Andrew J. Cook
May 14, 2004

Translation Stuff

1 Introduction

The SPARK system produces verification conditions that, if shown to be true, will ensure that a program will never raise an exception at run time. Most of these conditions are trivial and can be easily removed by the SPADE simplifier however, some are more difficult and require the addition of loop invariants and the use of more complex proof systems.

The NuSPADE/CLAMSPADE project uses program analysis and the CLAM proof planner to help prove these conjectures. However, the tactics that CLAM produces need to be translated to a form that the SPADE checker can understand in order to show that the proofs are sound. The methods used in the proof plan can be used to produce a command script for the SPADE checker so that the proof plan can be run through the checker automatically but due to differences in the way CLAM and SPADE handle proofs the translation isn't always straight forward.

2 Plan structure

To properly generate the SPADE commands the translator needs some information about the goal and the hypotheses it is to be proven from.

Within each verification condition there may be more than one conclusion to be proven these need to be proven consecutively. This allows the translator to keep track of the hypotheses that it introduces in one proof and uses that to adjust the next proof script accordingly.

The top level structure of a group of plans to be translated looks like this:

$$\texttt{plans(}VC\texttt{, }H\#\texttt{, [plan(}Conc_1\texttt{, }Plan_1\texttt{), ..., plan(}Conc_n\texttt{, }Plan_n\texttt{)])}$$

2.0.1 Top level information

- VC: The number of the verification condition to be proven.
- $H\#$: The number hypotheses in the hypothesis list.

The verification condition is needed to allow the translator to tell SPADE which of the list of verification conditions to prove. The number of hypotheses is necessary to allow the translator to target a hypothesis that it has introduced in during the course of the proof

2.0.2 Next level information

- *Conc*: The number of the conclusion to be proven within the particular verification condition.
- *Plan*: The tactic generated within CLAM that is to be translated into the SPADE proof.

The lable of a conclusion within a verification condition does not change as each conclusion is discharged. This means that the translator needs to know the lable of the conclusion to be proven at the start so that it can target it.

3 Wrapping and Chaining proofs

Information about an series of proofs to be translated is given at the top level and carried around by the translator. It can also use this information to perform some preliminary operations to set up the checker for the commands in the rest of the translation.

The number of the verification condition(VC) can be used to automatically switch to the correct verification condition inside SPADE.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

VC.

Or if being executed from within spade:

```
newvc.
yes.
```
VC.

If the name of the subprogram(*subprogram*) that is being checked is included it can be used to automatically load the generated rules for it.

`consult('`*subprogram*`.rls').`

For a number conclusions in the same goal the proof commands need to be chained together so that the translator can keep track of the number of hypotheses introduced in each one. It can then adjust the numbering of commands that target hypotheses to take into account the increasing length of the hypothesis list.

So we have:

Tactic	Command
`plans([`*subprogram*`],` *VC*`,` *H#*`, [plan(`$Conc_1$`,` $Plan_1$`), ..., plan(`$Conc_n$`,` $Plan_n$`)])`	`newvc.` `yes.` *VC*. `consult('`*subprogram*`.rls').` *commands for* $Plan_1$ ... *commands for* $Plan_n$

3.1 Problems

We may have several custom rules that we need to properly perform the plan in SPADE, the loading of these rules cannot be included in the translated commands because the information about where they might be found isn't available.

4 Equality Rewrites

CLAM uses rewrites based on equalities both for wave rules and for simplification steps. In SPADE the use of equality rewrites correspond quite closely with how they are used in CLAM so the translation is relatively simple.

The equality rewrite tactics CLAM produces look like this:

`wave(`*Rule*`,` *Pos*`, equ,` *Before*`,` *After*`)`
`simplify(`*Rule*`,` *Pos*`, equ,` *Before*`,` *After*`)`

- *Rule*: The rule name.
- *Pos*: The term position of the subterm to be rewritten within the goal.
- `equ`: A flag indicating that the rewrite to be used is based on an equality.
- *Before*: The goal before the rewrite.
- *After*: The goal after the rewrite.

To use an equality rewrite within SPADE is quite easy using the `replace` command. We need to specify the structure of the subterm to be rewritten before and after rewriting(*subterm(Before)*, *subterm(After)*) so the translator calculates it using the subterm position(*Pos*) given in the CLAM tactic.

