Incremental Solving with Vampire

Giles Reger[1] Martin Suda[2]

(1) School of Computer Science, University of Manchester, UK

(2) Institute for Information Systems, Vienna University of Technology, Austria

The 4th Vampire Workshop

Introduction

This talk will be about

- **What** we mean by incremental solving
- **Why** we want to be incremental
- **How** we can achieve this
- **When** we we will i.e. what have we actually done so far

What is Incremental Solving

- Solving a problem in increments

What is Incremental Solving

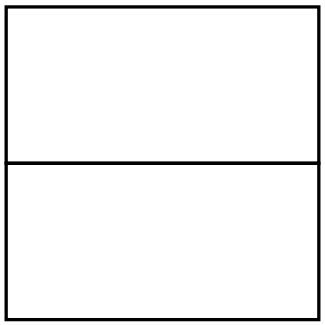
- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency

What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency

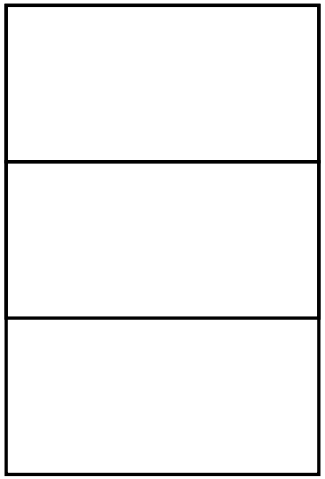
What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency



What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency



What is Incremental Solving

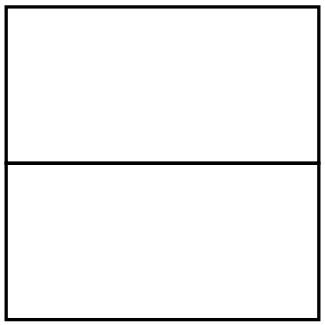
- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts

What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts

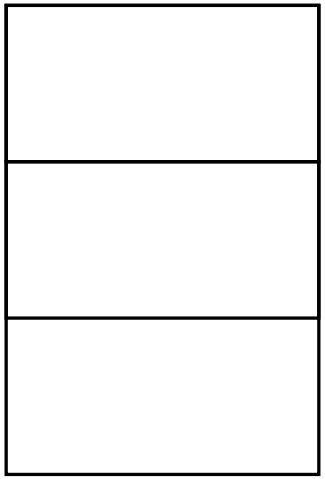
What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts



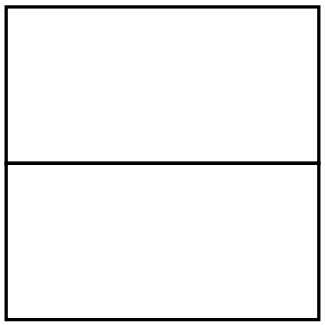
What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts



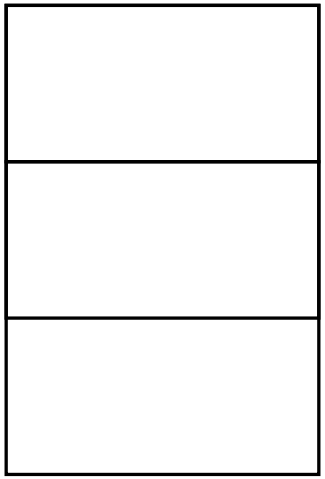
What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts



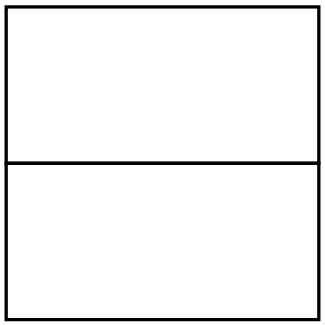
What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts



What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts



What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts

What is Incremental Solving

- Solving a problem in increments
- Two flavours
  - A growing problem - add new assertions and check consistency
  - A stack of assertions - push and pop solving contexts

- The idea of both is that there is some previous context and you add some new assertions and try and solve the resulting problem
- The second clearly subsumes the first and is more general, but the first is 'easier' as it does not require backtracking

