Proof-Carrying-Code

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

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

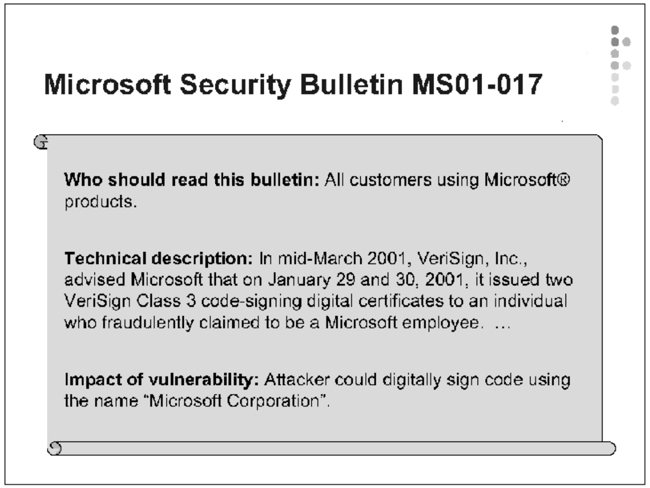
Windows:

- 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 device 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.

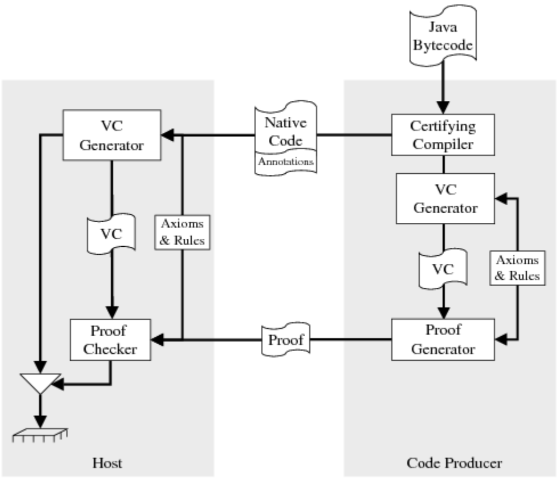
> *PCC is a software mechanism that allows a host system to determine with certainty that it is safe to execute a program supplied by an untrusted source.*

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

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”.

Examples:

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

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.

- 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:

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

A judgement has this form (for now!)

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

A judgement is valid if the following holds

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

A Simple Hoare-style Logic

$$\frac{}{\vdash \{P\}\ \texttt{skip}\ \{P\}}\ (\textsc{Skip}) \qquad \frac{}{\vdash \{\lambda z\ s.\ P\ z\ s[t/x]\}\ x := t\ \{P\}}\ (\textsc{Assign})$$

$$\frac{\vdash \{P\}\ e_1\ \{R\} \quad \{R\}\ e_2\ \{Q\}}{\vdash \{P\}\ e_1; e_2\ \{Q\}}\ (\textsc{Comp})$$

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

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

$$\frac{\vdash \{P\}\ body\ \{Q\}}{\vdash \{P\}\ \texttt{CALL}\ \{Q\}}\ (\textsc{Call})$$

A Simple Hoare-style Logic (structural rules)

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

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

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:

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

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

$$\frac{(P, e, Q) \in \Gamma}{\Gamma \vdash \{P\}\ e\ \{Q\}} \qquad \text{(AX)}$$

