Petra: Pragmatic & Compositional Software Verification

By Aran Hakki

Supervisors: Corina Cirstea & Julian Rathke

University of Southampton

Light System (entry point):

```java
public static void main(String... args){
    start(Light::toggleLight,Light.class);
}
```

Automatically instantiates the root object and invokes the root action on it.

Button X Power = { (off,off), (off,on), (on,off), (on,on) }

```java
public interface Button {

    PBoolean active();

    default boolean off() { return !active().get(); }
    default boolean on() {return active().get(); }

    @Edge static void buttonOff(Button b){
        kases(b,
            kase(
                button->button.off() ^ button.on(),
                button->button.off(),
                button->{
                    button.active().set(false);
                })
        );
    }

    @Edge static void buttonOn(Button b){
        kases(b,
            kase(
                button->button.off() ^ button.on(),
                button->button.on(),
                button->{
                    button.active().set(true);
                })
        );
    }
}
```

```java
public interface Power {

    PBoolean active();

    default boolean off() { return !active().get(); }
    default boolean on() {return active().get(); }

    @Edge static void powerOff(Power b){
        kases(b,
            kase(
                power->power.off() ^ power.on(),
                power->power.off(),
                power->{
                    power.active().set(false);
                })
        );
    }

    @Edge static void powerOn(Power b){
        kases(b,
            kase(
                power->power.off() ^ power.on(),
                power->power.on(),
                power->{
                    power.active().set(true);
                })
        );
    }
}
```

State Space Abstraction

Process Abstraction & Refinement

```java
public interface Light {

    Button button();
    Power power();

    default boolean off(){ return button().off() || power().off();}

    default boolean on(){return button().on() && power().on();}

    static void toggleLight(Light l){
        kases(l,
            kase(light->light.off(), light->light.on(), light->{
                join(light,
                    par(light_->light_.button(),Button::buttonOn),
                    par(light_->light_.power(),Power::powerOn));
            }),
            kase(light->light.on(), light->light.off(), light->{
                seq(light.button(),Button::buttonOff);
                seq(light.power(),Power::powerOff);
            })
        );
    }
}
```

Pre

Post

Conditions

Implementations which provably conform to Contracts

Composable Conditional Branching using kase blocks that have disjoint pre-conditions

```
Software development is hard because we do not have an efficient way
of gaining high confidence in the code we write. Testing and code reviews
are validation methods which demonstrate the presence of errors,
but not the absence of errors which is achieved by verification.
We need both verification and validation to achieve high confidence in our
systems. Petra aims to achieve both of these efficiently through a simple
to read, object-oriented (OO), modular, compositional programming language,
which has a formal semantics that supports efficient automatic checking of
strong correctness properties including reachability, termination and
determinism for both sequential and parallel programs.
Petra has been designed to be implemented as an embedded domain specific
language (EDSL) within mainstream OO programming languages
(in order to reduce the development cost of implementation and adoption)
and its formal semantics is designed to be easy to reason about
and automate in a scalable way. Petra aims to be a standard, the de facto
way to reason about and develop large and complex software systems.
```

Abstract Syntax

$obj ::= \text{object } A : \overline{\alpha}\ \{\overline{\beta}\ \overline{\gamma}\ \overline{\delta}\}$
$\overline{\alpha} ::= A_h$
$\overline{\beta} ::= B_i\ b_i$
$\overline{\gamma} ::= p_j = e_j$
$\overline{\delta} = m_k = ks_k$
$e ::= b_i.p_j \mid e \wedge e \mid e \vee e \mid e + e \mid \neg e$
$ks ::= kases(k_i)$
$k ::= kase(v, w)$
$c ::= p_j \mid e$
$v ::= c_{pre}, c_{post}$
$s ::= A < X > .m \mid A.m$
$w ::= w \,;z \mid z$
$z ::= s \mid parc$
$parc ::= parc \| s \mid s$

Concrete Syntax (Java)

$obj ::= \text{interface } A \text{ extends } \overline{\alpha}\ \{\overline{\beta}\ \overline{\gamma}\ \overline{\delta}\}$
$\overline{\alpha} ::= A_h$
$\overline{\beta} ::= B_i\ b_i;$
$\overline{\gamma} ::= default\ boolean\ p_j\{\ return\ e_j;\}$
$\overline{\delta} = static\ void\ m_k(X\ x)\{ks_k\}$
$e ::= b_i.p_j \mid e \,\&\&\, e \mid e \| e \mid e\hat{\ }e \mid !e$
$ks ::= kases(k_i);$
$k ::= kase(v, x \rightarrow w)$
$v ::= p, q$
$c ::= p_j \mid e$
$p ::= x \rightarrow c_{pre}$
$q ::= x \rightarrow c_{post}$
$w ::= w \,; z\,; \mid z\,;$
$z ::= seqc \mid join(parc)$
$seqc ::= seqc\ ;\ seq(x, s) \mid seq(x, s)$
$parc ::= parc\ ,\ par(x \rightarrow b_i, s) \mid par(x \rightarrow b_i, s)$
$s ::= A :: m$

