Towards Optimal Instrumentation for CFTL

1 - Program Transformation - Loops

The first in a series of talks on theory for instrumentation

Joshua Dawes

Contents of this talk

Success of VyPR so far

What is instrumentation, and why do it well?

When is instrumentation optimal?

(Partial) Loop unrolling

Which loops do we unroll?

Experiments with VyPR so far

Found 2 performance bottlenecks in CMS' Conditions Uploader.

1 is easily fixable (hash checking for payload upload reduction), the other is not so easy (frequent insertions to Oracle leads to blow up in latency, for some reason...).

A decent start, and a good first step in developing more sophisticated software analysis tools.

VyPR is very easy to apply to a Python (Flask-based, for the moment) web service.

Instrumentation and monitoring is automatic.

`http://cern.ch/vypr/`

What is instrumentation, and why do it well?

To check if a program respects some state and time constraints, we need a way to extract during runtime the data talked about by the constraints.

Basic approach: add instructions to the code that will send the relevant data to another thread for monitoring.

Possible instrumentation: take everything. But this will induce massive (unacceptable) overhead.

Better instrumentation: take only what we need. Minimal overhead. Much harder to do this.

Key problem: how do we know that we need an observation?

What is instrumentation, and why do it well? continued

For the constraints we check, we need to know what an observation will look like that we can use to check the constraints.

CFTL cares about state and time constraints, so observations are values from variables and durations of events (functions calls, etc).

If a logic is high-level (like most existing ones), instrumenting for it requires a mapping to say how the low-level events at runtime relate to the high level specification.

always($a \implies b$) - straightforward meaning, but mapping a and b to events at runtime is tricky.

CFTL is low level, so no mapping is required.

$\forall t \in \text{calls}(f) : \text{duration}(t) < 1$ - meaning is clear without any mapping.

When is instrumentation optimal?

Simply:

When we observe everything that we need, and never anything that we don't need.

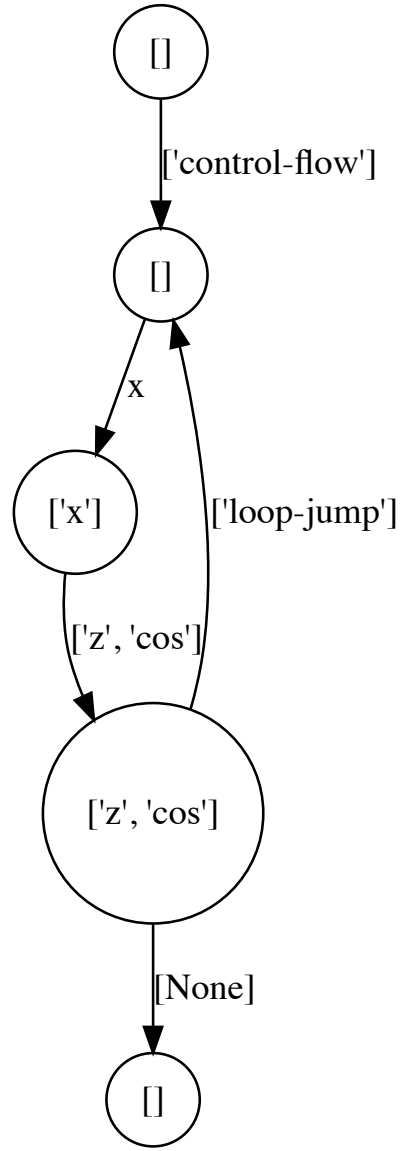
VyPR uses a static model of programs called a Symbolic Control-Flow Graph (SCFG) to perform instrumentation.

This is a directed graph with vertices V and edges E. Vertices represent values changing or functions being called, and edges represent instructions causing the state to change.

VyPR's instrumentation algorithm, atom-driven instrumentation, for a CFTL property φ gives a set $\mathsf{Inst}(\varphi) \subset V \cup E$.