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
```

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

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

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

Recursive functions

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

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

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.

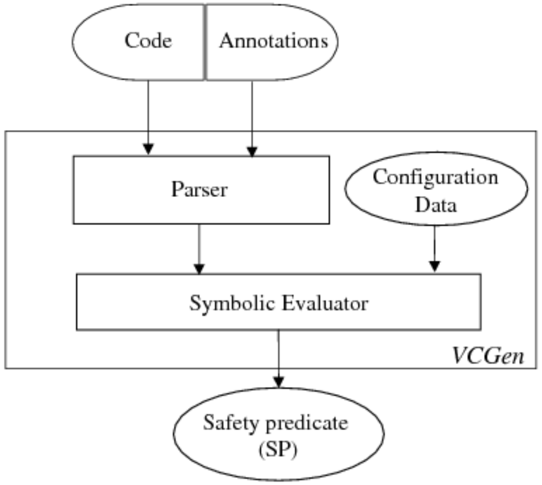
Running the verification condition generator VCG over e and Q, generates a set of conditions, that need to be fulfilled in order for the program to be safe.

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.

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

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.

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

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

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`

PCC for Resources: Motivation

Resource-bounded computation is one specific instance of PCC.

Safety policy: resource consumption is lower than a given bound.

Resources can be (heap) space, time, or size of parameters to system calls.

Strong demand for such guarantees for example in embedded systems.

Mobile Resource Guarantees

Objective:
Development of an infrastructure to endow mobile code with independently verifiable certificates describing resource behaviour.

Approach:
Proof-carrying code for **resource-related properties**, where proofs are generated from typing derivations in a **resource-aware type system**.

Motivation

Restrict the execution of mobile code to those adhering to a certain resource policy.

Application Scenarios:

- A user of a **handheld device** might want to know that a downloaded application will definitely run within the limited amount of memory available.
- A provider of **computational power in a Grid infrastructure** may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption.

Proof-Carrying-Code with High-Level-Logics

Our approach to PCC: Combine high-level type-systems with program logics and build a **hierarchy of logics** to construct a logic tailored to reason about resources.

Everything is **formalised in a theorem prover**.

Classic vs Foundational PCC: best of both worlds

- **Simple reasoning**, using specialised logics;
- **Strong foundations**, by encoding the logics in a theorem prover

Proof-Carrying-Code with High-Level-Logics

High-Level Type System	$G \vdash_H t : \tau$
Specialised Logic	$\rhd \ulcorner t \urcorner : D(G, \tau)$
Termination Logic	$\vdash_T \{P\}\ e \downarrow$
Program Logic	$\Gamma \rhd e : A$
Operational Semantics	$E \vdash h, e \Downarrow (h', v, p)$

Motivating Example of this Hierarchical Approach

High-level language: ML-like.

Safety policy: well-formed datatypes.

Define a predicate $h \models_t a$, expressing that an address a in heap h is the start of a (high-level) data-type t.

Prove: $f :: \tau\ list \rightarrow \tau\ list$ adheres to this safety policy.

Directly on the program logic

$$\rhd\ f(x)\ :\ \lambda E\ h\ h'\ v\ .\ h \models_{list} E\langle x\rangle \ \longrightarrow\ h' \models_{list} v$$

NOT: reasoning on this level generates huge side-conditions.

Motivating Example of this Hierarchical Approach

Instead, define a higher-level logic $\vdash_H$ that abstracts over the details of datatype representation, and that has the property

$$G \vdash_H t : \tau \implies \rhd \ulcorner t \urcorner : D(\Gamma, \tau)$$

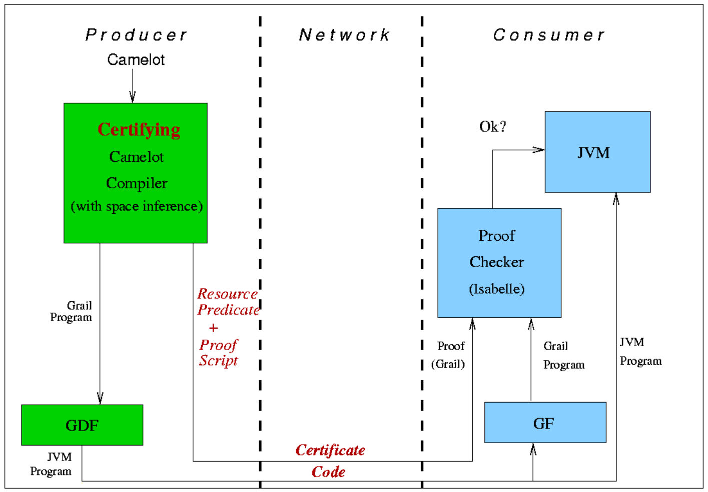
We specialise the form of assertions like this

$$\begin{array}{lll} D(\{x : list, y : list\}, list) & \equiv & \\ \lambda E\ h\ h'\ v. & h \models_{list} E\langle x\rangle \wedge h \models_{list} E\langle y\rangle \longrightarrow & \\ & h' \models_{list} E\langle x\rangle \wedge h' \models_{list} E\langle y\rangle \wedge h' \models_{list} v & \end{array}$$

Now we can formulate rules, that match translations from the high-level language:

$$\frac{\rhd \ulcorner t_1 \urcorner : D(\Gamma, \tau) \qquad \rhd \ulcorner t_2 \urcorner : D(\Gamma, \tau\ list)}{\rhd \ulcorner cons(t_1, t_2) \urcorner : D(\Gamma, \tau\ list)}$$

A Proof-Carrying-Code Infrastructure for MRG



Camelot

- Strict, first-order functional language with CAML-like syntax and object-oriented extensions
- Compiled to subset of JVM (Java Virtual Machine) bytecode (Grail)
- Memory model: 2 level heap
- Security: Static analyses to prevent deallocation of live cells in Level-1 Heap: linear typing (folklore + Hofmann), readonly typing (Aspinall, Hofmann, Konencny), layered sharing analysis (Konencny).
- Resource bounds: Static analysis to infer linear upper bounds on heap consumption (Hofmann, Jost).

Example: Insertion Sort

Camelot program:

```
let ins a l = match l with
                Nil -> Cons(a,Nil)
              | Cons(x,t)@_ -> if a < x then Cons(a,Cons(x,t))
                                         else Cons(x, ins a t)

let sort l =  match l with  Nil -> Nil
                          | Cons(a,t)@_ -> ins a (sort t)
```


In-place Operations via a Diamond Type

Using operators, such as `Cons`, amounts to heap allocation.

Additionally, Camelot provides extensions to do **in-place operations** over arbitrary data structures via a so called **diamond type** $\diamond$ with **d** $\in \diamond$:

```
match l with Nil@d => e1
           | Cons (h,t)@d => ... Cons (x,t)@d ...
```

The memory occupied by the cons cell can be **re-used** via the diamond `d`.
Note:

- $\diamond$ is an abstract data-type
- structured use of diamonds in branches of pattern matches

How does this fit with referential transparency?

Using a diamond type, we can introduce side effects:

```
type ilist = Nil | Cons of int*ilist
let insert1 x l =
    match l with Nil -> Cons (x, l)
               | Cons(h,t)@d ->
                   if x <= h then Cons(x, Cons(h,t)@d)
                             else Cons(h, insert1 x t)@d

let sort l =  match l with Nil -> Nil
                         | Cons(h,t) -> insert1 h (sort t)
```

Now, what's the result of

```
let start args = let l = [4,5,6,7] in
                 let l1 = insert1 6 l in
                 print_list l
```


Linearity saves the day

We can characterise the class of programs for which referential transparency is retained.

Theorem

*A **linearly typed** Camelot program computes the function specified by its purely functional semantics (Hofmann, 2000).*

Beyond Linearity

But: linearity is too restrictive in many cases; we also want to use diamonds in programs where **only the last access to the data structure is destructive**.

More expressive type systems to express such access patterns are **readonly types** (Aspinall, Hofmann, Konecny, 2001) and types with **layered sharing** (Konecny 2003).

As with pointers, diamonds can be a powerful gun to shoot yourself in the foot. We need a **powerful type system** to prevent this, and want a **static analysis** to predict resource consumption.

Space Inference

Goal: Infer a linear upper bound on heap consumption.

Given Camelot program containing a function

```
start : string list -> unit
```

find **linear function** s such that `start`(l) will not call `new()` (only `make()`) when evaluated in a heap h where

- **the freelist has length not less than** $s(n)$
- l points in h to a linear list of some length n
- the freelist which forms a part of h is well-formed
- the freelist does not overlap with l

Composing `start` with *runtime environment* that binds input to, e.g., `stdin` yields a standalone program that runs within predictable heap space.

Extended (LFD) Types

Idea: Weights are attached to constructors in an extended type-system.

```
ins        : 1, int -> list(...<0>) -> list(...<0>), 0
```

says that the call `ins x xs` requires 1 heap-cell plus 0 heap cells for each `Cons` cell of the list `xs`.

```
sort       : 0, list(...<0>) -> list(...<0>), 0
```

`sort` does not consume any heap space.