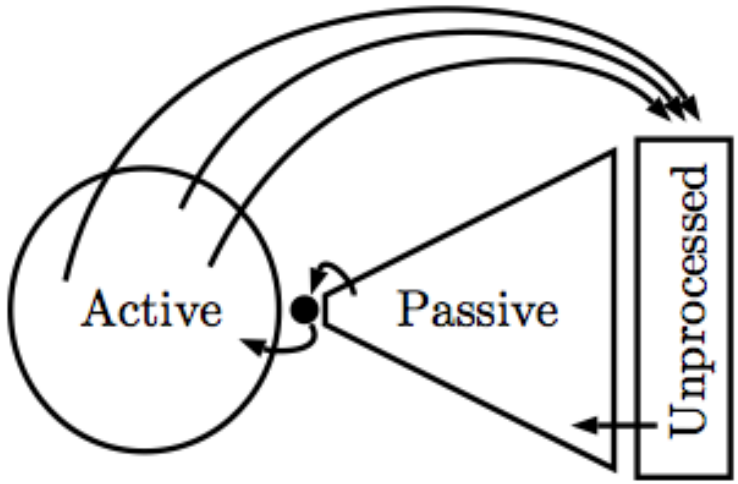
Why do we want it?

- Very useful in applications (such as program analysis) where there is some general encoding and different queries are made of this
- The cool kids are doing it (SMT)

Why is it Hard?

- SMT solves are typically model-based i.e. they attempt to build a model. Therefore, incrementally adding new information means attempting to extend that model.
- Vampire is saturation-based and adding new information means continuing saturation with this new information.
- However
  - Finite saturations may not exist
  - Finding a model means finding one satisfiable branch whereas saturating means exploring all possibilities i.e. it is harder for us
  - In many cases saturation does not mean satisfiable (e.g. theories and incomplete preprocessing)

Dealing with a Growing Problem



1. Receive formulas
2. Clausify and add to Unprocessed
3. If saturated report and goto 1
4. Else stop with unsat

Completeness

To be useful we probably want to be complete

Don't throw things away

- Avoid preprocessing steps such as
  - pure literal removal
  - function definition elimination
  - set of support or SiNE selection
- Avoid limited resource strategy or weight limits

Preserve completeness criteria

- Use all necessary inference rules
- Only use complete versions of selection
- If we have theories (interpreted symbols) it's game over

Assumptions about the Signature

We need to be careful as Vampire makes decisions based on the signature But we do not know the full signature when we start solving.

Some things we need to take care of:

- Preprocessing may add symbols to the signature!
- Inference rules are selected based on what is needed - need to add everything as we do not know
- Term ordering relies on a symbol precedence but new symbols can appear, should only be suboptimal instead of wrong
- Indexing data structures cannot be specialised (e.g. for EPR)
- Discrimination indexing trees index directly on the signature! Need to modify these to expand as needed
- Theory symbols treated specially, may need to decide from the start whether they are going to appear

Assumptions about the Signature

We need to be careful as Vampire makes decisions based on the signature But we do not know the full signature when we start solving.

Some things we need to take care of:

- Preprocessing may add symbols to the signature!
- Inference rules are selected based on what is needed - need to add everything as we do not know
- Term ordering relies on a symbol precedence but new symbols can appear, should only be suboptimal instead of wrong
- Indexing data structures cannot be specialised (e.g. for EPR)
- Discrimination indexing trees index directly on the signature! Need to modify these to expand as needed
- Theory symbols treated specially, may need to decide from the start whether they are going to appear

New Problem: Changing Conjecture

- Goal-directed proof search with a changing goal!
- Vampire might give extra weight to goal clauses and their children
- Do we adjust these weights when the goal changes?
- Not a big deal but something to think about

Tracking Solving Contexts

We have two approaches for dealing with a stack of solving contexts

1. Fork a new process for each push
   - This is what we do in competition modes for each strategy
   - The idea can allow us to try multiple proof attempts on the same solving context, this could be very important
   - But lose everything when we pop
   - But this means we are also allowed to be incomplete, throw away things from earlier solving contexts etc

2. Use labelled clauses to track stack information
   - Thought: Work in a different solving context can help
   - To preserve this we can label clauses with the most specific solving context they are relevant to
   - Also allows us to be a bit more clever... see later

The Forking Approach

- Theoretically simple
- A few issues with concurrecy
- Probably very helpful practically
  - view each conjecture as a new problem with some pre-saturation
- But mostly just engineering

- Not implemented yet

The Labelled Clause Approach

- Clauses become labelled
  - $L \rightarrow C$ where C is a clause and L a conjunction of labels
- Solving contexts are labelled
- Clauses are labelled by their solving context
- Solving is under the assumption that the active labels hold
- Inferences must preserve labels
- Popping asserts that the current label as false
- Reductions may need to be backtracked if they no longer hold