$\xrightarrow{trans}\ \subset K \times K,\ k \in K$

$\xrightarrow{symb}\ \subset \Sigma \times \Sigma,\ \sigma \in \Sigma,\ \sigma = \langle k, \alpha, isProoved\rangle$

$\xrightarrow{conc}\ \subset X \times X,\ x \in X,\ x = \langle k, \gamma, isProoved\rangle$

Above is the structure of a rewriting semantics over Petra's concrete Java syntax. The **trans** relation automatically normalizes a Petra program into kases of only sequential, deterministic steps.
If successful the **symb** relation then attempts to automatically demonstrate reachability,
by symbolically executing these steps on abstract states. If successful the operational semantics can be understood using **conc** which is related to **symb** , however it concretely executes the same steps on the concrete state which is input to the Petra program.

Below is a denotational semantics over the abstract syntax, which provides a maths friendly description of Petra.

The concrete syntax closely mirrors the abstract syntax!!!
We expect to able to prove an isomorphism between the more human friendly denotational semantics over the abstract syntax and a more computer friendly rewriting semantics over the concrete syntax.
The rewriting semantics will provide a way of efficiently implementing the verification system and a path towards a precise operational semantics of Petra using Java (its host language).

Use of the **Denotational Semantics** to prove refinement of the 1st kase statement in Light.

Example proof: Refinement of 1st kase statement in Light.

1. Define objects in abstract syntax:

$object\ Power\{\ Bool\, b,\ \ on = true,\ off = false,\ on = kases(kase(on \vee off, on, b := true)),\ off = kases(kase(on \vee off, off, b := false))\}$
$object\ Button\{\ Bool\, b,\ \ on = true,\ off = false,\ on = kases(kase(on \vee off, on, b := true)),\ off = kases(kase(on \vee off, off, b := false))\}$
$object\ Light\{\ Power\ p,\ Button\ b,\ on = p.on \wedge b.on,\ off = p.off \vee b.off,$
$\quad toggle = kases(kase(off, on, Power.powerOn \| Button.buttonOn),\ kase(on, off, Power.powerOff; Button.buttonOff))\}$

2. Apply semantic rules:

$$\begin{gathered} Apply\ STEP\_RULE: \\ [\![kase(off, on, Pwr.on \| Btn.on)]\!] = \\ [\![Btn.on \| Pwr.on]\!]^{Lht}\restriction_R\ if\ [\![Btn.on \| Pwr.on]\!]^{Lht}\restriction_R \subseteq [\![off, on]\!]^{Lht} \\ where\ R = dom([\![off, on]\!]^{Lht}) \end{gathered} \tag{8}$$

$$\begin{gathered} Apply\ RESOLVE\_CONTRACT\_RULE: \\ [\![off, on]\!]^{Lht} = [\![off]\!]^{Lht} \rightarrow [\![on]\!]^{Lht} = \{(t,f),(f,f),(f,t)\} \rightarrow \{(t,t)\} \end{gathered} \tag{9}$$

$$\begin{gathered} Apply\ ASSUME\_STEP\_RULE: \\ [\![Btn.on]\!] = [\![on \vee off]\!]^{Bnt} \rightarrow [\![on]\!]^{Bnt} = \{t,f\} \rightarrow \{t\} = \{t \mapsto t, f \mapsto t\} \\ [\![Pwr.on]\!] = [\![on \vee off]\!]^{Pwr} \rightarrow [\![on]\!]^{Pwr} = \{t,f\} \rightarrow \{t\} = \{t \mapsto t, f \mapsto t\} \end{gathered} \tag{10}$$

$$\begin{gathered} Apply\ SEP\_SEQ\_PAR\_RULE: \\ [\![Btn.on \| Pwr.on]\!] = [\![Btn.on]\!] \times [\![Pwr.on]\!] \\ [\![Btn.on \| Pwr.on]\!] = \\ \{(t,t) \mapsto (t,t), \\ (t,f) \mapsto (t,t), \\ (f,t) \mapsto (t,t), \\ (f,f) \mapsto (t,t)\} \end{gathered} \tag{11}$$

