Proof-Carrying-Code

Applying formal methods in a distributed world

Hans-Wolfgang Loidl

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

July 1, 2005

1. PCC for Resources
2. Camelot: Our High-level Language
3. Space Inference
4. Grail: Our intermediate language
5. A Program Logic for Grail
6. Heap Space Logic
7. Summary

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**.

Project partners: LFCS, Univ of Edinburgh (D. Sannella) and Inst Informatik, LMU Univ, Munich (M. Hofmann). This work is funded by the EU under the IST-FET project *Mobile Resource Guarantees* No. IST-2001-33149.

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	$G \vdash_{\mathsf{H}} t : \tau$
Specialised Logic	$\rhd \ulcorner t \urcorner : D(G, \tau)$
Termination Logic	$\vdash_{\mathsf{T}} \{P\} \;\; \mathsf{e} \;\; \downarrow$
Program Logic	$\Gamma \;\; \rhd \;\; \mathsf{e} : A$
Operational Semantics	$E \vdash h, \mathsf{e} \Downarrow (h', v, p)$

Motivating Example of this Hierarchical Approach

High-level language: ML-like.

Motivating Example of this Hierarchical Approach

High-level language: ML-like.

Safety policy: well-formed datatypes.

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.

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: $f :: \tau\ \mathit{list} \rightarrow \tau\ \mathit{list}$ adheres to this safety policy.

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

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

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: $f :: \tau\ \mathit{list} \rightarrow \tau\ \mathit{list}$ adheres to this safety policy.