```
start      : 0, list(...<1>) -> unit, 0;
```

gives rise to the desired linear bounding function $s(n) = n$.

High-level Type System: Function Call

A, B, C are types, $k, k', n, n' \in \mathbb{Q}^+$, $\mathtt{f}$ is a Camelot function and $\mathtt{x}_1, \ldots, \mathtt{x}_p$ are variables, Σ is a table mapping function names to types.

$$\frac{\Sigma(\mathtt{f}) = (A_1, \ldots, A_p, k) \longrightarrow (C, k') \qquad n \geq k \qquad n - k + k' \geq n'}{\Gamma, \mathtt{x}_1 : A_1, \ldots, \mathtt{x}_p : A_p, n \vdash \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_p) : C, n'} \quad (\textsc{Fun})$$

Grail

Grail is an abstraction over virtual machine languages such as the JVM.

$$
\begin{array}{lcl}
e \in expr & ::= & \texttt{null} \mid \texttt{int}\ i \mid \texttt{var}\ x \mid \texttt{prim}\ p\ x\ x \mid \texttt{new}\ c\ [t_1 := x_1, \ldots, t_n := x_n] \mid \\
 & & x.t \mid x.t{:=}x \mid c \diamond t \mid c \diamond t{:=}x \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e\ ;\ e \mid \\
 & & \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{call}\ f \mid x \cdot m(\overline{a}) \mid c \diamond m(\overline{a}) \\
a \in args & ::= & \texttt{var}\ x \mid \mathrm{null} \mid i
\end{array}
$$

Example: Insertion sort

Grail code:

