Assignment B1

Julian Kaindl — k11908881 — kaindl.julian@gmail.com

Contents

Denotational Semantics

Our first task will be to define denotational semantics for the language given in assignment A. Our starting point is the grammar and type system from the previous assignment and the denotational semantics for the language with terms, formulas and procedures from the textbook. In particular figure 7.20 with all the entailed definitions.

In figure 7.20 there are already some definitions that are also applicable to this assignment so in this section we will only define semantics for the commands that are not given in figure 7.20 and our domain of expressions.

Commands

The changes here are mostly just passing through the environment and top of stack pointer.

$[\![C_1; C_2]\!]^e_a\langle s, s'\rangle :\Leftrightarrow$

$\quad \exists s_1 \in \mathit{State}.[\![C_1]\!]^e_a\langle s, s_1\rangle \wedge [\![C_2]\!]^e_a\langle s_1, s'\rangle$

$[\![\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]^e_a\langle s, s'\rangle :\Leftrightarrow$

$\quad [\![E]\!]\checkmark\langle s\rangle \wedge \texttt{if}[\![E]\!]^e_a\ \texttt{then}\ [\![C_1]\!]^e_a\langle s, s'\rangle\ \texttt{else}\ [\![C_2]\!]^e_a\langle s, s'\rangle$

$[\![\texttt{if}\ E\ \texttt{then}\ C]\!]^e_a\langle s, s'\rangle :\Leftrightarrow$

$\quad [\![E]\!]\checkmark\langle s\rangle \wedge \texttt{if}[\![E]\!]^e_a\ \texttt{then}\ [\![C]\!]^e_a\langle s, s'\rangle\ \texttt{else}\ s' = s$

$[\![\texttt{while}\ E\ \texttt{do}\ C]\!]^e_a\langle s, s'\rangle :\Leftrightarrow$

$\quad \texttt{let}$

$\qquad \texttt{inductive}\ W \ \subset \mathit{State} \times \mathit{State}$

$\qquad W\langle s, s'\rangle :\Leftrightarrow$

$\qquad\quad [\![E]\!]\checkmark\langle s\rangle \wedge \texttt{if}\ [\![E]\!]^e_a\ \texttt{then}\ \exists s_1 \in \mathit{State}.[\![C]\!]^e_a\langle s, s_1\rangle \wedge W\langle s_1, s'\rangle\ \texttt{else}\ s' = s$

$\quad W\langle s, s'\rangle$

Expressions

For expressions we adopt the semantics for terms and formulas from the textbook. In particular we adopt the definitions for $[\![\ .\ ]\!]\checkmark$ and $[\![\ .\ ]\!]_s$ from section “The Evaluation of Expressions” to work with our domain of expressions and instead of the state we evaluate the expression with regards to the environment and the top of stack pointer.

Since the semantics for our expressions are mostly trivial we skip these here and will show them with the implementation later.

Implementation

Extending the Typechecker and Grammar

Our first important step is to extend the typechecker and grammar to be able to annotate the syntax tree. We need this to enable procedure overloading and the lookup for the procedure semantics. Since the tagging is only important for procedures we leave the rest of the tree untagged. The changes consist of only a few new lines of code. In the grammar we add the two lists of sorts and in the typechecker we set those lists accordingly.

```
  ProcedureCall(Identifier, Expressions, Variables, Vec<Sort>, Vec<Sort>),
```

Store and Values

Our language has one domain of expression with integer and boolean values. Therefore we need a store that can assign addresses to both types of values. Therefore we create a new type.

```
pub enum Value {
  Int(i32),
  Bool(bool),
}
```

Now we can implement the store that maps addresses to values. It also stores and solely manages the top of the stack pointer. The `usize` type used for our addresses is a rust primitive that has system specific pointer size.

```
pub struct Store {
  top: usize,
  memory: HashMap<usize, Value>,
}
```