$$\begin{gathered} [\![Btn.on \| Pwr.on]\!]^{Lht}\restriction_R \subseteq [\![off, on]\!]^{Lht} \\ where\ R = dom([\![off, on]\!]^{Lht}) = \\ \{(t,t) \mapsto (t,t), (t,f) \mapsto (t,t), (f,t) \mapsto (t,t), (f,f) \mapsto (t,t)\} \\ \subseteq \\ \{(t,f),(f,f),(f,t)\} \rightarrow \{(t,t)\} \\ = \\ \{(t,t) \mapsto (t,t), (t,f) \mapsto (t,t), (f,t) \mapsto (t,t), (f,f) \mapsto (t,t)\} \\ \subseteq \\ \{(t,f) \mapsto (t,t), (f,f) \mapsto (t,t), (f,t) \mapsto (t,t)\} \\ \Longrightarrow \\ [\![kase(off, on, Pwr.on \| Btn.on)]\!] = [\![Btn.on \| Pwr.on]\!]^{Lht}\restriction_R = \\ \{(t,f) \mapsto (t,t), (f,t) \mapsto (t,t), (f,f) \mapsto (t,t)\} \\ \blacksquare \end{gathered} \tag{12}$$

Petra Program

A petra program is a tuple $< A.m,\ A,\ \overline{obj} >$, where A.m is the root object step (the entry point), A is the root object, $\overline{obj}$ is the set of object definitions.

Correct by Construction

A Petra program is correct by construction if:

$$[\![A.m]\!] : X \rightarrow X$$

where $X = [\![A]\!]_1 \cup \{\bot\}$ *and* $[\![A.m]\!]$ *is a total function with the codomain is restricted to* $[\![A]\!]_1$

Objects

$$\begin{gathered} [\![A]\!] = (A, [\![\overline{\gamma}]\!], [\![\overline{\delta}]\!]) \\ \text{where each } P_j \subseteq A \\ \text{and each } f_k : A \rightarrow A \\ where \\ [\![\overline{\gamma}]\!] = \{p_j = P_j\} \\ [\![\overline{\delta}]\!] = \{m_k = f_k\} \end{gathered} \tag{1}$$

$$[\![object\ A\{\ B_i\, b_i,\ p_j = e_j,\ m_k = ks_k\}]\!] = ([\![B_1]\!]_1 \times [\![B_2]\!]_1 .... \times [\![B_I]\!]_1, \{p_j = [\![e_j]\!]^A\}, \{m = [\![ks]\!]^A\}) \quad \text{(COMPUTE\_OBJECT\_RULE)}$$

Each primitive type has a corresponding primitive object e.g.

$$[\![Bool]\!] = (B, \{isTrue = \{t\}, isFalse = \{f\}\}, \{setTrue = \{t,f\} \rightarrow \{t\}\},\ setFalse = \{t,f\} \rightarrow \{f\}\}) \quad \text{(ASSUME\_PRIMATIVE\_RULE)}$$

The assume primitive rule allows the semantics of primitive objects to be assumed primitive. Note objects are either primitive or non-primitive.

Predicates

$$[\![b_i.p_j]\!]^A = [\![B_1]\!]_1 \times [\![B_2]\!]_1 \times .... \times ([\![e_j]\!]^A \cap [\![B_i]\!]_1) \times [\![B_{i+1}]\!]_1 \times ... \times [\![B_I]\!]_1$$

$$where\ [\![b_i.p_1]\!]^A \cap [\![b_i.p_2]\!]^A ... \cap [\![b_i.p_J]\!]^A = \emptyset \quad \text{(PREDICATE\_RULE)}$$

$$[\![p_j]\!]^A = [\![e_j]\!]^A \quad \text{(RESOLVE\_PREDICATE\_RULE)}$$

$$[\![e \wedge e']\!]^A = [\![e]\!]^A \cap [\![e']\!]^A \quad \text{(AND\_RULE)}$$

$$[\![e \vee e']\!]^A = [\![e]\!]^A \cup [\![e']\!]^A \quad \text{(OR\_RULE)}$$

$$[\![e + e']\!]^A = [\![e]\!]^A \cup [\![e']\!]^A \setminus [\![e]\!]^A \cap [\![e']\!]^A \quad \text{(XOR\_RULE)}$$

$$[\![\neg e]\!]^A = [\![A]\!]_1 \setminus [\![e]\!]^A \quad \text{(NOT\_RULE)}$$

Steps

$$[\![A.m]\!] = [\![ks]\!]^A \quad \text{(RESOLVE\_KASES)}$$

The resolve kases rule allows a step label i.e. a kases statement label, to be equated to the total function constructed by the interpretation of the kases statement.

