function

$$\{0 \leq y \ \wedge \ x = X \wedge \ y = Y\} \ \mathtt{exp}(x, y) \ \{r = X^Y\}$$

  Note: X, Y are **auxiliary variables** and must not appear in e

- VDM-style logics: $\mathtt{e} : P$
  Assertions are parameterised over pre- and post-state.
  Because we have both pre- and post-state in the post-condition we do not need a separate pre-condition.
  Example: Specification of an exponential function

$$\{0 \leq y\} \ \mathtt{exp}(x, y) \ \{r = \grave{x}^{\grave{y}}\}$$

A Simple while-language

Language:

$$
\begin{array}{rcl}
e & ::= & \texttt{skip} \\
 & | & x := t \\
 & | & e_1 \texttt{;} e_2 \\
 & | & \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & | & \texttt{while}\ b\ \texttt{do}\ e \\
 & | & \texttt{call}
\end{array}
$$

A Simple while-language

Language:

$$
\begin{array}{rcl}
e & ::= & \texttt{skip} \\
& | & x := t \\
& | & e_1 ; e_2 \\
& | & \texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2 \\
& | & \texttt{while } b \texttt{ do } e \\
& | & \texttt{call}
\end{array}
$$

A judgement has this form (for now!)

$$\vdash \{P\}\ e\ \{Q\}$$

A judgement is valid if the following holds

$$\forall z\ s\ t.\ s \overset{e}{\leadsto} t \Rightarrow\ P\ z\ s \Rightarrow\ Q\ z\ t$$

A Simple Hoare-style Logic

$$\frac{}{\vdash \{P\}\ \texttt{skip}\ \{P\}} \quad (\text{SKIP}) \qquad \frac{}{\vdash \{\lambda z\ s.\ P\ z\ s[t/x]\}\ x := t\ \{P\}} \quad (\text{ASSIGN})$$

$$\frac{\vdash \{P\}\ e_1\ \{R\} \quad \{R\}\ e_2\ \{Q\}}{\vdash \{P\}\ e_1;e_2\ \{Q\}} \quad (\text{COMP})$$

$$\frac{\vdash \{\lambda z\ s.\ P\ z\ s\ \wedge\ b\ s\}\ e_1\ \{Q\} \quad \vdash \{\lambda z\ s.\ P\ z\ s\ \wedge\ \neg(b\ s)\}\ e_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2\{Q\}} \quad (\text{IF})$$

$$\frac{\vdash \{\lambda z\ s.\ P\ z\ s\ \wedge\ b\ s\}\ e\ \{P\}}{\vdash \{P\}\ \texttt{while}\ b\ \texttt{do}\ e\{\lambda z\ s.\ P\ z\ s\ \wedge\ \neg(b\ s)\}} \quad (\text{WHILE})$$

$$\frac{\vdash \{P\}\ body\ \{Q\}}{\vdash \{P\}\ \texttt{CALL}\ \{Q\}} \quad (\text{CALL})$$

A Simple Hoare-style Logic (structural rules)

The consequence rule allows us to weaken the pre-condition and to strengthen the post-condition:

$$\frac{\forall s\ t.\ (\forall z.\ P'\ z\ s \Rightarrow P\ z\ s) \quad \vdash \{P'\}\ e\ \{Q'\} \quad \forall s\ t.\ (\forall z.\ Q\ z\ s \Rightarrow Q'\ z\ s)}{\vdash \{P\}\ e\ \{Q\}} \quad (\text{CONSEQ})$$

Recursive Functions

In order to deal with recursive functions, we need to collect the knowledge about the behaviour of the functions.

We extend the judgement with a context Γ, mapping expressions to Hoare-Triples:

$$\Gamma \vdash \{P\}\ e\ \{Q\}$$

where Γ has the form $\{\ldots, (P', e', Q'), \ldots\}$.

Recursive Functions

Now, the call rule for recursive, parameter-less functions looks like this:

$$\frac{\Gamma \cup \{(P, \texttt{CALL}, Q)\} \vdash \{P\}\ body\ \{Q\}}{\Gamma \vdash \{P\}\ \texttt{CALL}\ \{Q\}} \qquad (\textsc{call})$$

We collect the knowledge about the (one) function in the context, and prove the body.