Directly on the program logic

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

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

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

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

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

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

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

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

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

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)
```

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**.

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).

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.

$$\frac{\Sigma(\mathtt{f}) = (A_1, \ldots, A_p, k) \longrightarrow (C, k') \quad n \geq k \qquad n - k + k' \geq n'}{\Gamma, \mathtt{x_1} : A_1, \ldots, \mathtt{x_p} : A_p, n \vdash \mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_p}) : C, n'} \quad (\text{Fun})$$

Grail

Characteristics of Grail (**G**uaranteed **R**esource **A**ware **I**ntermediate **L**anguage):

- Abstract representation of virtual machine languages
- Language of dual identity: (impure) functional semantics and (object-oriented) imperative semantics via expansion to virtual machine code
- Syntactic restrictions on functions to obtain coincidence of semantics: no nesting; only tail-calls; λ-lifted; arguments and parameters must match
- Operational semantics with cost model $E \vdash h, e \Downarrow (h', v, p)$ relating expression e, environment E, (pre-)heap h, result v, (post-)heap h' and cost component

$$p = \langle clock \quad callc \quad invkc \quad invkdpth \rangle.$$

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

Grail: Syntax

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

Grail: Semantic Domains

$$
\begin{array}{l}
l \in Loc \equiv nat \\
r \in Ref ::= \texttt{null} \mid \texttt{ref}\ Loc \\
v \in Val \equiv int \cup Ref \cup \{\bot\} \\
\eta \in Env \equiv (iname \rightharpoonup int) \uplus (rname \rightharpoonup Ref) \\
h \in Heap \equiv (Loc \rightharpoonup cname) \\
(ifldname \to Loc \to int)(rfldname \to Loc \to Ref) \\
\mathbf{p} \in RRec \equiv nat \times nat \times nat \times nat
\end{array}
$$

Grail: Modelling resources

Resources are an extra component in operational and axiomatic semantics ("resource record").

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

Grail: Modelling resources

Resources are an extra component in operational and axiomatic semantics ("resource record").

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

We use the following shorthand notation: $\langle 1\ 0\ 0\ 0 \rangle$
Operations on resource vectors are $\oplus$, as component-wise addition, and $\smile$:

$$(c, cc, id, ms) \smile (c', cc', id', ms') = (c + c', cc + cc', id + id', \max(ms, ms'))$$

Grail: Modelling resources

Resources are an extra component in operational and axiomatic semantics ("resource record").

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

We use the following shorthand notation: $\langle 1\ 0\ 0\ 0 \rangle$
Operations on resource vectors are $\oplus$, as component-wise addition, and $\smile$:

$$(c, cc, id, ms) \smile (c', cc', id', ms') = (c + c', cc + cc', id + id', \max(ms, ms'))$$

Resource vectors can be generalised to abstract operations **resource algebras**.

Judgement of the Operational Semantics

Two semantics for Grail:

- imperative: small-step call-by-value semantics, using a big state structure;
- functional: big-step call-by-value semantics, with side-effecting operations;

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

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

Operational Semantics: Let- and Call-rules

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

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

A Program Logic for Grail

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

"in context Γ expression e fulfills the assertion A"

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

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

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:

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

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

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

Program Logic Rules

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

$$\frac{\Gamma \cup \{(\texttt{call}\ f, P)\} \rhd body_f : \lambda E\, h\, h'\, v\, p.\, P\ E\ h\ h'\ v\ \langle \mathbf{1\ 1\ 0\ 0} \rangle \oplus \mathbf{p_1},}{\Gamma \rhd \texttt{call}\ f : A} \quad (\text{VCALL})$$

Specific Features of the Program Logic

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

$$\frac{(\Gamma, \mathrm{e} : A) \;\; \mathit{goodContext}}{\rhd\, \mathrm{e} : A} \qquad (\textsc{Mutrec})$$

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

- 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:

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

Lemma:

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

Definition of Termination

Goal: Put termination on top of the core logic without changing it.

Termination for a given state E, h and expression e means that a final state exists in the semantics.

Semantic definition of termination of e under precondition P:

$$\{P\}\ \mathsf{e}\ \downarrow \equiv\ \forall\ E\ h\ .\ P\ E\ h\ \longrightarrow \exists\ h'\ v\ p\ .\ E \vdash h, \mathsf{e} \Downarrow (h', v, p)$$

Type of a precondition: $\mathcal{P} \equiv \mathcal{E} \rightarrow \mathcal{H} \rightarrow \mathcal{B}$.

The "rules" of the termination logic are lemmas derived from the Grail Logic.

Selected Rules of the Termination Logic

$$\frac{\forall E\ h.\ P\ E\ h\ \longrightarrow E\lfloor x \rfloor \neq \mathsf{null}}{\{P\}\ x.f\ \downarrow} \qquad (\textsc{GetF})$$

Note that the operational semantics gets stuck in case of a null-reference.

$$\frac{\{P\}\ \mathtt{e}\ \downarrow \quad \{P'\}\ \mathtt{e}'\ \downarrow \quad \rhd\ \mathtt{e}\ :\ P \longrightarrow_{\lfloor x:= \rfloor} P'}{\{P\}\ \mathtt{let}\ x = e\ \mathtt{in}\ e'\ \downarrow} \qquad (\textsc{Let})$$

We use the following combinators to capture bindings in lets and express the side condition in terms of a VDM assertion:

$$P \longrightarrow_{\lfloor x:= \rfloor} Q \equiv \lambda\, E\ h\ h'\ v\ p.\ P\ E\ h\ \longrightarrow \exists r.\ v = r \wedge Q\ E\lfloor x := r \rfloor\ h'$$

Mutual Recursion

Same approach as for mutual recursion in partial correctness logic: extend judgements to work over sets of expression and preconditions.

Definition of **goodContexT** over a context G:

$$\begin{array}{lll} \mathit{goodContexT}\ G & \equiv & \forall(\texttt{call}\ f, P) \in G.\ \forall n. \\ & & \quad (\forall(\texttt{call}\ f', P') \in G. \\ & & \quad \forall m < n.\ \{P'\ \ m\}\ \texttt{call}\ f'\ \downarrow) \longrightarrow \{P\ \ n\}\ \mathit{body}\ f\ \downarrow \end{array}$$

With this predicate we can prove the following mutual recursion lemma:

$$\frac{\mathit{finite}\ G \quad \mathit{goodContexT}\ G \quad (\texttt{call}\ f, P) \in G}{\{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ \texttt{call}\ f\ \downarrow}\ (\mathrm{MUTREC})$$

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 $\longrightarrow$ invariants in general program logics $\longrightarrow$ specific logic for interesting (resource-)properties
- Aim: exploit structure of Camelot compilation (freelist) and program analysis

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

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

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

- 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 want to need to!**

Proof System

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

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

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

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

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

Discussion of the Heap Space Logic

- ☺ LFD assertions practically useful and conceptually appealing
- ☺ 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
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.pd`
- 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 GMG05,
  `http://www-sop.inria.fr/everest/soft/Jack/doc/papers/gmg05.p`