$$[\![A.m\ ;\ A.m']\!]^A = [\![A.m]\!] \circ [\![A.m']\!] \quad \text{(SEQ\_RULE)}$$

$$[\![B_x.m\ ;\ B_{x+c}.m']\!]^A = [\![B_x.m \ \|\ B_{x+c}.m']\!]^A = z$$

where

$$\begin{gathered} z = id_1 \times id_2 ... id_{x-2} \times id_{x-1} \times [\![B_x.m]\!] \times \\ id_{x+1} \times id_{x+2} ... id_{x+c-1} \times [\![B_{x+c}.m']\!] \times \\ id_{x+c+1} \times id_{x+c+2} ... id_{I-2} \times id_{I-1} \times id_I, \\ dom([\![B_x.m]\!]^A) \cap dom([\![B_{x+c}.m']\!]^A) = \emptyset\ , \\ [\![B_x]\!]_1 * [\![B_{x+c}]\!]_1, \\ s, h \models P_1 * P_2 \text{ where there exist } h_1, h_2 \\ \text{such that } h_1 \perp h_2 \text{ and } h = h_1 \cup h_2 \text{ and } s, h_1 \models P_1 \\ \text{and } s, h_2 \models P_2, \end{gathered}$$

where $*$ is a separating conjunction which asserts that the heap can be split into two disjoint parts where its two arguments hold. (SEP_SEQ_PAR_RULE)

Separated sequential and parallel composition is achieved through take the product of a sequence of functions. Where the sequence contains a function for each field in the object in which the composition is taking place. For each field the function is either an identity function over the set of observable states in the field or the interpretation of a named step which acts on the field. For this to work the domains of the functions within the product need to be disjoint. Implementations will use separate memory regions to ensure that the observable states will be disjoint and the parallel composed steps can only mutate separate regions hence they cannot interfere with one another. The function product (which can be understood as a generalized version of equation 2, for any number of functions) and there is an identity function id_i for every field not used by any of the steps in the composition. This provides composition and lifting of transforms on separate state spaces, into the state space of the parent object A.

$$\begin{gathered} f : X \rightarrow Y,\ g : U \rightarrow V \\ f \times g : X \times U \rightarrow Y \times V \\ (f \times g)(x, u) = (f(x), g(u)) \end{gathered} \tag{2}$$

Kases

$$\begin{gathered} [\![ks]\!]^A : [\![A]\!]_1 \cup \{\bot\} \rightarrow [\![A]\!]_1 \cup \{\bot\} \\ [\![ks]\!]^A = [\![k_1]\!]^A \cup [\![k_2]\!]^A ... [\![k_I]\!]^A\ if \\ dom([\![k_1]\!]^A) \cap dom([\![k_2]\!]^A) ... dom([\![k_I]\!]^A) = \emptyset\ and \\ [\![k_1]\!]^A,\ [\![k_2]\!]^A, ...\ [\![k_I]\!]^A ... \neq error \\ else\ [\![ks]\!]^A = error,\ error = \{\bot \mapsto \bot\} \end{gathered}$$

(COMBINE_KASES_RULE)

$$\begin{gathered} [\![k]\!]^A : [\![A]\!]_1 \cup \{\bot\} \rightarrow [\![A]\!]_1 \cup \{\bot\} \\ [\![k]\!]^A = [\![kase(v, w)]\!]^A \end{gathered} \tag{3}$$

The function signature and definition above provides an interpretation of kase statements.

$$[\![kase(v, w)]\!]^A = [\![v]\!]^A\ if \text{ w is a primitive step } \wedge\ [\![v]\!]^A\ is\ total\ function \quad \text{(ASSUME\_STEP\_RULE)}$$

The assume step rule interprets a primitive kase as the interpretation of its contract, where the contract must be equivalent to a total function.

$$\begin{gathered} [\![kase(v, w)]\!]^A = [\![w]\!]^A\restriction_R\ if \text{ w is a non-primitive step } \wedge \\ [\![w]\!]^A\restriction_R \subseteq [\![v]\!]^A \\ [\![kase(v, w)]\!]^A = error\ if \text{ w is a non-primitive step } \wedge \\ [\![w]\!]^A\restriction_R \not\subseteq [\![v]\!]^A \\ where\ R = dom([\![v]\!]^A) \end{gathered}$$

(STEP_RULE)

The step rule interprets non-primitive kase as the function which results from the interpretation of the kases step composition (w), which has been restricted by the domain of the kases contract, only if the interpreted function fits within the relation provided by the kase contract. Otherwise error returned which is a total function which maps bottom to bottom.

$$\begin{gathered} [\![v]\!]^A = [\![c_p, c_q]\!]^A = [\![c_p]\!]^A \rightarrow [\![c_q]\!]^A \\ [\![c_p]\!]^A = \text{set of observable input states given by the pre-condition} \\ [\![c_q]\!]^A = \text{set of observable output states given by the post-condition} \\ \text{(note v is the kases contract)} \end{gathered}$$

(RESOLVE_CONTRACT_RULE)

The resolve contract rule takes the pre/post condition pair and interprets it as a relation between the sets of observable states captured by the pre and post conditions according to the PREDICATE_RULE.

Aran Hakki
07399472347
ah1m20@soton.ac.uk