Assignment A

Julian Kaindl — k11908881

Contents

Grammar **2**

Type System **3**
- Program P 3
- Declaration D and Declarations Ds 3
- Parameters X 3
- Command C 4
- Expression E and Expressions Es 4
- Variable V and Variables Vs 5

Implementation **6**
- Grammar 6
  - Linear Representation — Command 7
  - Linear Representation — Parameters 8
- Typesystem 9
  - Program Rules 10
  - Command — IfThenElse 11
  - Expression — Binary Integer Operations 11

Examples **11**

Grammar

The first task is to define the grammar for our programming language. As a starting point we use the definition 7.11 (and all the definition it references) from "Thinking Programs". An important requirement is that the two domains for formulas and terms are combined into one syntactic domain of "expressions", that includes integers and booleans with literals and several operators. While we try to stay as close as possible to the grammar from mentioned definitions, where sensible we will try to make the language look more like a typical programming language. We define the syntax for our programming language with the following domains and grammar.

$$P \in Program,\ D \in Declarations,\ Ds \in Declarations,$$
$$X \in Parameters,\ C \in Command,\ E \in Expression,\ , Es \in Expressions,$$
$$V \in Variable,\ Vs \in Variables,\ S \in Sort,\ I \in Identifier,\ Z \in \mathbb{Z}$$

$$Sort = \{\texttt{int},\ \texttt{bool}\}$$

$$P ::= Ds\ \texttt{;}\ \texttt{program}\ I(X)\ C$$
$$D ::= \texttt{var}\ V \texttt{:} S \mid \texttt{procedure}\ I(X_1;\ \texttt{ref}\ X_2)\ C$$
$$Ds ::= \ \_ \mid Ds, D$$
$$X ::= \_ \mid X, V : S$$
$$C ::= V := E \mid \texttt{var}\ V : S \texttt{;} C \mid C_1 \texttt{;} C_2 \mid \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{if}\ E\ \texttt{then}\ C \mid \texttt{while}\ E\ \texttt{do}\ C \mid$$
$$\texttt{call}\ I(Es,\ Vs)$$

The changes here are that we introduce the domain of expressions and alter the grammar rules for commands to use this new domain. Of course now we can have statements like `if (1)` or `while (1)` in our syntax tree which is something we need to account for in the type checker later. We also define the set of possible sorts here, since there should only be integers and booleans.

$$E ::= \ V \mid Z \mid E_1 \texttt{+} E_2 \mid E_1 \texttt{*} E_2 \mid \texttt{-}E \mid E_1 \texttt{-} E_2 \mid E_1 \texttt{/} E_2 \mid \texttt{true} \mid \texttt{false} \mid E_1 \texttt{==} E_2 \mid \texttt{not}\ E \mid$$
$$E_1\ \texttt{and}\ E_2 \mid E_1\ \texttt{or}\ E_2 \mid E_1\ \texttt{?}\ E_2\ \texttt{:}\ E_3$$
$$Es ::= \ \_ \mid Es, E$$

The grammar definition for expressions basically combines the two definitions for terms and formulas (from Definition 2.1). To fulfill the requirements of the assignment we add integer operations (+, -, *, /). All the required boolean operators are already defined by 2.1, but for completeness we list them here anyways. There are also grammar rules we don't need and leave them out. Namely the first order logic quantifiers and all rules related to any of the symbol domains. To look more like a typical programming language the inline conditional operator is changed to E_1 `?` E_2 `:` E_3 and equivalence to E_1 `==` E_2, with two equal signs. Also all boolean connectives are written out.

Type System

Now we need to define the type-checking rules where we can also build on the existing type checker from the manuscript. We use the data structure Vt for holding variable typings and Pt for procedure typings. A variable typing $v \in Vt$ is of form $\langle V, S \rangle$. Assigning a sort to a variable. Likewise a procedure typing $p \in Pt = \langle I, Ss \rangle$ assigns an identifier to a list of sorts. Since we know all sorts and other elements of the signature are not really needed we can omit the signature Σ in all the following rules.

Program P

$$\frac{Ds\colon \text{declarations}(V_t, P_t) \qquad X\colon \text{parameters}(V_t', Ss) \qquad V_t'' = V_t \leftarrow V_t' \qquad P_t' = P_t \cup \{\langle I, Ss\rangle\} \qquad V_t'',\ P_t' \vdash C : \text{command}}{Ds;\ \texttt{program}\ I(X)\ C :\ \text{program}}$$

Since our language should have global procedures and variables we have a list of declarations before the first command of the program. This command can then use all the defined variables, procedures and the passed program arguments. For the possibility of recursion we also add the program identifier itself to Pt.

Declaration D and Declarations Ds

$$\frac{S \in \{\texttt{int}, \texttt{bool}\} \qquad Vt' = Vt \leftarrow \{\langle V, S\rangle\} \qquad Pt' = Pt}{Vt, Pt \vdash \text{var}\ V : S\colon \text{declaration}(Vt', Pt')}$$

$$\frac{\begin{array}{c} X_1 : \text{parameters}(Vt_1, Ss_1) \qquad X_1 : \text{parameters}(Vt_2, Ss_2) \qquad Vt_3 = Vt \leftarrow (Vt_1 \cup Vt_2) \\ \neg\exists V \in Variable, S_1, S_2 \in Sort.\ \langle V, S_1\rangle \in Vt_1 \wedge \langle V, S_2\rangle \in Vt_2 \\ Vt_3, Pt \vdash C : \text{command} \qquad Vt' = Vt \qquad Pt' = Pt \cup \{\langle I,\ \langle Ss_1, Ss_2\rangle\rangle\} \end{array}}{Vt, Pt \vdash \texttt{procedure}\ I(X_1; \texttt{ref}\ X_2)\ C : \text{declaration}(Vt', Pt')}$$