`replace c#`*n*`:` *subterm(Before)* `by` *subterm(After)* `using` *Rule*`.`

n is the lable of the conclusion to be rewritten.

After the rewrite step SPADE asks if the replacement is desired and if any more replacement is to be done. We answer `yes` to complete the rewrite and `no` to make the checker drop back to the normal command prompt for the next step.

So we have:

Tactic	Command
`wave(`*Rule*`,` *Pos*`, equ,` *Before*`,` *After*`)` `simplify(`*Rule*`,` *Pos*`, equ,` *Before*`,` *After*`)`	`replace c#`*n*`:` *subterm(Before)* `by` *subterm(After)* `using` *Rule*`.` `yes.` `no.`

4.1 Problems

A possible problem arises when we have two occurrences of the subterm within the goal. Because SPADE asks the user to identify which occurrence is to be replaced the translator will not only need to know the occurrence that is to be translated but anticipate when this will happen otherwise the the command file will become out of sync with the checker. In the examples we have looked at so far this problem has not arisen but in an arbitrary example it remains a possibility.

5 Implication rewriting

As well as using equality for rewriting CLAM also allows implication rules to be used for rewriting but only in one direction. The CLAM plan also uses two special purpose methods `trans` and `decomp` which are based on an implication rewriting step. Because CLAM uses a goal based approach while SPADE doesn't this causes problems for the translation. Though the translation is more round about we can use the same format of tactic used for an equality rewrite to perform the translation.

`wave(`*Rule*`,` *Pos*`, imp,` *Before*`,` *After*`)`
`simplify(`*Rule*`,` *Pos*`, imp,` *Before*`,` *After*`)`
`trans(`*Rule*`,` *Pos*`, imp,` *Before*`,` *After*`)`
`decomp(`*Rule*`,` *Pos*`, imp,` *Before*`,` *After*`)`

- *Rule*: The rule name.
- *Pos*: The term position of the subterm to be rewritten within the goal.
- `imp`: A flag indicating that the rewrite to be used is based on an implication.
- *Before*: The goal before the rewrite.
- *After*: The goal after the rewrite.

In SPADE the `infer` command can be used to apply these rules to a hypothesis to create a new rewritten hypothesis but they cannot be used directly on the conclusion. This means that the step corresponding to rewriting the goal in CLAM cannot be performed directly in SPADE, instead we use the `infer` command and SPADE's subgoaling to replicate the effect.

First we introduce the implication rule we want to use into the hypotheses. For this step we use the entire goal before and after rewriting.

`infer` *After* `->` *Before* `using inference(1).`

The rule `inference(1)` allows us to introduce a new hypothesis but because *After -¿ Before* isn't shown to be trivially true we are asked by SPADE if we want to subgoal. We answer `yes` we want to subgoal and `yes` the first subgoal it finds will do.

We now need to show that *After* `->` *Before* is true. To show this we enter a proof by implication attempt which moves *After* to the hypotheses and requires us to prove *Before*. We prove this by appealing to to the rule we wanted to use as the rewrite. Because *After* is now in the hypotheses we can perform the required rewrite on it in reverse using `infer`. This step relies on us being able to get at the subterm that is being rewritten so we use `simplify` to attempt to break up the hypotheses.

`infer` *subterm(Before)* `using` *Rule* `.`

This will introduce *subterm(Before)* into the hypotheses and this should be enough to enable SPADE's `done` command to prove it. This drops us out of the proof by implication and there will be nothing left to prove of the subgoal we use `done` again to drop us out to the main proof.

Now we have *After* `->` *Before* available as a hypothesis. To continue the proof we use `infer` again to infer the result we want from the rewrite.

`infer` *After* `using inference(1).`

This won't succeed but we will be asked if we want to subgoal we again reply `yes` twice and we have the resulting rewritten goal in the subgoal conclusion. After completing the rest of the proof we will drop out of the subgoal and be asked to justify the rewrite by proving *Before* from *After*. Since we now have *After* `->` *Before* as a hypothesis we can use SPADE's `forwardchain` command on it to introduce *Before* into the hypothesis using *After* as the justification. The problem is now trivial and `done` can be used to discharge it.

