Development notes for ProbCompCert

Sam Stites

December 16, 2021

Contents

The aim of this document aims to provide a roadmap to the current ProbCompCert codebase (revision 0f5530f6 on the feature/targetproof branch). It is a review of development workflows, debugging tips, and learning challenges for myself (stites). The current status of the work, and the expected next steps, form the final section of this document.

1 A motivating example

At a high level, when navigating `stanfrontend/`, all transformations can be summarized as being found in `driver/Sparse.ml`, responsible for parsing and type resolution, and `stanfrontend/Scompiler.v`, which runs the remainder of the pipeline.

Note that, at present, only the density compilation has been verified. Given a Stan program `simple.stan` a library in the form of a `simple.s` will be produced instead of a runnable binary. To perform inference on the described model function, the provided `Runtime.c` must be compiled with the resulting `simple.s` to produce a runnable binary. This is described in more detail in Compiling and running. The required C files for compiling are described in Additional C files.

Given the following program `simple.stan`:

```
cat ../stanfrontend/tests/simple/simple.stan
```

We initially want to take the parsed `Stan` AST (defined in Stan.v) and enrich it with type information (defined in StanE.v). This "elaborated Stan" forms our first intermediate representation and still closely resembles the original Stan AST.

Shortly after, StanE is transformed into CStan, which remains the intermediate representation of choice for the following, ordered, transformations:

1. converting sampling statements `_ ~ uniform(0,1)` and `_ ~ bernoulli(mu)` into external C calls from the packaged `libstan` library.
2. moving all free variables into one of two structs, the "parameters" struct for `mu` and the "data" struct for `flips`.
3. applying the constraint transformations of `lower=0.0` and `upper=1.0` to `mu` (according to the Stan documentation).
4. removal of the implicit target expression returned at the end of the `model` block.

Additional library functions will be generated to get and set a generated parameter struct as well as to initialize data from the command line (a work in progress).

2 Project structure

Slightly updating the initial proposed architecture[1] ProbCompCert has the following transformation passes:

From the originaly proposal the following changes have been made:

- the CStan is used for most transformations.

[1]Referencing the NSF grant proposal.

- the “Truncation” and “Vectors” passes have been removed and may be introduced at a later point.
- Denumpification (the StanE → CStan transformation) has been added.

`driver/Sparse.ml` is responsible for the first two transforms, the remaining transforms are found in `stanfrontend/Scompiler.v` the the function:

```
Definition transf_stan_program(p: StanE.program): res Clight.program :=
  OK p
  @@@ time "Denumpyification" Denumpyification.transf_program
  @@@ time "Sampling" Sampling.transf_program
  @@@ time "Constraints" Constraints.transf_program
  @@@ time "VariableAllocation" VariableAllocation.transf_program
  @@@ time "Target" Target.transf_program
  @@@ time "Backend" backend.
```

Note that no “Transformed parameters” exists in this pipeline for the time being.

2.1 `driver/Sparse.ml`

Sparse denotes a lot of the “messier” code which loads up manually defined external function signatures and puts them in the global scope for our StanE functions.

2.2 `stanfrontend/Denumpification.v`

Desugar StanE from a high-level language into a thin layer above Clight.

2.3 `stanfrontend/Sampling.v`

Convert sampling statements into external log-pdf or log-pmf calls.

2.4 `stanfrontend/Constraints.v`

Defines enough constraint transformations to correctly compile `simple.stan`.

2.5 `stanfrontend/VariableAllocation.v`

The intention of this pass is to takes the original list of free variables of parameters, which can be found on a CStan program, and creates a `Param` struct. It does a similar thing in order to construct the `Data` struct.

In addition, there are many other helper functions necessary to have the compiler generate in order to correctly implement the inference procedure defined in Runtime.c. These include:

- `get_state` and `set_state` – these get and set the global parameters struct and are used to perform parameter updates during inference
- `propose` provide a proposal parameter struct
- `print_state` and `print_data` which contains some naive generated debugging output

- `set_data` (work in progress) the ability to set the global data struct – this allows us to initialize this struct from the command line.

This pass additionally ensures that each of these functions correctly reference parameter or data structs. This is done by case analysis on the `CStan.blocktype` which is available on every `CStan.function` via `CStan.fn_blocktype`.

2.6 `stanfrontend/Target.v`

compile away any `target += s` statements, then compile away all `etarget` expressions.

2.7 `stanfrontend/Sbackend.v`

Convert a CStan program into a Clight program. This is as simple as stripping away extra stan-related information from the AST.

After this, the compiled Clight enters the CompCert pipeline at the `SimplExpr` transformation. The `SimplLocals` transformation is not used for ProbCompCert.