We already have a system for dealing with labelled clauses: AVATAR

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$0 \rightarrow p(X)$

$0 \rightarrow \neg p(Y) \vee q(Y)$

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$0 \rightarrow p(X)$

$0 \rightarrow \neg p(Y) \vee q(Y)$

$1 \rightarrow \neg q(a)$

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$0 \rightarrow p(X)$
$0 \rightarrow \neg p(Y) \vee q(Y)$
$1 \rightarrow \neg q(a)$
$0 \rightarrow q(Y)$
$0 \wedge 1 \rightarrow \neg p(a)$
$0 \wedge 1 \rightarrow \bot$

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$$0 \rightarrow p(X)$$
$$0 \rightarrow \neg p(Y) \vee q(Y)$$
$$\cancel{1 \rightarrow \neg q(a)}$$
$$0 \rightarrow q(Y)$$
$$\cancel{0 \wedge 1 \rightarrow \neg p(a)}$$
$$\cancel{0 \wedge 1 \rightarrow \bot}$$

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$0 \rightarrow p(X)$

$0 \rightarrow \neg p(Y) \vee q(Y)$

$0 \rightarrow q(Y)$

$0 \rightarrow \neg q(sk)$

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$0 \rightarrow p(X)$

$0 \rightarrow \neg p(Y) \vee q(Y)$

$0 \rightarrow q(Y)$

$0 \rightarrow \neg q(sk)$

$0 \rightarrow \bot$

Example

```
fof(p,axiom,![X] : p(X)).
fof(q,axiom,![Y] : ~p(Y) | q(Y)).
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

$0 \rightarrow p(X)$

$0 \rightarrow \neg p(Y) \vee q(Y)$

$0 \rightarrow q(Y)$

$0 \rightarrow \neg q(sk)$

$0 \rightarrow \bot$

We used the generation of $q(Y)$ from the inner solving context when finding the refutation later i.e. we reused some of this proof search.

Removing Clauses

- Clauses added in a solving context that is then popped can be safely removed along with any children.
- Is it worth explicitly removing such clauses?
- Clauses may be re-added in a future solving context. In which case we could detect this and, instead of adding a new clause, reactive the old clause along with relevant children. This is an idea we have explored within the context of AVATAR.
- If we really want to do this then do we make it explicit in the input

Incremental or Mutually Exclusive?

Consider a problem

```
tff(all_pos,axiom,
  $greater(a,0) &
  $greater(b,0) &
  $greater(c,0)).
tff(fermat,conjecture,
  $sum($product(a,$product(a,a)), $product(b,$product(b,b)))
  != $product(c,$product(c,c))).
tff(abc,conjecture,$greatereq(a,b) & $greatereq(a,c)).
```

The two conjectures are mutually exclusive, we could tackle them in either order. One is easier than the other.

Incremental or Mutually Exclusive?

Consider another problem

```
fof(a,axiom, a=b & b=c & c=d).
push().
fof(b,conjecture,a!=c).
pop().
push().
fof(c,conjecture,a!=d).
pop().
```

The two solving contexts are mutually exclusive and could be tackled in either order, or at the same time.

Multiple Conjectures with Labelled Clauses

- Just add the various labels and press Go
- Don't halt on an empty clause, just report the label
- Caveat: interaction with AVATAR not completely straightforward
- Efficiency: probably want to avoid unhelpful inferences (those that combine conjectures and their children)
- Now lots of things we could play with:
  - Attempt all mutually exclusive conjectures at once
  - Group them in chunks
  - Give each conjecture a bit of time but never give up
- Caveat: could be messy in general if relationship between different conjectures (in terms of exclusiveness, signature etc) is non-trivial

What is Implemented?

- New `--mode incremental` which currently accepts SMT-LIB with
  - Multiple `(check-sat)` commands
  - Matching `(push 1)` and `(push 2)` commands

  But currently requires full signature exists before first `(check-sat)`

- On `(check-sat)` we send problem so far to Vampire
  - Don't restrict completeness but track it

- Push/Pop handled by labelled clauses
  - Initial hack extends clauses with propositional symbol and registers this with AVATAR

What is Left and Other Thoughts

- Relaxing signature issues
- Forking push/pop approach
- Experiments and finding more benchmarks
- Solving under assumptions
- We want an API
- Playing with new set-of-support ideas for throttling