The following listing shows all methods provided by the store.

```
impl Store {
  pub fn new() → Store {
    Store {
      top: 0,
      memory: HashMap::new(),
    }
  }

  pub fn write_address(&mut self, address: usize, value: Value) {
    self.memory.insert(address, value);
  }

  pub fn read_address(&self, address: usize) → Value {
    self.memory.get(&address).unwrap().clone()
  }

  pub fn create_and_write(&mut self, value: Value) {
    Store::write_address(self, self.top, value);
    self.top += 1;
```

```rust
  }

  pub fn create_and_write_sequence(&mut self, values: Vec<Value>) {
    values
      .iter()
      .for_each(|v| Store::create_and_write(self, *v));
  }

  pub fn read_sequence(&self, start: usize, end: usize) -> Vec<Value> {
    (start..=end)
      .map(|i| self.memory.get(&i).unwrap().clone())
      .collect()
  }

  pub fn create_and_write_default(&mut self, sort: &Sort) {
    match sort {
      Sort::Int => Self::create_and_write(self, INT_DEFAULT),
      Sort::Bool => Self::create_and_write(self, BOOL_DEFAULT),
    }
  }

  pub fn set_top(&mut self, new_top: usize) {
    self.top = new_top;
  }
}
```

Environment and Procedure Semantics

The implementation for our environment is implemented in reference to definition 7.14 from the textbook. We have a struct that stores the mappings `env.var` and `env.proc`. Where the first maps variables to addresses and the second procedure environments to procedure semantics.

```rust
pub struct Environment {
  var: HashMap<Variable, usize>,
  proc: HashMap<ProcedureEnvironment, ProcedureSemantics>,
}
```

The `Variable` was already defined and just represents an identifier. The types for the procedure mapping have to be introduced. The `ProcedureEnvironment` type is also implemented on the basis of definition 7.14 and hold the identifier aswell as the sequence of sorts for both value and reference parameters.

```rust
pub struct ProcedureEnvironment {
  identifier: Identifier,
  value_param_sorts: Vec<Sort>,
  ref_param_sorts: Vec<Sort>,
}

impl ProcedureEnvironment {
  pub fn new(
    identifier: &Identifier,
    ss1: Vec<Sort>,
    ss2: Vec<Sort>,
  ) -> ProcedureEnvironment {
```

```
    ProcedureEnvironment {
      identifier: identifier.clone(),
      value_param_sorts: ss1,
      ref_param_sorts: ss2,
    }
  }
}
```

The `ProcedureSemantics` type stores a copy of the environment from the point of declaration (closure) as well as the procedure code (command) and the value und reference parameters. The `eval` function then implements the semantics where we create a new environment from the closure, set the variables and execute the command in the new environment.

```
pub struct ProcedureSemantics {
  closure: Environment,
  cmd: Command,
  args: Parameters,
  ref_args: Parameters,
}

impl ProcedureSemantics {
  pub fn new(
    closure: Environment,
    cmd: Command,
    args: Parameters,
    ref_args: Parameters,
  ) -> ProcedureSemantics {
    ProcedureSemantics {
      closure,
      cmd,
      args,
      ref_args,
    }
  }

  pub fn eval(&mut self, as1: Vec<usize>, as2: Vec<usize>, store: &mut Store) {
    let mut cmd_env = Environment::from(self.closure.clone());

    as1.iter().enumerate().for_each(|(i, a)| {
      cmd_env.set_variable_address(&self.args[i].0, *a);
    });
    as2.iter().enumerate().for_each(|(i, a)| {
      cmd_env.set_variable_address(&self.ref_args[i].0, *a);
    });

    eval_command(&self.cmd, &mut cmd_env, store);
  }
}
```