2.8 Proof code

Two proofs exist. Each proof uses it's own semantics under the following namings scheme:

`<Name>.v` transformation `Name`

`<Name>proof.v` proof of correctness

`CStanSemantics<Name>proof.v` operational semantics for CStan for this transformation pass.

`CStanCont.v` shared continuations across the operational semantics

The reason why we have transformation-specific operational semantics is superficial at the moment – primarily so that we can experimentally adjust the semantics for `Targetproof` without breaking the existing `Sbackendproof`.

2.8.1 `stanfrontend/Sbackendproof.v`

All but two cases are complete here. These are the cases that evaluating expressions are correct for the `cast` and `field struct` expressions. these have been admitted because `VariableAllocation.v` was produced after the proof was written. There are no forseen challenges with these cases, however the reference proofs of `SimplExprproof.v` and `Simpllocalsproof.v` are a little intricate so parsing these cases out is just a matter of time.

2.8.2 `stanfrontend/Targetproof.v`

This proof has proceeded very slowly and has hit a few stumbling blocks. Largely, this is not an issue with the nature of the proof – just my comfort with theorem proving.

One characteristic of this proof is that it introduces a prelude and epilogue to initialize the target identifier to 0, and to return this identifier as a result of program execution. This necessitates being comfortable with CompCert's continuations infrastructure – suggestions for understanding this can be found in Learning CompCert.

The current proof aims is straightforward in nature. there exists a target identifier on the `CStanSemanticsTarget.Model` which maintains the target density. This real must be in sync with, and preserve, the local target identifier that is being updated during execution of the program. As a program is executed it starts in a `CStanSemanticsTarget.block_state` of `Other` and, when calling the `model` function, initializes a `Model 0` state. When the model function has been transformed, the resulting state is discarded and computation continues in the `Other` state.

An alternative proof would have entailed postulating an existential target exists at the top level. This change represented a large refactor late into the development of this proof, and this course of action was not pursued extensively.

As I recall, a third way to approach this would have been to run a validator to garuntee that the target identifier had the correct semantics before proceeding with the correctness proof. I am a bit fuzzy on these details.

2.9 Additional C files

2.9.1 `Runtime.c`

A simple inference engine. This is very readable and should be intuitive.

2.9.2 `stanlib.c`

- Includes lpdf and lpmf functions used during the Sampling transformation
- Includes some helper sampling functions.
- some basic math functions exist like `logit` and `expit`. These are used during constraint transformation.
- `init_unconstrained` is the initialization function as determined by the Stan reference manual (all unconstrained parameters are sampled from Uniform(-2, 2))
- some rudamentary helper functions for printing are included. These should be removed in the long term.

2.9.3 `staninput.c`

This includes a first attempt at initializing the Data struct from the command line. Currently this is not functional.

2.10 Miscellaneous files

while in `stanfrontend/` we see:

```
ls ../stanfrontend | grep 'v$' | grep -Ev "(Stan|CStan|Scompil|Denum|Sampl|Constr|Variable|Targ
```

Runtime.v compcert-compiled Coq file of `Runtime.c`.

Sops.v Stan operators

Sparser.v The output parser from `Sparser.vy`

Stypes.v Stan types

Ssemantics.v A placeholder for the final semantics of a StanE program

Sutils.v An attempt to share utility functions across modules

System.v initial attempt to state the desired final theorem of ProbCompCert

3 Development Workflow

I had two development workflows when working on this project. For context, this project was developed on a NixOS box using flakes, but because of some version mismatches, opam was used to provide the development environment.

3.1 Opam information

Opam outputs an unreadable end-of-file character for emacs, here I supress the terminal output and paste it in a comment (see stan-development.org).

```
opam list --dev --normalise --color=never --readonly --columns=name,version

opam list --dev --normalise --color=never --readonly --columns=name,version --pin

# Packages matching: pinned
# Name # Version
coq    8.12.0
num    1.3
```

In addition to the pinned packages above, the following are probably relevant manually installed packages:

```
menhir                  20210419
merlin                  3.5.0
num                     1.3
ocaml-base-compiler     4.10.2
ocamlformat             0.18.0
sexplib                 v0.14.0
user-setup              0.7
utop                    2.7.0
```

3.2 Compiling and running

If you drop into a `nix develop` shell, you can access my helper scripts by invoking `menu`. I have this loaded with direnv via a `use flake` .envrc file.