This is the set of points to which we will add instruments. Sometimes it's optimal, sometimes not.

```
for i in range(0, 10):
  x = i
  z = cos(x)
```



An example of optimality

```
for i in range(0, 10):
  x = i
  z = cos(x)
```

$$\forall q \in \text{changes}(x) : \\ q(x) \in [0, \pi/2] \\ \implies \text{next}(q, \text{changes}(z))(z) \geq 0$$

In this case, there are no wasted observations, and we gather all the information we need.

```
for i in range(0, 10):
  x = i # instrumented
  z = cos(x) # instrumented
```

An example of sub-optimality

```
x = 0
for i in range(0, 10):
  z = cos(x)
```

$$\forall q \in \text{changes}(x):$$
$$q(x) \in [0, \pi/2]$$
$$\implies \text{next}(q, \text{changes}(z))(z) \geq 0$$

In this case, we observe 11 messages from instruments (1 for `x = 0` and 10 for `z = cos(x)`). 9 are wasted because the property only asks for the *next* state change. This is far below optimal.

```
x = 0# instrumented
for i in range(0, 10):
  z = cos(x) # instrumented
```

Loop Unrolling

Normally used by compilers to optimise loops.

We can do it here in the case that we only need the first observation from a loop.

```
x = 0 # instrumented
z = cos(x) # inst...
for i in range(1, 10):
  z = cos(x)
```

$$\forall q \in \text{changes}(x):$$
$$q(x) \in [0, \pi/2]$$
$$\implies \text{next}(q, \text{changes}(z))(z) \geq 0$$

If we partially unroll the loop before the program is instrumented, atom-driven instrumentation yields an optimal set $\text{Inst}(\varphi)$.

Note: more recent work raises the suspicion that there may be cases where even loop unrolling fails to reach optimality.

Which loops do we unroll?

Loop unrolling worked for the property

$$\begin{aligned} &\forall q \in \text{changes}(x) : \\ &\quad q(x) \in [0, \pi/2] \\ &\quad\quad \implies \text{next}(q, \text{changes}(z))(z) \geq 0 \end{aligned}$$

To fully explain why, and to determine when it will work in general, we will look at the neighbourhoods of instrumentation points in which there can be wasted observations that loop unrolling could prevent.

The Dynamic Context: Concrete Bindings

Consider the following toy property

$$\begin{aligned} &\forall c \in \mathsf{calls}(g) : \\ &\quad \mathsf{duration}(c) < 1 \\ &\quad\quad \implies \mathsf{next}(c, \mathsf{changes}(x))(x) < 10 \end{aligned}$$

"Everytime g is called and takes less than a second, the next time x is changed, the new value should be less than 10".

Semantically, we consider $\forall c \in \mathsf{calls}(g)$ as generating a space of maps (concrete bindings). Each one has the form $[c \mapsto v]$ where v is a transition representing a call to the function g.

So, for each binding $[c \mapsto v]$, in $(\mathsf{duration}(c) < 1)$, c takes the value v.

The Static Context: Symbolic Support

Fix a $c \in \text{calls}(g)$. To resolve $\psi \equiv (\text{duration}(c) < 1 \implies \text{next}(c, \text{changes}(x))(x) < 10)$, a set consisting of a transition c and a state $\text{next}(c, \text{changes}(x))$ is required.

In this set, transitions correspond to edges in the SCFG and states correspond to vertices. This relationship is known as symbolic support. We denote by $s(v)$ the edge or vertex in the SCFG to which a state or transition v corresponds. We say $s(v)$ is the symbolic support of v.

Concrete bindings therefore also have symbolic supports, which we call static bindings.

A static binding is normally denoted by $s([c \mapsto v])$ and is the map $[c \mapsto s(v)]$.

An Intuitive Meaning

Static bindings are the lines of code that could generate verdicts - a formula $\forall s \in \text{changes}(x)$... applied over n changes of x generates n verdicts.

```
for i in range(0, 10):
  x = i
  z = cos(x)
```

$$\varphi \equiv \begin{array}{l} \forall s \in \text{changes}(x) : \\ \qquad \text{dest}(\text{next}(q, \text{calls}(\cos)))(z) \in [-1, 1] \end{array}$$

For φ, there is 1 static binding, the state generated by `x = i` (restricting to primitive types), that gives 10 concrete bindings at runtime and so 10 verdicts.

Binding Spaces and Neighbourhoods

The collection of all of the static bindings is called the *binding space* for the property φ and is denoted by $\mathcal{B}(\varphi)$. We refer to individual static bindings as $\beta \in \mathcal{B}(\varphi)$.

Fix φ and $\beta \in \mathcal{B}(\varphi)$. Denote by $\mathsf{Inst}(\varphi, \beta)$ the subset of the SCFG required to be sure that we observe enough data to check φ at a concrete binding whose symbolic support is β.