```
method static public List ins (int a, List l) = ...Make(..,..,..
method static public List sort (List l) =
  let fun f(List l) =
    if l = null then null
                else let val h = l.HD
                         val t = l.TL
                         val () = D.free (l)
                         val l = List.sort (t)
                     in List.ins (h, l) end
  in f(l) end
```

This is a 1-to-1 translation of JVM code

Judgement of the Operational Semantics

Modelling resources: Resources are an extra component in operational and axiomatic semantics (“resource record”).

$$\mathbf{p} \in RRec = (\!| clock : nat, callcount : nat, invokedepth : nat, maxstack : nat |\!)$$

A judgement in the functional operational semantics

$$E \vdash h, e \Downarrow_n (h', v, p)$$

is to be read as “starting with a heap h and a variable enviroment E, the Grail code e evaluates in n steps to the value v, yielding the heap h' as result and consuming p resources.”

Operational Semantics: Let- and Call-rules

$$\frac{E \vdash h, e_1 \Downarrow_n (h_1, w, p) \quad w \neq \bot \quad E\langle x := w\rangle \vdash h_1, e_2 \Downarrow_m (h_2, v, q)}{E \vdash h, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow_{max(n,m)+1} (h_2, v, \mathbf{p_1} \smile \mathbf{p_2})} \quad (\textsc{Let})$$

$$\frac{E \vdash h, body_f \Downarrow_n (h_1, v, p)}{E \vdash h, \texttt{call}\ f \Downarrow_{n+1} (h_1, v, \langle \mathbf{1\ 1\ 0\ 0} \rangle \oplus \mathbf{p_1})} \quad (\textsc{Call})$$

A Program Logic for Grail

VDM-style logic with judgements of the form $\Gamma \ \triangleright\ e : A$, meaning

"in context Γ expression e fulfills the assertion A"

Type of assertions (**shallow embedding**):

$$\mathcal{A} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{R} \to \mathcal{B}$$

No syntactic separation into pre- and postconditions.

Semantic validity $\models e : A$ means

"whenever $E \vdash h, e \Downarrow (h', v, p)$ then $A\ E\ h\ h'\ v\ p$ holds"

Note: Covers partial correctness; termination orthogonal.

A Program Logic for Grail

Simplified rule for parameterless function call:

$$\frac{\Gamma, (\texttt{Call f} : A) \ \triangleright\ \texttt{e} \ :\ A^{+}}{\Gamma\ \triangleright\ \texttt{Call f}\ : A} \qquad (\textsc{CallRec})$$

where `e` is the body of the function `f` and

$$A^{+} \equiv \lambda\, E\, h\, h'\, v\, p.\, A(E, h, h', v, p^{+})$$

where p^{+} is the updated cost component.
Note:

- Context Γ: collects hypothetical judgements for recursion
- Meta-logical guarantees: soundness, relative completeness

Program Logic Rules

$$\frac{\Gamma \triangleright e_1 : P \quad \Gamma \triangleright e_2 : Q}{\Gamma \triangleright \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \lambda\, E\, h\, h'\, v\, p.\, \exists\, p_1\, p_2\, h_1\, w.\ \ \begin{array}{l} P\ E\ h\ h_1\ w\ p_1\ \wedge\ w \neq \bot\ \wedge \\ Q\ (E\langle x := w\rangle)\ h_1\ h'\ v\ p_2)\ \wedge \\ p = \mathbf{p_1} \smile \mathbf{p_2} \end{array}} \qquad (\textsc{VLet})$$

$$\frac{\Gamma \cup \{(\texttt{call}\ f, P)\} \triangleright body_f : \lambda\, E\, h\, h'\, v\, p.\, P\ E\ h\ h'\ v\ \langle \mathbf{1\ 1\ 0\ 0}\rangle \oplus \mathbf{p_1},}{\Gamma \triangleright \texttt{call}\ f : A} \qquad (\textsc{VCall})$$

Specific Features of the Program Logic

- Unusual rules for **mutually recursive methods** and for **parameter adaptation** in method invocations

$$\frac{(\Gamma, \texttt{e} : A)\ \ goodContext}{\triangleright\, \texttt{e} : A} \qquad (\textsc{MutRec})$$

$$\frac{(\Gamma, c \diamond m(\overline{a}) : MS\ c\ m\ \overline{a})\ \ goodContext}{\triangleright c \diamond m(\overline{b}) : MS\ c\ m\ \overline{b}} \qquad (\textsc{Adapt})$$

- Proof via admissible Cut rule, **no extra derivation system**
- Global specification table *MS*, *goodContext* relates entries in *MS* to the method bodies

Example: Insertion sort

Specification:

$$
\begin{array}{lll}
\mathit{insSpec} & \equiv & \mathit{MS}\ \text{List ins}\ [a_1, a_2] = \\
& & \quad \lambda\, E\, h\, h'\, v\, p . \forall\, i\, r\, n\, X\, . \\
& & \quad\ (E\langle a_1\rangle = i \wedge E\langle a_2\rangle = \text{Ref}\, r \wedge h, r \models_X n \\
& & \quad\ \longrightarrow |\mathit{dom}\,(h)| + 1 = |\mathit{dom}\,(h')| \wedge \\
& & \qquad p \leq \ldots) \\
\mathit{sortSpec} & \equiv & \mathit{MS}\ \text{List sort}\ [a] = \\
& & \quad \lambda\, E\, h\, h'\, v\, p . \forall\, i\, r\, n\, X\, . \\