So we have:

Tactic	Command
`wave(`*Rule*`, `*Pos*`, imp, `*Before*`, `*After*`)` `simplify(`*Rule*`, `*Pos*`, imp, `*Before*`, `*After*`)`	`infer` *After* `->` *Before* `using inference(1).` `yes.` `yes.` `prove c#1 by implication.` `simplify.` `infer` *subterm(Before)* `using` *Rule*`.` `done.` `done.` `infer` *After* `using inference(1).` `yes.` `yes.` *more proof(...)* `forwardchain h#`n`.` `done.`

The translator carries around the number of hypotheses and can calculate the correct hypothesis number n to use from that.

5.1 Problems

If our `simplify` step doesn't work we will be unable to get at the subterm we want to rewrite and we will be unable to discharge the implication using our rule. In the examples we have used the subterm is contained in a conjunction of terms which are easily split apart by `simplify`.

If the *subterm(Before)* isn't enough to allow `done` to discharge *Before*. The proof by implication will fail. In the examples we have used the subterm is at the top or a high level and the subterm is enough to allow `done` to prove it.

6 Simplify

At certain points in the proof CLAM uses integer arithmetic and reasoning to evaluate the conclusion to `true`. At these times SPADE's `done` command will usually discharge these without any problem but sometimes if there are variables present and their bounds are represented by constants rather than values in the hypotheses `done` will be unable to simplify them. At these times it is necessary to add some information about the bounds to the hypotheses to allow SPADE to use `done`.

In the tactics this step is represented by:

`simplify(`*Rule*`, hyps, equ, `*Before*`, `*After*`)`

- *Rule*: The rule name.
- `hyps`: A flag indicating that the rewrite is to be introduced as a new hypothesis.
- `equ`: A flag indicating that the rewrite to be used is based on an equality.
- *Before*: The goal before the rewrite.
- *After*: The goal after the rewrite.

We introduce the new hypothesis using the `infer` command.

`infer` *Before* `=` *After* `using inference(1).`

We subgoal on this by answering `yes` twice and we now have to prove *Before* = *After*. Since the rewrite is based on an equality we can use replace on the new goal to rewrite *Before* to *After*.

`replace c#1` *Before* `by` *After* `using` *Rule*`.`

We reply `yes` and `no` to confirm the replacement. We are left with a trivial goal of the form *After* `=` *After* and can use `done` to get rid of it.

In the main proof we are now presented with CLAM evaluation step.

`simplify(eval, \textit{Pos}, equ, `*Before*`, true)`

- `eval`: An evaluation step.

- *Pos*: A flag indicating that the rewrite is to be introduced as a new hypothesis.
- `equ`: A flag indicating that the rewrite to be used is based on an equality.
- *Before*: The goal before the rewrite.
- `true`: The goal after the rewrite.

With the extra information provided by our new hypothesis SPADE's `done` command should now be able to perform this step.

So we have:

Tactic	Command
`simplify(`*Rule*`, hyps, imp,` *Before*`,` *After*`)`	`infer` *Before* `=` *After* `using inference(1).` `yes.` `yes.` `replace c#1:` *Before* `by` *After* `using` *Rule*`.` `yes.` `no.` `done.`
`simplify(eval,` *Pos*`, equ,` *Before*`, true)`	`done.`

6.1 Problems

The only problem with this is if even after the the extra information has been added the `done` command cannot simplify the goal. In the examples we have looked at we have come across an instance of this where a goal contained the `div` operator.

7 Fertilisation

In the CLAM planner an appeal to the hypotheses is called fertilisation.

`fertilize(`*Hyp*`)`

- *Hyp*: The hypothesis to appeal to.

Unfortunately SPADE doesn't contain a mechanism for directly appealing to the hypotheses but we can construct a custom rule to perform this step. The `fertilise(1)` conditional rule allows an occurrence of a hypothesis in the goal to be rewritten to `true`. It does this by specifying that `X` can rewritten to `true` if `X` is true. This means that although it is meant to allow fertilisation it might also be used as a simplify step.

So we have:

Tactic	Command
`fertilize(`*Hyp*`)`	`replace c#`*n*`:` *Hyp* `by true using fertilise(1).` `yes.` `no.`

n is the number of the conclusion to be rewritten and is carried around by the translator.

7.1 Problems

The only problem with this is the introduction of the custom rule. Though if kept in a standard place the loading of this rule could be in included in the commands generated by the translator.