Note: This is a rule for partial correctness: for total correctness we need some form of measure.

Recursive Functions

To extract information out of the context we need and axiom rule

$$\frac{(P, e, Q) \in \Gamma}{\Gamma \vdash \{P\}\ e\ \{Q\}} \qquad (\textsc{Ax})$$

Recursive Functions

To extract information out of the context we need and axiom rule

$$\frac{(P, e, Q) \in \Gamma}{\Gamma \vdash \{P\}\ e\ \{Q\}} \qquad (\textsc{Ax})$$

Note that we now use a **Gentzen-style** logic (one with contexts) rather than a Hilbert-style logic.

More Troubles with Recursive Functions

Assume we have this simple recursive program:

```
if i=0 then skip else i := i-1 ; call ; i := i+1
```

-

More Troubles with Recursive Functions

Assume we have this simple recursive program:

```
if i=0 then skip else i := i-1 ; call ; i := i+1
```

The proof of $\{i = N\}$ `call` $\{i = N\}$ proceeds as follows

$$\dfrac{\dfrac{}{}}{\vdash \ \{i = N\}\ \texttt{CALL}\ \{i = N\}}$$

More Troubles with Recursive Functions

Assume we have this simple recursive program:

```
if i=0 then skip else i := i-1 ; call ; i := i+1
```

The proof of $\{i = N\}$ `call` $\{i = N\}$ proceeds as follows

$$\frac{\{(i = N, \mathtt{CALL}, i = N)\} \vdash \ \{i = N\}\ \mathtt{i} := \mathtt{i} - \mathtt{1}; \mathtt{CALL}; \mathtt{i} := \mathtt{i} + \mathtt{1}\ \{i = N\}}{\vdash \ \{i = N\}\ \mathtt{CALL}\ \{i = N\}}$$

More Troubles with Recursive Functions

Assume we have this simple recursive program:

```
if i=0 then skip else i := i-1 ; call ; i := i+1
```

The proof of $\{i = N\}$ `call` $\{i = N\}$ proceeds as follows

$$\cfrac{\cfrac{\{(i = N, \mathtt{CALL}, i = N)\} \vdash \ \{i = N - 1\}\ \mathtt{CALL}\ \{i = N - 1\}}{\{(i = N, \mathtt{CALL}, i = N)\} \vdash \ \{i = N\}\ \mathtt{i := i - 1; CALL; i := i + 1}\ \{i = N\}}}{\vdash \ \{i = N\}\ \mathtt{CALL}\ \{i = N\}}$$

More Troubles with Recursive Functions

Assume we have this simple recursive program:

```
if i=0 then skip else i := i-1 ; call ; i := i+1
```

The proof of $\{i = N\}$ `call` $\{i = N\}$ proceeds as follows

$$\dfrac{\dfrac{\{(i = N, \texttt{CALL}, i = N)\} \vdash \ \{i = N-1\}\ \texttt{CALL}\ \{i = N-1\}}{\{(i = N, \texttt{CALL}, i = N)\} \vdash \ \{i = N\}\ \texttt{i} := \texttt{i} - 1; \texttt{CALL}; \texttt{i} := \texttt{i} + 1\ \{i = N\}}}{\vdash \ \{i = N\}\ \texttt{CALL}\ \{i = N\}}$$

But how can we prove $\{i = N-1\}\texttt{CALL}\{i = N-1\}$ from $\{i = N\}\texttt{CALL}\{i = N\}$?

More Troubles with Recursive Functions

Assume we have this simple recursive program:

```
if i=0 then skip else i := i-1 ; call ; i := i+1
```

The proof of $\{i = N\}$ `call` $\{i = N\}$ proceeds as follows

$$\cfrac{\cfrac{\{(i = N, \mathtt{CALL}, i = N)\} \vdash \ \{i = N - 1\}\ \mathtt{CALL}\ \{i = N - 1\}}{\{(i = N, \mathtt{CALL}, i = N)\} \vdash \ \{i = N\}\ \mathtt{i := i - 1; CALL; i := i + 1}\ \{i = N\}}}{\vdash \ \{i = N\}\ \mathtt{CALL}\ \{i = N\}}$$

But how can we prove $\{i = N - 1\}\mathtt{CALL}\{i = N - 1\}$ from $\{i = N\}\mathtt{CALL}\{i = N\}$?
We need to **instantiate** N with $N - 1$!