& & \quad\ (\ E\langle a\rangle = \text{Ref}\, r \wedge h, r \models_X n \ \longrightarrow |\mathit{dom}\,(h)| = |\mathit{dom}\,(h')|\ \wedge\ p \leq \ldots)
\end{array}
$$

Lemma: $\mathit{insSpec}\ \wedge\ \mathit{sortSpec} \longrightarrow\ \rhd \text{List} \diamond \mathit{sort}([\mathit{xs}]) : \mathit{MS}\ \text{List}\ \mathit{sort}\ [\mathit{xs}]$

Discussion of the Program Logic

- Expressive logic for correctness and resource consumption
- Logic proven **sound and complete**
- Termination built on top of a logic for partial correctness
- Less suited for immediate program verification: not fully automatic (case-splits, $\exists$-instantiations,...), verification conditions large and complex
- Continue abstraction: loop unfolding in op. semantics $\rightarrow$ invariants in general program logics $\rightarrow$ specific logic for interesting (resource-)properties
- Aim: exploit structure of Camelot compilation (freelist) and program analysis

$$
\begin{array}{rcl}
\text{List.ins} & : & 1, \mathbf{I} \times \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0 \\
\text{List.sort} & : & 0, \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0
\end{array}
$$

Heap Space Logic (LFD-assertions)

- Translation of Hofmann-Jost type system to Grail, types interpreted as relating initial to final freelist
- Fixed assertion format $[\![U, n, [\Delta] \blacktriangleright T, m]\!]$

$$
\begin{array}{rcl}
\text{List.ins} & : & [\![\{a, l\}, 1, [a \mapsto \mathbf{I}, l \mapsto \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \\
\text{List.sort} & : & [\![\{l\}, 0, [l \mapsto \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!]
\end{array}
$$

- LFD types express space requirements for datatype constructors, numbers n, m refer to the freelist length
- Semantic definition by expansion into core bytecode logic, derived proof rules using linear affine context management
- Dramatic reduction of VC complexity!

Semantic interpretation of $[\![U, n, [\Delta] \blacktriangleright T, m]\!]$

$$
\begin{array}{l}
[\![U, n, [\Delta] \blacktriangleright T, m]\!] \equiv \\
\lambda\, E\, h\, h'\, v\, p. \\
\ \ \forall\, F\, N.\quad (\mathit{regionsExist}(U, \Delta, h, E) \wedge \mathit{regionsDistinct}(U, \Delta, h, E) \wedge \\
\qquad\qquad\ \mathit{freelist}(h, F, N) \wedge \mathit{distinctFrom}(U, \Delta, h, E, F)) \\
\qquad\qquad \longrightarrow \\
\qquad\qquad (\exists\, R\, S\, M\, G.\ \ v, h' \models_T R, S\ \wedge \mathit{freelist}(h', G, M) \wedge R \cap G = \emptyset \wedge \\
\qquad\qquad\qquad\qquad \mathit{Bounded}((R \cup G), F, U, \Delta, h, E) \wedge \mathit{modified}(F, U, \Delta, h, E, h') \wedge \\
\qquad\qquad\qquad\qquad \mathit{sizeRestricted}(n, N, m, S, M, U, \Delta, h, E) \wedge \mathit{dom}\, h = \mathit{dom}\, h')
\end{array}
$$

- Formulae defined by BC expansion:

$$
\begin{array}{l}
\mathit{regionsDistinct}(U, \Delta, h, E) \equiv \\
\quad \forall\, x\, y\, R_x\, R_y\, S_x\, S_y . \\
\qquad (\{x, y\} \subseteq U \cap \mathit{dom}\, \Delta \wedge x \neq y \wedge E\langle x\rangle, h \models_{\Delta(x)} R_x, S_x \wedge E\langle y\rangle, h \models_{\Delta(y)} R_y, S_y) \\
\qquad \longrightarrow R_x \cap R_y = \emptyset \\
\mathit{sizeRestricted}(n, N, m, S, M, U, \Delta, h, E) \equiv \\
\quad \forall\, q\, C.\ \mathit{Size}(E, h, U, \Delta, C) \wedge n + C + q \leq N \longrightarrow m + S + q \leq M
\end{array}
$$

- **You don't want to read this — and you don't need to!**

Proof System

Proof system with linear inequalities and linear affine type system (U, Δ) that guarantees benign sharing;

$$\frac{\Delta(x) = T \quad n \leq m}{\Gamma \rhd \texttt{var}\ x : [\![\{x\}, m, [\Delta] \blacktriangleright T, n]\!]} \quad (\textsc{Var})$$

$$\frac{\begin{array}{ll}\Gamma \rhd e_1 : [\![U_1, n, [\Delta] \blacktriangleright T_1, m]\!] & \Gamma \rhd e_2 : [\![U_2, m, [\Delta, x \mapsto T_1] \blacktriangleright T_2, k]\!] \\ U_1 \cap (U_2 \setminus \{x\}) = \emptyset & T_1 = \mathbf{L}(\_)\end{array}}{\Gamma \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : [\![U_1 \cup (U_2 \setminus \{x\}), n, [\Delta] \blacktriangleright T_2, k]\!]} \quad (\textsc{Let})$$

$$\frac{\Delta(x) = \mathbf{L}(k) \quad l = n + k \quad \Gamma \rhd e : [\![U, l, [\Delta, t \mapsto \mathbf{L}(k)] \blacktriangleright T, m]\!] \quad x \notin U \setminus \{t\}}{\Gamma \rhd \texttt{let}\ t = x.TL\ \texttt{in}\ e : [\![(U \setminus \{t\}) \cup \{x\}, n, [\Delta] \blacktriangleright T, m]\!]} \quad (\textsc{LetTL})$$