8 Done and Ignored

The final step in the spade proof is use of the `done` command to discharge any remaining residue (usually `true`). In the tactic there is an occurrence of `done` to indicate when the proof of a certain branch has finished.

`done`

- `done`: Indicates that the proof is finished.

We translate this into SPADE's `done` command to complete the branch of the spade proof.

```
done.
```

Unfortunately the simplify translations also contain `done` at the end which will automatically cause SPADE to drop out of a proof branch. For this reason a special case of the simplify translation is used when it is followed by `done` to ensure that `done` does not appear in the command output more times than is necessary.

There are also parts of the plan that are specific to the planner and do not correspond to any step in the object level proof. The `annotate` method annotates the goal to allow wave rules to be applied by since the underlying term structure is left intact we can safely ignore this step.

If CLAM doesn't produce a plan for a particular conclusion of a verification condition because it is easily dealt with by the SPADE simplifier then we may still wish to do a translation of the plans for the remaining conclusions. To allow this we simply ignore the `undef` placeholder that CLAM places in the list of plans for ones that do not have a corresponding tactic to translate.

So we have:

Tactic	Command
`done`	`done.`
`simplify(eval,` *Pos*`, equ,` *Before*`, true) then done`	`done.`
`annotate(..., ...)`	-
`undef`	-

9 Conclusion

The translation works for the `filters` example but it's quite fragile and will probably break for other examples.

A Translations

Tactic	Command
plans([*subprogram*], *VC*, *H#*, [plan($Conc_1$, $Plan_1$), ..., plan($Conc_n$, $Plan_n$)])	newvc. yes. *VC*. consult('*subprogram*.rls'). *commands for* $Plan_1$ ... *commands for* $Plan_n$
wave(*Rule*, *Pos*, equ, *Before*, *After*) simplify(*Rule*, *Pos*, equ, *Before*, *After*)	replace c#*n*: *subterm(Before)* by *subterm(After)* using *Rule*. yes. no.
wave(*Rule*, *Pos*, imp, *Before*, *After*) simplify(*Rule*, *Pos*, imp, *Before*, *After*)	infer *After* -> *Before* using inference(1). yes. yes. prove c#1 by implication. simplify. infer *subterm(Before)* using *Rule*. done. done. infer *After* using inference(1). yes. yes. *more proof(...)* forwardchain h#*n*. done.
fertilize(*Hyp*)	replace c#*n*: *Hyp* by true using fertilise(1). yes. no.
simplify(*Rule*, hyps, imp, *Before*, *After*)	infer *Before* = *After* using inference(1). yes. yes. replace c#1: *Before* by *After* using *Rule*. yes. no. done.
simplify(eval, *Pos*, equ, *Before*, true)	done.
done	done.
simplify(eval, *Pos*, equ, *Before*, true) then done	done.
annotate(..., ...)	-
undef	-

B filter_sum example

For the filter_sum example there are two verification conditions for CLAM to prove, VC3 containing one conjecture to prove and VC6 containing two.

For VC6 the tactic output looks something like this:

```
plans(vc6,11,[
plan(c1,
simplify(simp3,[2],equ,r+element(d,[loop__1__i])>=integer__first,
        r+element(d,[loop__1__i])>= -32768) then
  trans(dec2,[],0+0,r+element(d,[loop__1__i])>= -32768,
          r+element(d,[loop__1__i])>=0+0 and 0+0>= -32768) then
    decomp(dec4,[1],0+0,r+element(d,[loop__1__i])>=0+0 and 0+0>= -32768,
            r>=0 and element(d,[loop__1__i])>=0 and 0+0>= -32768) then
      fertilize(r>=0) then
        fertilize(element(d,[loop__1__i])>=0) then
          simplify(logical_and(1),[1],equ,
                  true and true and 0+0>= -32768,true and 0+0>= -32768) then
```

```
            simplify(logical_and(2),[],equ,
                    true and 0+0>= -32768,0+0>= -32768) then
              simplify(eval,[],equ,0+0>= -32768,true) then
                done
),
plan(c2,
simplify(simp2,[2],equ,r+element(d,[loop__1__i])<=integer__last,
        r+element(d,[loop__1__i])<=32767) then
  trans(dec1,[],loop__1__i*100+100,r+element(d,[loop__1__i])<=32767,
          r+element(d,[loop__1__i])<=loop__1__i*100+100 and loop__1__i*100+100<=32767) then
    decomp(dec3,[1],loop__1__i*100+100,
            r+element(d,[loop__1__i])<=loop__1__i*100+100 and loop__1__i*100+100<=32767,
            r<=loop__1__i*100 and element(d,[loop__1__i])<=100 and loop__1__i*100+100<=32767) then
      fertilize(r<=loop__1__i*100) then
        fertilize(element(d,[loop__1__i])<=100) then
          simplify(logical_and(1),[1],equ,true and true and loop__1__i*100+100<=32767,
                  true and loop__1__i*100+100<=32767) then
            simplify(logical_and(2),[],equ,true and loop__1__i*100+100<=32767,
                    loop__1__i*100+100<=32767) then
              simplify(rr5,hyps,equ,i_type__last,9) then
                simplify(eval,[],equ,loop__1__i*100+100<=32767,true) then
                  done
)]).
```