Program Semantics

In this and the following sections we define the rest of the semantics for our domains. All our semantic functions will take the environment and store as arguments. The program evaluation also need the input values and returns the output values.

```rust
pub fn eval_program(
  program: &Program,
  env: &mut Environment,
  store: &mut Store,
  input: &Vec<Value>,
) -> Vec<Value> {
  match program {
    Program::New(declarations, _identifier, args, cmd) => {
      declarations
        .iter()
        .for_each(|d| eval_declaration(d, env, store));

      let main_pointer = store.top; // pointer to first local in main

      if args.len() != input.len() {
        panic!("Wrong number of inputs");
      }

      input.iter().enumerate().for_each(|(i, val)| {
        if val.is_sort(args[i].1) {
          env.set_variable_address(&args[i].0, store.top);
          store.create_and_write(*val);
        } else {
          panic!("Input has invalid type(s)");
        }
      });

      eval_command(cmd, env, store);

      store.read_sequence(main_pointer, main_pointer + args.len())
    }
  }
}
```

Declaration Semantics

The variable declaration is very straight forward. In the procedure declaration we have to first create the procedure semantics and then store them in the environment. Since Parameters are of type `Vec<(Variable, Sort)>` we can deconstruct them to the sequences of sorts that we need for procedure environment.

```rust
fn eval_declaration(
  declaration: &Declaration,
  env: &mut Environment,
  store: &mut Store,
) {
  match declaration {
    Declaration::Var(variable, sort) => {
      env.set_variable_address(variable, store.top);
      store.create_and_write_default(sort);
    }
    Declaration::Procedure(identifier, args, ref_args, cmd) => {
      let closure = Environment::from(env.clone());
```

```
      let proc_sem = ProcedureSemantics::new(
        closure,
        cmd.clone(),
        args.clone(),
        ref_args.clone(),
      );
      let proc_env = ProcedureEnvironment::new(
        identifier,
        args.iter().map(|v| v.1).collect(),
        ref_args.iter().map(|v| v.1).collect(),
      );
      env.set_procedure_semantics(proc_env, proc_sem);
    }
  }
}
```

Command Semantics

Most variants just pass the environment and store through. Because we can use methods from the environment and store structs the variable assignment and variable block also don't need a lot of code. In the procedure call we first store the old stack pointer. Then we calculate the addresses for the expressions (value parameters) and find the addresses for the variables (reference parameters) from the environment. The values from the value parameters are then stored on the stack and we evaluate the procedure environment.

```
fn eval_command(cmd: &Command, env: &mut Environment, store: &mut Store) {
  match cmd {
    Command::None => (),
    Command::VarAssignement(variable, exp) => {
      let value = eval_exp(exp, env, store);
      let address = env.get_variable_address(variable);
      store.write_address(address, value);
    }
    Command::VarBlock(variable, sort, c) => {
      let mut cmd_env = Environment::from(env.clone());
      cmd_env.set_variable_address(variable, store.top);
      store.create_and_write_default(sort);
      eval_command(c, &mut cmd_env, store);
    }
    Command::Sequence(c1, c2) => {
      eval_command(c1, env, store);
      eval_command(c2, env, store);
    }
    Command::IfThenElse(exp, c1, c2) => {
      if let Value::Bool(is_true) = eval_exp(exp, env, store) {
        if is_true {
          eval_command(c1, env, store);
        } else {
          eval_command(c2, env, store);
        }
      }
    }
    Command::IfThen(exp, c) => {
      if let Value::Bool(is_true) = eval_exp(exp, env, store) {
```

```
        if is_true {
          eval_command(c, env, store);
        }
      }
    }
    Command::WhileDo(exp, c) => {
      while unwrap_bool(eval_exp(exp, env, store)) {
        eval_command(c, env, store);
      }
    }
    Command::ProcedureCall(identifier, exps, vars, ss1, ss2) => {
      let old_top = store.top;

      let val_addresses: Vec<usize> =
        (store.top..(store.top + exps.len())).collect();

      let ref_addresses: Vec<usize> = vars
        .iter()
        .map(|variable| env.get_variable_address(variable))
        .collect();

      let values: Vec<Value> =
        exps.iter().map(|exp| eval_exp(exp, env, store)).collect();
      store.create_and_write_sequence(values); //increments stack pointer

      let proc_env =
        ProcedureEnvironment::new(identifier, ss1.clone(), ss2.clone());

      let mut proc_sem = env.get_procedure_semantics(&proc_env);
      proc_sem.eval(val_addresses, ref_addresses, store);

      store.set_top(old_top);
    }
  }
}
```