$\mathsf{Inst}(\varphi, \beta)$ gives us a way to focus on the local instrumentation neighbourhood of a point.

To decide which loops to unroll, we will characterise the neighbourhoods $\mathsf{Inst}(\varphi, \beta)$ in which there are wasted observations.

Loops are Cycles

A point $p \in \text{Inst}(\varphi)$ is in a loop if it is in a cycle in the SCFG. It is in a cycle if and only if $p \in \text{reaches}(p)$, where $p' \in \text{reaches}(p) \iff \exists$ a (non-empty) path π from p to p'.

This is the first part of the characterisation - any $\text{Inst}(\varphi, \beta)$ containing a point p that is in a loop. Given that we're looking at unrolling loops, this is expected...

What's the Difference?

$$\forall q \in \text{changes}(x):$$
$$q(x) \in [0, \pi/2]$$
$$\implies \text{next}(q, \text{changes}(z))(z) \geq 0$$

```
for i in range(0, 10):
  x = i
  z = cos(x)
```

No observations are wasted.

```
x = 0
for i in range(0, 10):
  z = cos(x)
```

9 out of 11 observations are wasted.

A Path through Inst(φ, β)

Fix some $p \in \text{Inst}(\varphi)$ such that p is in a cycle. Denote by α_p the atom in φ used to find p during atom-driven instrumentation. Then, $\text{var}(\alpha_p)$ is the (quantified) variable on which α_p is based.

For example, $\text{var}(\text{duration}(\text{next}(q, \text{calls}(f))) < 1) = q$.

With $q = \text{var}(\alpha_p)$, consider $\beta(q) \in V \cup E$.

Given $\beta(q)$, for all $p \in \text{Inst}(\varphi, \beta) \backslash \{\beta(q)\}$, there is a path π_p from $\beta(q)$ to p. This is guaranteed because all other p were found by reachability analysis during atom-driven instrumentation.

A Complete Characterisation

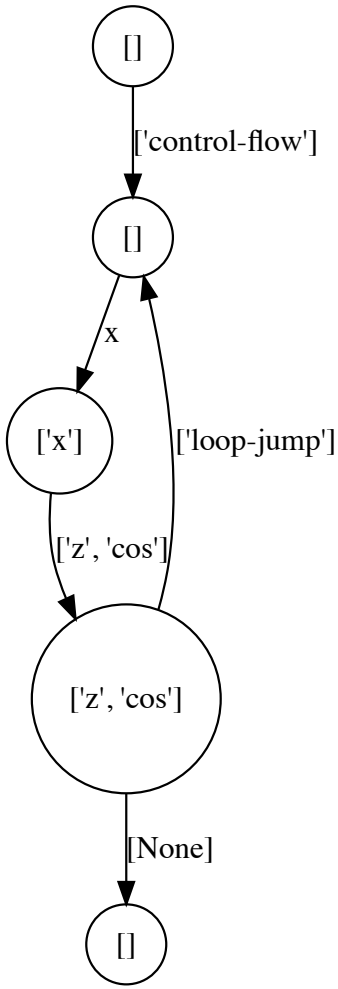
Whenever this $\beta(q)$ is traversed at runtime, this triggers the start of evaluation of a new formula tree. If the same $\beta(q)$ is traversed multiple times, there will be multiple formula trees and they will all be updated with each observation.

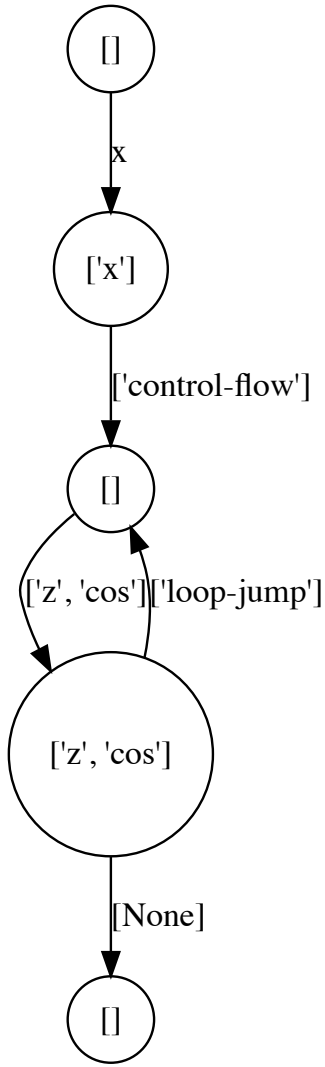
A point $p \in \mathsf{Inst}(\varphi, \beta) \backslash \{\beta(q)\}$ can generate wasted observations if it can be traversed multiple times without traversing $\beta(q)$ inbetween.