On transformation this becomes:

```
newvc.
yes.
6.
replace c#1: integer__first by -32768 using filter_s_rules(3).
yes.
no.
infer r+element(d,[loop__1__i])>=0+0 and 0+0>= -32768 ->
        r+element(d,[loop__1__i])>= -32768 using inference(1).
yes.
yes.
prove c#1 by implication.
simplify.
infer r+element(d,[loop__1__i])>= -32768 using dec(2).
done.
done.
infer r+element(d,[loop__1__i])>=0+0 and 0+0>= -32768 using inference(1).
yes.
yes.
infer r>=0 and element(d,[loop__1__i])>=0 and 0+0>= -32768 ->
        r+element(d,[loop__1__i])>=0+0 and 0+0>= -32768 using inference(1).
yes.
yes.
prove c#1 by implication.
simplify.
infer r+element(d,[loop__1__i])>=0+0 using dec(4).
done.
done.
infer r>=0 and element(d,[loop__1__i])>=0 and 0+0>= -32768 using inference(1).
yes.
yes.
replace c#1: r>=0 by true using fertilise(1).
yes.
no.
replace c#1: element(d,[loop__1__i])>=0 by true using fertilise(1).
```

```
yes.
no.
replace c#1: true and true by true using logical_and(1).
yes.
no.
replace c#1: true and 0+0>= -32768 by 0+0>= -32768 using logical_and(2).
yes.
no.
done.
forwardchain h#13.
done.
forwardchain h#12.
done.
replace c#2: integer__last by 32767 using filter_s_rules(4).
yes.
no.
infer r+element(d,[loop__1__i])<=loop__1__i*100+100 and loop__1__i*100+100<=32767 ->
        r+element(d,[loop__1__i])<=32767 using inference(1).
yes.
yes.
prove c#1 by implication.
simplify.
infer r+element(d,[loop__1__i])<=32767 using dec(1).
done.
done.
infer r+element(d,[loop__1__i])<=loop__1__i*100+100 and loop__1__i*100+100<=32767
        using inference(1).
yes.
yes.
infer r<=loop__1__i*100 and element(d,[loop__1__i])<=100 and
        loop__1__i*100+100<=32767 -> r+element(d,[loop__1__i])<=loop__1__i*100+100 and
        loop__1__i*100+100<=32767 using inference(1).
yes.
yes.
prove c#1 by implication.
simplify.
infer r+element(d,[loop__1__i])<=loop__1__i*100+100 using dec(3).
done.
done.
infer r<=loop__1__i*100 and element(d,[loop__1__i])<=100 and
        loop__1__i*100+100<=32767 using inference(1).
yes.
yes.
replace c#1: r<=loop__1__i*100 by true using fertilise(1).
yes.
no.
replace c#1: element(d,[loop__1__i])<=100 by true using fertilise(1).
yes.
no.
replace c#1: true and true by true using logical_and(1).
yes.
no.
replace c#1: true and loop__1__i*100+100<=32767 by loop__1__i*100+100<=32767
        using logical_and(2).
yes.
no.
infer i_type__last = 9 using inference(1).
yes.
yes.
```

```
replace c#1: i_type__last by 9 using filter_s_rules(9).
yes.
no.
done.
done.
forwardchain h#15.
done.
forwardchain h#14.
done.
```

At the moment the custom and program specific rules need to be loaded into SPADE by hand.