Recursive functions

To be able to instantiate auxiliary variables we need a more powerful consequence rule:

$$\frac{\Gamma \vdash \{P'\}\ e\ \{Q'\} \quad \forall s\ t.\ (\forall z.\ P'\ z\ s \Rightarrow Q'\ z\ t) \ \Rightarrow\ (\forall z.\ P\ z\ s \Rightarrow Q\ z\ t)}{\Gamma \vdash \{P\}\ e\ \{Q\}} \quad (\textsc{conseq})$$

Now we are allowed to proof $P \Rightarrow Q$ under the knowledge that we can choose z freely as long as $P' \Rightarrow Q'$ is true.
This complex rule for **adaptation** is one of the main disadvantages of Hoare-style logics.

Extending the Logic with Termination

The Call and While rules need to use a well-founded ordering $<$ and a side condition saying that the body is smaller w.r.t. this ordering:

$$\frac{\begin{array}{c} wf\ < \\ \forall s'.\ \{(\lambda z\ s.P\ z\ s \wedge\ s < s', \texttt{CALL}, Q)\} \\ \vdash_T \{\lambda z\ s.P\ z\ s \wedge\ s = s'\} body\ \{Q\} \end{array}}{\vdash_T \{P\}\ \texttt{CALL}\{Q\}}$$

Note the explicit quantification over the state s'. Read it like this

> *The pre-state s must be smaller than a state s', which is the post-state.*

Extending the Logic with Mutual Recursion

To cover mutual recursion a different derivation system $\vdash_M$ is defined.

Judgements in $\vdash_M$ are extended to sets of Hoare triples, informally:

$$\Gamma \vdash_M \ \{(P_1, e_1, Q_1), \ldots, (P_n, e_n, Q_n)\}$$

The Call rule is generalised as follows

$$\frac{\bigcup p.\ \{(P\ \ p, \texttt{CALL p}, Q\ \ p)\} \vdash_M \bigcup p.\{(P\ \ p, body\ p, Q\ \ p)\}}{\emptyset \vdash_M \bigcup p.\ \{(P\ \ p, \texttt{CALL p}, Q\ \ p)\}}$$

Further Reading

- Thomas Kleymann, *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*, Lab. for Foundations of Computer Science, Univ of Edinburgh, LFCS report ECS-LFCS-98-392, 1999.
  `http://www.lfcs.informatics.ed.ac.uk/reports/98/ECS-LFCS-98-3`
- Tobias Nipkow, *Hoare Logics for Recursive Procedures and Unbounded Nondeterminism*, in CSL 2002 — Computer Science Logic, LNCS 2471, pp. 103–119, Springer, 2002.

Challenge: Minimising the TCB

This aspect is the emphasis of the **Foundational PCC** approach.

An infrastructure developed by the group of Andrew Appel at Princeton [1].

Motivation: With complex logics and VCGs, there is a big danger of introducing bugs in software that needs to be trusted.

Validator

What exactly is proven?

The safety policy is typically encoded as a pre-post-condition pair (P/Q) for a program e, and a logic describing how to reason.

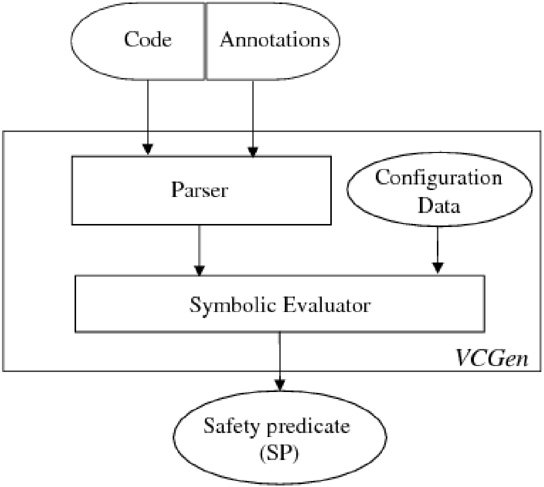
Running the verification condition generator VCG over e and Q, generates a set of conditions, that need to be fulfilled in order for the program to be safe.