This is the second and final part of the characterisation of neighbourhoods that can generate wasted observations.

```
for i in range(0, 10):
  x = i
  z = cos(x)
```

```
x = 0
for i in range(0, 10):
  z = cos(x)
```





$$\forall q \in \text{changes}(x):$$
$$q(x) \in [0, \pi/2]$$
$$\Longrightarrow$$
$$\text{next}(q, \text{changes}(z))(z) \geq 0$$

Example

```
x = 0
for i in range(0, 10):
  z = cos(x)
```

$$\forall q \in \text{changes}(x) : \text{next}(q, \text{changes}(z))(z) \geq 0$$

$$\mathcal{B}(\varphi) = \{[q \mapsto (\texttt{x = 0})]\}$$

$$\text{Inst}(\varphi, [q \mapsto (\texttt{x = 0})]) = \{\texttt{z = cos(x)}\}$$

The observation of `z = cos(x)` is wasted when the loop iterates, generating more of the same observation. Notice that, when the loop goes back around, $\beta(q) = (\texttt{x = 0})$ is not traversed again. Hence, the observation is wasted.

Example with modified code

```
for i in range(0, 10):
  x = i
  z = cos(x)
```

$$\forall q \in \text{changes}(x) : \text{next}(q, \text{changes}(z))(z) \geq 0$$

$\mathcal{B}(\varphi) = \{[q \mapsto (\texttt{x = i})]\}$

$\text{Inst}(\varphi, (\texttt{x = i})) = \{\texttt{z = cos(x)}\}$

The observation of `z = cos(x)` is not wasted when the loop iterates since, when the loop goes back around, $\beta(q) = (\texttt{x = i})$ is traversed again and a new formula tree is instantiated.

A Useful Result

Lemma

Some $p \in \mathsf{Inst}(\varphi, \beta)$ can generated a wasted observation if and only if, on a path from p back to p (excluding the empty path), $\beta(\mathsf{var}(\alpha_p))$ is not encountered.

Theorem

If a point $p \in \mathsf{Inst}(\varphi, \beta)$ is in a cycle in the SCFG, the loop represented by the cycle should be unrolled if and only if p can generate a wasted observation.

I now introduce Graph Isomorphism before giving an algorithm for loop unrolling.

Graph Isomorphism

When a loop is unrolled, the SCFG of the program in which the loop was unrolled is modified to include a copy of the loop body before the loop.

The unrolled loop and the original loop body are isomorphic - they share structure.

Let G and G' be directed graphs.

Definition

$G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ with $E \subset V \times V$ and $E' \subset V' \times V'$ are isomorphic, written $G \cong G'$, if and only if there are bijections $b_{\text{vert}} : V \to V'$ and $v_{\text{edge}} : E \to E'$ such that
$\langle v_1, v_2 \rangle \in E \iff b_{\text{edge}}(\langle v_1, v_2 \rangle) = \langle b_{\text{vert}}(v_1), b_{\text{vert}}(v_2) \rangle \in E'$.

Matching

If two graphs are isomorphic, we can pick a point $p \in V \cup E$ and say that it matches a point $p' \in V' \cup E'$.

p' is the match of p.

I omit the precise details of how to construct this relation, but matching depends on isomorphism and, for SCFGs, it is automatic.

An Algorithm for Loop Unrolling

- Compute $\mathsf{Inst}(\varphi)$ using atom-driven instrumentation.
- For each $\beta \in \mathcal{B}(\varphi)$ [iteration through neighbourhoods, since β gives $\mathsf{Inst}(\varphi, \beta)$]
  - For each $p \in \mathsf{Inst}(\varphi, \beta)$:
    - If p is in a cycle and can generate a wasted observation, unroll the loop and replace $p \in \mathsf{Inst}(\varphi, \beta)$ by the match of p in the newly inserted code from loop unrolling.

This approach is too simple when the body of the loop becomes more complex; more work needs to be done to efficiently deal with this.

After developing this first theory on loop unrolling, I suspect that reaching optimal instrumentation for arbitrary programs may be impossible.

Thank you - any questions?