Report: Isar Chapter 5 and Isar Semantics

Brando Miranda

May 2020

1 Files

1.1 Chapter 5 Exercises

I completed exercises 5.1-5.6. They are located with their Isar proofs on the file `chapter5_exercises.thy` in the `project` folder.

Every proof has it's formal Isar proof with an accompanying bulk of text with the informal but rigorous proof in English. Each step of the English proof has a number that maps to the Isar proof line for improved clarify (of both proofs).

1.2 Isar constructs

I wrote out the examples from the textbook and made sure to use every construct at least once in the `isar_playground_pg.thy` and created my own examples/variations of their proofs. In this file I also play around with variations of the proofs I did for my exercises. Please skim this file because it also has alternative proofs to exercises and many experiments playing around the Isar language to understand it's semantics.

Note that some examples are deliberately very minimal, to help me understand how Isar really works.

I also have a few examples from the lecture notes and mp1 to really test how Isar works on step by step proofs without more complex tactics and step by step rules. For those check file `mp1_isar.thy` and `predicate_logic_examples_isar.thy`. These minimal examples were useful for me to understand the semantics of Isar. These minimal examples also helped me to gain confidence and practice on Isar.

File `transitive_closure_star.thy` is a file where I recall the definition of transitive closure a do the proof that it's indeed is transitive in apply style and Isar.

2 MSOS style semantics for Isar

2.1 Notation

Recall the notation for MSOS:

$$P \xrightarrow{\Delta} P'$$

where P, P' are program fragments and Δ a label describing the semantic configuration components both before and after the transition.

The modified notation that I in the handwritten notes is as follows:

$$\Gamma\ P\ Update(\Gamma)$$

where P is the isar code fragment and Γ is the initial proof states before executing P and is the final proof statesΓ after executing Γ. The function $Update$ express the delta Δ changes for the variables in Γ that do change. Note that for the ease of expressing the semantics I use pseudo-code, dot notation for calling helper functions that I would theoretically implement and updates to the states according with $\Gamma[key] = value$.

2.2 Why MSOS

The reason I choose something similar to Modular Structural Operational Semantics (MSOS) is because I discovered when I attempt to do the big/small step semantics that I kept needing to add new fields to the proof state. The reason for this is that to my understanding, in Isar, one needs to keep track of different things. For example, I realized that it was important to keep track of the proof depth for me to be able to delete the right local facts and add to the right place the new proved facts. To make this more of a state machine and allow flexibility to define names as I pleased as I went along the design was something that help design the semantics. In addition I only needed to specify what changed after Isar code was executed.

2.3 Proof level

There are also flags that I'd in theory keep track. One important one is something I call "proof_level" that keeps track of a unique name for the depth of the proof. For example if in Isar we decided to create a new local proof for a new show/have then we'd have unique name for that and thus using that name keep track of all the facts defined in it. At the end of the proof for a current proof level we'd remove all facts that belong originally to that level and nothing else and add the new fact to the previous proof level. Note that if this were implemented in a normal programming language (instead of something like Maude or inference rules), then after the recursive calls we'd simply return to the previous function call automatically and we wouldn't need to manually delete local facts (as these would be deleted automatically by the programming language).

2.4 Flags for assume

Note that there is an assume flag $\Gamma.assume$ that keeps track if we've done an assume statement, since those require a Unification check of our assumed facts

and the premises for the current subgoal/proof obligation. Otherwise, the `show` code fails and gives a refinement error. Note that I forgot to write the proof level in the proofs explicitly but they should be there.

2.5 Flags for have, show

We also need flags for the state machine to know if we are in a have or show statement. This is useful because when the complete a proof with either a `by method` or `qed` knows if to remove the current proved fact from the proof obligations or not, since a `have` only adds to facts to the local enviornment at the current proof level.

2.6 Syntactic Sugar

Note I did not introduce the semantics for trivial statements like `thus` since they are just a composition of other constructs that already have semantics.