3.2.1 Compiling ccompstan

To compile a program, you can use the `ccompstan` script found in flake.nix:

```
#!/usr/bin/env bash
set -euo pipefail

# store some directory variables.
current_dir=$PWD
root_dir=$(git rev-parse --show-toplevel)
stan_dir=$root_dir/stanfrontend

# store the name of the program
```

```
prog=${1##*/}
name=${prog%.*}
parent_dir="$(dirname -- "$(readlink -f -- "$1")")"
cd $root_dir

# working directory is stan dir
cd stanfrontend

# ccomp doesn't compile down to object files, just asm
ccomp -g3 -c $parent_dir/$name.stan && ccomp -c ${name}.s || exit 1

# build libstan.so
ccomp -g3 -c stanlib.c
ld -shared stanlib.o -o libstan.so

# runtime is dependent on libstan, temporarily.
ccomp -g3 -I${stan_dir} -c Runtime.c

# compile the final binary
ccomp -g3 -L${stan_dir} -Wl,-rpath=${stan_dir} -L../out/lib/compcert -lm -lstan ${name}.o Runti

# tell the user what to do next
echo "compiled! ./stanfrontend/runit INT"
```

3.2.2 Working on non-proof code

My workflow to prototype to non-proof code is to start a continuous loop of this script on save by using the `watchexec` tool. This is accessible via the `watch-stan-debug` command in flake.nix.

3.2.3 Working on proofs

For proof code. I use ProofGeneral in doom-emacs. I try to keep two windows open at a time so that I don't have to interrupt developing a proof in order to poke around a reference proof from somewhere else in CompCert.

I encountered two hiccups with ProofGeneral in doom emacs:

- if you use company-coq, the flycheck spinner takes up a lot of cpu cycles and can slow everything down. This was solved in a github issue on company-coq (but I just disabled this).
- doom emacs loads coq slowly because it loads PG to find the correct indent of the file, but this slows down files considerably. Joe coadvises a student who came up with this workaround:

  ```
  ;; the regular smie-config-guess takes forever in Coq mode due to some advice
  ;; added by Doom; replace it with a constant
  (defun my-smie-config-guess ()
    (if (equal major-mode 'coq-mode) 2 nil))
  (advice-add 'smie-config-guess :before-until #'my-smie-config-guess)
  ```

4 Debugging Tips

- If something goes terribly wrong and you need to `make clean` I usually comment out lines 44-47 (the coq-proba files) before running clean. This is largely safe to do.
- never `make`, always `make -j`
- Don't be afraid to hard code a lot of values and mess with Runtime.c
- If you do this, you will need to initialize a lot of types in `driver/Sparse.ml`

Originally, I updated `clightgen` to work with stan files. I compared the Coq-generated clight IR from a stan file to the `clightgen` output of reference C files (what I expected this output to be).

In retrospect, learning about the `dclight` and `dcminor` flags would have saved me a lot of time. The output of `clightgen` is not sufficient and is not as good of a representation as the true `*.light.c` files from the compilers. To invoke this on the `simple.stan` program you would use: `ccomp -dclight -dcminor -c stanfrontend/tests/simple/simple.stan`. This is instrumental in understanding where your errors come during compilation.

Prior to finding this out, I also resorted to adjusting the extracted ocaml (in `extraction/`) to debugging programs. This can be quite painful since Coq strings get broken down to lists of characters in OCaml and all changes are lost after a fresh `make`.

5 Learning CompCert

This project constitutes my first foray into learning Coq. Previously, I had some experience working with purely functional programming languages (namely working with Haskell on several moderate-to-large industry projects).

5.1 DSSS17 workshop

`https://xavierleroy.org/courses/DSSS-2017/`

Xavier's workshop proved very useful in understanding continuations and the overall structure of compcert. There are four videos on youtube as well which are worth reviewing before starting to hack on ProbCompCert.

After watching these videos, pull down the DSSS 2017 repository and go through the example problems. These were a challenging enough that I actually reset the solutions and redid these a couple of times.

(Aside, if you are using nixos, John Wiegley has a working nix-shell that needs to be pinned to any "pre coq-8.6" commit on nixpkgs[2]: `https://github.com/jwiegley/dsss17`

5.2 CompCert resources

The CompCert conference paper is too high-level to be useful in understanding the proof. You can give it a skim, but it will not assist in understanding how CompCert works. Instead, the Journal of Automated Reasoning article should be used.

[2] I used 5215ed6b216fedb37bfd241666048d9a4126b2b4

5.3 CompCert tactics

monadInv do this to apply inversion to any Result monad. sometimes, when it does not work, you will need to manually push terms through the monad and unfold it.

exploit less common. I forget exactly what this does, but it does get used in reference proofs.

5.4 Pen-and-paper proofs

It is important to understand what each symbol is doing before you run the code. To this end, I recommend writing out the goal and premises on paper. CompCert is a little too large, from my perspective, to do real pen-and-paper proofs – but the act of physically writing down a judgement tree really helped in this process.

In addition, a lot of effort needs to be

6 Current status

6.1 Expected next steps

Targetproof.