Note: Linearity relaxed in rules for compiled `match`-expressions

Discussion of the Heap Space Logic

- ☺ Exploit program structure and compiler analysis: most effort done once (in soundness proofs), application straight-forward
- ☺ “Classic PCC”: independence of derived logic from Isabelle (no higher-order predicates, certifying constraint logic programming)
- ☺ “Foundational PCC”: can unfold back to core logic and operational semantics if desired
- ☹ Generalisation to all Camelot datatypes needed
- ☹ Soundness proofs non-trivial, and challenging to generalise to more liberal sharing disciplines

Certificate Generation

Goal: Automatically generate proofs from high-level types and inferred heap consumption.

Approach: Use inferred space bounds as invariants. Use powerful Isabelle tactics to automatically prove a statement on heap consumption in the heap logic.

Example certificate (for list append):

$\Gamma \rhd$ snd (methtable Append append) : SPEC append
by (Wp append_pdefs)

$\rhd$Append.append([RNarg x0, RNarg x1]) : sMST Append append [RNarg x0, RNarg x1]
by (fastsimp intro: Context_good GCInvs simp: ctxt_def)

Summary

MRG works towards **resource-safe global computing**:

- **check resource consumption** before executing downloaded code;
- **automatically generate certificate** out of a Camelot type.

Components of the picture

- Proof-Carrying-Code infrastructure
- Inference for space consumption in Camelot
- Specialised derived assertions on top of a general program logic for Grail
- Certificate = proof of a derived assertion
- Certificate generation from high-level types

Further Reading

- David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella and Ian Stark, *Mobile Resource Guarantees for Smart Devices* in CASSIS04 — Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, LNCS 3362, 2005.
  `http://groups.inf.ed.ac.uk/mrg/publications/mrg/cassis2004.pdf`
- David Aspinall and Lennart Beringer and Martin Hofmann and Hans-Wolfgang Loidl and Alberto Momigliano, *A Program Logic for Resource Verification*, in TPHOLs2004 — International Conference on Theorem Proving in Higher Order Logics, Utah, LNCS 3223, 2004.
- Martin Hofmann, Steffen Jost, *Static Prediction of Heap Space Usage for First-Order Functional Programs*, in POPL'03 — Symposium on Principles of Programming Languages, New Orleans, LA, USA, Jan 2003.

Further Reading

- K. Crary and S. Weirich, *Resource Bound Certification* in POPL'00 — Symposium on Principles of Programming Languages, Boston, USA, 2000.
  `http://www-2.cs.cmu.edu/ crary/papers/1999/res/res.ps.gz`
- Gilles Barthe, Mariela Pavlova, Gerardo Schneider, *Precise analysis of memory consumption using program logics* in International Conference on Software Engineering and Formal Methods (SEFM 2005), 7–9 September 2005, Koblenz, Germany.
  `http://www-sop.inria.fr/everest/soft/Jack/doc/papers/gmg05.pd`

Summary

PCC is a powerful, general mechanism for providing safety guarantees for mobile code.

It provides these guarantees without resorting to a trust relationship.

It uses techniques from the areas of type-systems, program verification and logics.

It is a very active research area at the moment.

Current Trends

Using formal methods to check specific program properties.

- Program logics as the basic language for doing these checks attract renewed interest in PCC.
- A lot of work on program logics for low-level languages.
- Immediate applications for smart cards and embedded systems.

Future Directions

Embedded Systems as a domain for formal methods.

- Some of these systems have strong security requirements.
- Formal methods are used to check these requirements.
- Model checking is a very active area for automatically checking properties.

Links to other areas

Checking program properties is closely related to inferring quantitative information.

- **Static analyses** deal with extracting quantitative information (e.g. resource consumption)
- A lot of research has gone into making these techniques efficient.
- **Model checking** can deal with a larger class of problems (e.g. specifying safety conditions in a system)
- Just recently these have become efficient enough to be used for main stream programming.

Reading List:

`http://www.tcs.ifi.lmu.de/~hwloidl/PCC/reading.html`