Obtain statement

Since I forgot to include the `Obtain` statement in the semantics in handwriting I will give it here:

Γ `obtain where l: "P(x1 ... xn)"` $\Gamma[proof\_level.xi] =$ `xi` s.t. $P(x1...xi...xn) \equiv True$

3 Appendix

3.0.1 Cases

Consider the example from the `cases_playground.thy` for the lemma:

```
lemma shows "ev n ⟹ (n - 2)"
```

The syntax `rule ev.cases` produces 3 rules according to the inductive definition of `ev n` (that can be found in that file). The first one is `ev n` $\Longrightarrow$ `ev ?a` that requires unification. However, the syntax `proof (cases)` does the unification (and sometimes additional simplifying).

3.0.2 Case vs Induction

Note that induction and cases is very similar except that induction introduces the IH after doing the unification. For example consider the cases thm for ev:

`ev ?a` $\Longrightarrow$ `(?a = 0` $\Longrightarrow$ `?P)` $\Longrightarrow$ `(n. ?a = Suc (Suc n)` $\Longrightarrow$ `ev n` $\Longrightarrow$ `?P)` $\Longrightarrow$ `?P`

With induction you get that the mapping of $\sigma = \{x \rightarrow Suc((Sucn))\}$ as the final goal of a new proof obligation that would appear:

`ev ?x ?P 0` $\Longrightarrow$ `(n. ev n` $\Longrightarrow$ `?P n` $\Longrightarrow$ `?P (Suc (Suc n)))` $\Longrightarrow$ `?P ?x` more concretely, the IH is `?P n` introduced by the new `ev n` from the rule `evSS`.

3.0.3 Induction

The main files to read for my experiments to understand the semantics of how Isar did induction are in files `inductive_predicates_pg.thy` and `induction_pg.thy`.

3.0.4 With datatypes

The syntax for datatypes is `induction x` where `x` is usually the datatype to be broken down into the different cases for each constructor (and constants) along with the induction hypothesis these introduce. The datatype is very simple and simply introduces a new goal with the constructor in the new goal and the induction hypothesis in the assumptions. You can find plenty of simple examples I explored with different syntax in file `induction_pg.thy` but the general idea is this:

If you have goal `P x` and one of the constructors is `C x x' y` where the first two are of the same type then the goal and IH is going to be:

$\wedge$ `x x'.  [| P x ; P x' |]` $\Longrightarrow$ `P (C x x' y)`

3.0.5 With inductive predicates

The syntax for doing induction with inductive predicates is either the general `rule ev.induct` or `induction rule:  iter.induct`. The second one is the prefered method as it does the "obvious" unification. The first one requires you to do an manual unification with the assumption being "replaced". The file to check for this is `inductive_predicates_pg.thy`.

For example consider the following (made up) rule from my file:

`rule10:  "Q (x::tau)` $\Longrightarrow$ `I C0` $\Longrightarrow$ `I x` $\Longrightarrow$ `I (C3 x x' x'')` $\Longrightarrow$ `I (C2 x x')"`

and say we want to prove the implication:

`I x` $\Longrightarrow$ `P x`

what Isar will do is unify with the assumption `I x` and replace it with the rule assumptions and IH for the variables x of the right type.

$\wedge$ `Q x x' x''.  Q x` $\Longrightarrow$ `I C0` $\Longrightarrow$ `P C0` $\Longrightarrow$ `I (C3 x x' x'')` $\Longrightarrow$ `P (C3 x x' x'')` $\Longrightarrow$ `P (C2 x x')`

The first term `Q x` is an assumption from the rule. The second is an inductive predicate with a constant which introduces the same goal we are trying to prove as an IH with the constant. Similarly with `I (C3 x x' x'')` introduces the IH `P (C3 x x' x'')`. The ultimate goal for this proof obligation is `P (C2 x x')`.

Note that the universal quantification for `Q` is silly and I could have used then `for Q` construct to fix it. I did not because the inductive predicate I defined included a term with a constant without `Q` that I wanted for me to see how constants were treated in the semantics of Isabelle with inductive predicates and induction. But of course that is a silly inductive predicate that wasn't intended to model anything for real.