The condition that needs to be proven is:

$$P \Longrightarrow VC(e, Q)$$

.

Structure of the VCG



The Philosophy of Foundational PCC

Define safety policy directly on the **operational semantics** of the code.

Certificates are proofs over the operational semantics.

It minimises the TCB because no trusted verification condition generator is needed.

Pros and cons:

- ☺ **more flexible**: not restricted to a particular type system as the language in which the proofs are phrased;
- ☺ **more secure**: no reliance on VCG.
- ☹ **larger proofs**

Conventional vs Foundational PCC

Re-examine the logic for memory safety, eg.

$$\frac{m \vdash e : \tau\ \mathit{list} \qquad e \neq 0}{\begin{array}{l} m \vdash e : \mathit{addr}\ \wedge\ m \vdash e+4 : \mathit{addr}\ \wedge \\ m \vdash \mathit{sel}(m,e) : \tau\ \wedge\ m \vdash \mathit{sel}(m,e+4) : \tau\ \mathit{list} \end{array}} \quad (\textsc{ListElim})$$

The rule has **built-in knowledge about the type-system**, in this case representing the data layout of the compiler ("*Type specialised PCC*") $\Longrightarrow$ dangerous if soundness of the logic is not checked mechanically!

Logic rules in Foundational PCC

In foundational PCC the rules work on the operational semantics:

$$\frac{m \models e : \tau\ list \quad e \neq 0}{\begin{array}{l} m \models e : addr \ \wedge\ m \models e+4 : addr\ \wedge \\ m \models sel(m,e) : \tau\ \wedge\ m \models sel(m,e+4) : \tau\ list \end{array}} \ (\textsc{ListElim})$$

This looks similar to the previous rule but has a very different meaning: $\models$ is a predicate over the formal model of the computation, and the above rule can be proven as a lemma, $\vdash$ is an encoding of a type-system on top of the operational semantics and thus needs a **soundness proof**.

Components of a foundational PCC infrastructure

Operational semantics and safety properties are directly encoded in a **higher-order logic**.

As language for the certificates, the LF metalogic framework is used.

For development and for proof-checking the Twelf theorem proofer is used.

Specifying safety

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Specifying safety

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: operational semantics on assembler code.
Safety policy: "only readable addresses are loaded".
Define a predicate: $\mathit{readable}(x) \equiv 0 \leq x \leq 1000$

Specifying safety

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: operational semantics on assembler code.
Safety policy: "only readable addresses are loaded".
Define a predicate: $\mathit{readable}(x) \equiv 0 \leq x \leq 1000$
The semantics of a load operation `LD ri, c(rj)` is now written as follows:

$$\begin{array}{rcl} \mathit{load}(i,j,c) & \equiv & \lambda\ r\ m\ r'\ m'. \\ & & r'(i) = m(r(j)+c)\ \wedge\ \mathit{readable}(r(j)+c)\ \wedge \\ & & (\forall x \neq i.\ r'(x) = r(x))\ \wedge\ m' = m \end{array}$$

Specifying safety

To specify safety, the operational semantics is written in such a way, that it gets stuck whenever the safety condition is violated.

Example: operational semantics on assembler code.
Safety policy: "only readable addresses are loaded".
Define a predicate: $\mathit{readable}(x) \equiv 0 \leq x \leq 1000$
The semantics of a load operation $\mathtt{LD}\ \mathtt{r_i}, \mathtt{c}(\mathtt{r_j})$ is now written as follows:

$$\begin{array}{rcl} \mathit{load}(i,j,c) & \equiv & \lambda\ r\ m\ r'\ m'. \\ & & r'(i) = m(r(j)+c)\ \wedge\ \mathit{readable}(r(j)+c)\ \wedge \\ & & (\forall x \neq i.\ r'(x) = r(x))\ \wedge\ m' = m \end{array}$$

Note: the clause for nothing else changes, quickly becomes awkward when doing these proofs
$\Longrightarrow$ Separation Logic (Reynolds'02) tackles this problem.

Further Reading

- Andrew Appel, *Foundational Proof-Carrying Code* in LICS'01 — Symposium on Logic in Computer Science, 2001.
  `http://www.cs.princeton.edu/~appel/papers/fpcc.pdf`