Expression Semantics

Almost all expressions dont need the environment or store. Only the variable expression needs us to first loom up the address and then the value for the variable.

```
fn eval_exp(
  exp: &Expression,
  env: &mut Environment,
  store: &mut Store,
) -> Value {
  match exp {
    Expression::IntLiteral(value) => Value::Int(*value),
    Expression::BoolLiteral(value) => Value::Bool(*value),
    Expression::Variable(var) => {
      store.read_address(env.get_variable_address(var))
    }
    Expression::Sum(e1, e2) => {
      eval_exp(e1, env, store) + eval_exp(e2, env, store)
```

```
    }
    Expression::Product(e1, e2) => {
      eval_exp(e1, env, store) * eval_exp(e2, env, store)
    }
    Expression::Difference(e1, e2) => {
      eval_exp(e1, env, store) - eval_exp(e2, env, store)
    }
    Expression::IntNegation(e) => -eval_exp(e, env, store),
    Expression::Quotient(e1, e2) => {
      eval_exp(e1, env, store) / eval_exp(e2, env, store)
    }
    Expression::LessThanEqual(e1, e2) => {
      Value::Bool(eval_exp(e1, env, store) <= eval_exp(e2, env, store))
    }
    Expression::Equality(e1, e2) => {
      Value::Bool(eval_exp(e1, env, store) == eval_exp(e2, env, store))
    }
    Expression::BoolNegation(e) => !eval_exp(e, env, store),
    Expression::And(e1, e2) => {
      logical_and(eval_exp(e1, env, store), eval_exp(e2, env, store))
    }
    Expression::Or(e1, e2) => {
      logical_or(eval_exp(e1, env, store), eval_exp(e2, env, store))
    }
    Expression::TernaryConditional(e1, e2, e3) => {
      if let Value::Bool(is_true) = eval_exp(e1, env, store) {
        if is_true {
          return eval_exp(e2, env, store);
        } else {
          return eval_exp(e3, env, store);
        }
      }
      panic!()
    }
  }
}
```

Example

We still use the same example program from assignment A. The program computes the factorial for a given number and stores it in the second paramter. `factorial(value, result)`. The following listing is the linear representation of the program.

```
procedure is_greater_one(n: int; ref r: bool ) {
r := (not (n <= 1))
}

program factorial(value: int, result: int ) {
result := 1
var greater_one: bool
call is_greater_one(value; greater_one)
while greater_one do {
result := (result * value)
```

```
value := (value - 1)
call is_greater_one(value; greater_one)
}
}
```

We execute the program and print the results with following code.

```rust
fn main() {
  let mut program = factorial();

  println!("{}", program.str());

  let mut context: TypeContext = TypeContext::new();
  let result: TypeResult<Tag> = program.check(&mut context);

  match result {
    Ok(tag) => println!("\nTYPE CHECK: VALID {}", tag),
    Err(()) => println!("\nTYPE CHECK: ERROR"),
  }

  let mut env = Environment::new();
  let mut store = Store::new();

  let input = vec![Value::Int(5), Value::Int(0)];
  let output = eval_program(&program, &mut env, &mut store, &input);

  println!("\nINPUT\tOUTPUT");
  for i in 0..input.len() {
    println!("{0}\t{1}", input[i], output[i]);
  }
}
```

The output:

```
TYPE CHECK: VALID Program

INPUT   OUTPUT
5       1
0       120
```

As expected we have the result of 5! in the second parameter and the first parameter has been reduced to 1. The input of the second parameter doesn't matter since immediately set it to 1 in the program.