Project report TPT

Koen Wermer 3705951 Robert Hensing 3361063

July 4, 2015

Contents

1 Introduction

For this project, we have researched how a dependently typed language like Agda can be used to model the semantics of a language with mutable state using Agda as a proof checker.

The focus of our contribution is on the practical aspect of modeling, to show and compare some possible approaches to modeling mutable state in Agda.

We have extended a simplistic language with only boolean and number types and no lambda abstractions or any other sort of control flow structures. An inductive definition of the language is provided in section 3. In doing so, and proving theorems about this language, we have encountered some challenges, which we will describe in section 4. Some possibilities we would have liked to explore, but were too ambitious given the time frame, are described in section 5.

2 Related work

Swierstra

In his dissertation, Swierstra models some effects, of which, most prominently, mutable state. Agda is used to provide a total specification of effects, in order to be able to reason with it formally, which is a requirement for the implementation of Hoare Type Theory that is given.

Pierce

In this book includes an informal specification of mutable state.

Nanevski et al.

This paper introduces the concept of Hoare Type Theory, which allows tracking and enforcing of side effects. In the presence of polymorphism, it shows the soundness and compositionality of the theory, using separation logic to enable local reasoning.

3 Language definition

The term language we have extended with mutable state consists of

true, false, zero, succ are normal forms for boolean and number constants

if_then_else, iszero are built-in functions

No abstraction or binding constructs are included. We have extended the language with

var is a normal form for representing variables

ref creates a new cell with an initial value

:= assigns a value to an existing cell

! reads a value from an existing cell

=> make a pointer reference another cell

% evaluate the first term, discarding its value and then evaluating the second term

4 Findings

4.1 Small step semantics

4.1.1 Deterministic result term

We have proved that applying any two possible steps results in the same term. This proof, like the other two 'deterministic' proofs starts with case analysis on the possible steps. Although the type checker can automatically omit and check bogus combinations based on types, many cases remain.

4.1.2 Deterministic type of heap

We have proved that applying any two possible steps results in the same heap shape. While this theorem seems useful for proving "Deterministics values of heap", there was actually no need to use it in that proof. It is a desirable property to have, so we have kept it.

4.1.3 Deterministic values of heap

We have proved that applying any two possible steps results in the same heap state. To prove this lemma, we had to make many attempts to create type-correct pattern matches. In some cases we could not find a suitable order of pattern matches that type-checked, so we had to resort to case-specific lemma's.

4.1.4 Uniqueness of results

Using the 'deterministic' lemma's, we were able to prove easily that terms result in unique tuples of normal forms and heap states.

4.2 Denotational semantics

For the denotational semantics we started with the monadic specification by Swierstra as a basis. The denotational semantics would build on the monad, translating terms to GADT-defined monadic values, which can in turn be evaluated.

First, we adapted the monad to work with a heap that is defined by functions, instead of data types.

First attempt:

```
[[_]] : forall {ty f f'} -> (t : Term ty) -> St ty f f'
```

This allows to express the rule for allocation, but it is problematic in other cases where a subexpression needs to be evaluated and nothing is known about f'

Second attempt:

```
[[_]] : forall {ty f} -> (t : Term ty) -> St ty f (extend t f)
```

where extend is an appropriate function

This lets the caller know the environment after evaluating the subexpression, but it is not in constructor form and is therefore somewhat hard to reason with. It did work better than the earlier attempt, but we were not able to finish it.

In order to be able to implement the denotation semantics, we had to extend the representation of values to be able to reference the heap shape, because variable values have to carry the proof that they are consistent with a heap.

4.3 Hoare logic

We added a datatype for Hoare triples. A Hoare triple consists of a 2 state predicates (functions from State to Bool) and a term, together with a validity proof, i.e. a function of the following type:

```
(s s' : State f) -> Steps s t s'      v
-> Valid p s -> Valid q s'
```

Valid is a data type that provides a proof that applying to predicate to the state results in "True". Using this datatype we proved the rule of pre-condition strengthening:

```
{ty : Type} {f : TypeEnv} {t : Term ty}
{p p' q : State f -> Bool} -> HoareTriple p t q ->
({s : State f} -> Valid p' s -> Valid p s) ->
HoareTriple p' t q
```

as well as post-condition weakening:

```
{ty : Type} {f : TypeEnv} {t : Term ty}
{p q q' : State f -> Bool} -> HoareTriple p t q ->
({s : State f} -> Valid q s -> Valid q' s) ->
HoareTriple p t q'
```

We also tried proving the sequencing rule:

```
{f : TypeEnv} {t1 t2 : Term <>} {p q r : State f -> Bool} ->
HoareTriple p t1 q -> HoareTriple q t2 r ->
HoareTriple p (t1 % t2) r
```

However, this turned out be very hard to prove, because what the sequencing rule actually says is that if we want to create a Hoare triple for the term t1;t2, there exists a state during evaluation that we can use to combine the triples for t1 and t2. It seems intuitively clear that we arive in some state after evaluating t1, and the pre-condition for the second triple should hold for this state, but actually constructing this state and proving that the necessary properties hold is quite hard.

5 Further work

Besides having theorems about Hoare logic, the language could benefit from a 'Hoare type system'. Having such a type system will enhance the ability of the programmer to specify properties about stateful programs. Although Swierstra did include an implementation of a Hoare Type System, we did not find a chance to adapt that to fit our model.

The implemented language model is not sufficient for any non-trivial programming task. It lacks mechanisms for abstraction or non-trivial control flow. It could be extended with a control flow structure such as a while loop or with a fix point operator. It is desirable to preserve the strong normalization property, but this will put extra constraints on the language.

6 Literature

Swierstra, Wouter. A functional specification of effects. Diss. University of Nottingham, 2009.

Pierce, Benjamin C. Types and programming languages. MIT press, 2002.

Nanevski, Aleksandar, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. Journal of Functional Programming 18.5-6 (2008): 865-911.

Reynolds, John C. Separation Logic: A Logic for Shared Mutable Data Structures. LICS 2002.