4 Big-Step attempt at Semantics of Isar

Note this section is just here for reference to my first attempt at Isar with Big-Step. I decided to not take this route because I had to keep changing all the rules because whenever I needed to introduce new things to proof environment. In particular because Isar needs to remember when it has done an assume or if it's in the state of in a `have` or a `show` statement (since one would remove a goal of a higher level proof level but the other would not). I also used naming to track proof levels (e.g. when going into a new proof that introduced it's own assumptions I used unique naming to identify to what proof level it's facts corresponded).

The way I chose to model Isar proving is with the triple `[g,a,c]` where they stand for goals to be proved, assumptions we have and context. Context models the facts that are available (even across subproofs) while assumptions are only locally available. In principle the context would also hold the whole set of facts available in your Isabelle theory file (or we could add another field to make it clearer). The modeling is in big-step semantics in this format: `[g,a,c] Code ⇓ [g',a',c']` however to make the notation simpler and less cluttered I decided to remove the down arrow.

Note, chose not to model syntactic sugar constructs like `thus` etc because they did not provide a conceptually meaningful addition to the exercise.

4.1 Inference Rules

Note, when defining the inference rules sometimes I will use short hands (e.g. I won't say what the propositions the names are mapping to) or I will simply add environments. When adding environment Isar uses the most recent name and overwrites the old ones according to my experiments.

Note that some features are not explicitly added (e.g. being able to reference facts by value rather than by name), but one can assume they are available. Including them explicitly in the changes in state of the proof would make the proofs harder to understanding by adding additional notation.

I extensively used the query functionality of jedit to inspect context, cases, terms and theorems that were being updated as the proofs progressed.

4.1.1 Proving commands

$$\frac{}{\texttt{[g,a,c] by method [g',a',c']}}\ \text{Rule 1}$$

$$\frac{}{\texttt{[g,a,c] proof - [g,a,c]}}\ \text{Rule 2}$$

4.1.2 Fact using in Isar

$$\frac{}{\texttt{[g,a,c] from l1 ... ln [g',a'+[l1,...,ln],c']}}\ \text{Rule 3}$$

Isar sequencing

$$\frac{\texttt{[g,a,c] C1 [g',a',c']} \qquad \texttt{[g',a',c'] C2 [g'',a'',c'']}}{\texttt{[g,a,c] C1 C2 [g'',a'',c'']}}\ \text{Rule Seq}$$

4.1.3 Show and Have

$$\frac{}{\texttt{[g,a,c] show prop [g', a',c']}}\ \text{Rule 4}$$

$$\frac{}{\texttt{[[|A1;...;An|] ==> P,[[|A1;...;An|],c] show prop [P,[[|A1;...;An|],c']}}\ \text{Rule 1}$$

$$\frac{}{\texttt{[g,a,c] have prop [prop,\{\},c']}}\ \text{Rule 5}$$

$$\frac{}{\texttt{[g,a,c] from l1 ... ln have prop [prop,\{l1, ..., ln \},c']}}\ \text{Rule 6}$$

Note without my second rule, my have semantics always deletes the assumptions. To avoid that I create an additional rule using from (instead of using sequencing).

4.1.4 Shows and Assumes (in lemma definition)

$$\frac{}{\texttt{[\{\},\{\},c] assumes l: prop1 ... propn [\{\},\{ l -> prop1 ... propn \},c]}}\ \text{Rule 7}$$

$$\frac{}{\texttt{[\{\},a,c] and l: prop1 ... propn [\{\},a+\{ l -> prop1 ... propn \},c]}}\ \text{Rule 8}$$

$$\frac{}{\texttt{[\{\},\{\},c] shows n: prop [prop,\{\},c]}}\ \text{Rule 9}$$

Note this is attempt is incomplete because once I realized this wasn't a very good way to define the semantics. Though they were useful for me to understand how Isar worked.