These are the rules for variable and procedure declarations respectively. When declaring a variable we know all the valid sorts and can check if a given sort is either `int` or `bool`. In the rule for a single procedure nothing changes. Also the following rules for lists of declarations stay the same as in figure 7.11.

$$\frac{Vt = \emptyset \qquad Pt = \emptyset}{\_ : \text{declarations}(Vt, Pt)} \qquad \frac{Ds : \text{declarations}(V_t', Pt') \qquad Vt', Pt' \vdash D : \text{declaration}(Vt, Pt)}{Ds; D : \text{declarations}(Vt, Pt)}$$

Parameters X

$$\frac{Vt = \emptyset \qquad Ss = [\,]}{\_ : \text{parameters}(Vt, Ss)} \qquad \frac{X : \text{parameters}(Vt', Ss') \qquad S \in \{\texttt{int}, \texttt{bool}\} \qquad \neg\exists S' \in Sort.\ \langle V, S'\rangle \qquad Vt = Vt' \leftarrow \{\langle V, S\rangle\} \qquad Ss = Ss' \circ [S]}{X, V : S : \text{parameters}(Vt, Ss)}$$

Here we also check the sort to be either `int` or `bool`. Other than that nothing changes.

Command C

$$\frac{\langle V, S\rangle \in Vt \qquad Vt \vdash E : \exp(S)}{Vt, Pt \vdash V := E : \text{command}} \qquad \frac{Vt_1 = Vt \leftarrow \{\langle V, S\rangle\} \qquad Vt_1, Pt \vdash C : \text{command}}{Vt, Pt \vdash \texttt{var} V : S; C : \text{command}}$$

$$\frac{Vt, Pt \vdash C_1 : \text{command} \qquad Vt, Pt \vdash C_2 : \text{command}}{Vt, Pt \vdash C_1; C_2 : \text{command}} \qquad \frac{Vt \vdash E : exp(S) \qquad S = \texttt{bool} \qquad Vt, Pt \vdash C : \text{command}}{Vt, Pt \vdash \texttt{if}\ E\ \texttt{then}\ C_1 : \text{command}}$$

$$\frac{Vt \vdash E : exp(S) \qquad S = \texttt{bool} \qquad Vt, Pt \vdash C_1 : \text{command} \qquad Vt, Pt \vdash C_2 : \text{command}}{Vt, Pt \vdash \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 : \text{command}}$$

$$\frac{Vt \vdash E : \exp(S) \qquad S = \texttt{bool} \qquad Vt, Pt \vdash C : \text{command}}{Vt, Pt \vdash \texttt{while}\ E\ \texttt{do}\ C : \text{command}} \qquad \frac{\begin{array}{c} Vt \vdash Es : \text{exps}(Ss_1) \qquad Vt \vdash Vs : \text{variables}(Ss_2) \\ \langle I,\ \langle Ss_1, Ss_2\rangle\rangle \in Pt \end{array}}{Vt, Pt \vdash \texttt{call}\ I(Es; Vs) : \text{command}}$$

Of course we could allow integers to represent a boolean value for example like `if (1)`. And then when adding semantics interpret integers in a special way when encountered in a boolean context. For example like in C where anything that is not 0 is interpreted as `true`. Since this judgement is not the responsibility of our type checker we keep things simple and only allow boolean expressions as conditionals.

Expression E and Expressions Es

Rules for literals and variables. The rules return the sort of the expression, which can be either `int` or `bool`.

$$Vt \vdash Z : \exp(\texttt{int}) \qquad Vt \vdash \texttt{true} : \exp(\texttt{bool}) \qquad Vt \vdash \texttt{false} : \exp(\texttt{bool}) \qquad \frac{\langle V, S\rangle \in Vt}{Vt \vdash V : \exp(S)}$$

Rules for integers:

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2 = \texttt{int}}{Vt \vdash E_1 + E_2 : \exp(\texttt{int})}$$

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2 = \texttt{int}}{Vt \vdash E_1 * E_2 : \exp(\texttt{int})}$$

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2 = \texttt{int}}{Vt \vdash E_1 / E_2 : \exp(\texttt{int})}$$

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2 = \texttt{int}}{Vt \vdash E_1 - E_2 : \exp(\texttt{int})} \qquad \frac{Vt \vdash E : \exp(S) \qquad S = \texttt{int}}{Vt \vdash -E : \exp(\texttt{int})}$$

Rules for booleans:

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2}{Vt \vdash E_1 == E_2 : \exp(\texttt{bool})} \qquad \frac{Vt \vdash E : \exp(S) \qquad S \Leftrightarrow \texttt{bool}}{Vt \vdash \texttt{not}\ E : \exp(\texttt{bool})}$$

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2 = \texttt{bool}}{Vt \vdash E_1\ \texttt{and}\ E_2 : \exp(\texttt{bool})}$$

$$\frac{Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad S_1 = S_2 = \texttt{bool}}{Vt \vdash E_1\ \texttt{and}\ E_2 : \exp(\texttt{bool})}$$

For the inline conditional operator we need the first expression to be of sort `bool` and the other two expressions to be of the same sort. Also like mentioned before, the syntax is changed.

$$\frac{\begin{array}{c} Vt \vdash E_1 : \exp(S_1) \qquad Vt \vdash E_2 : \exp(S_2) \qquad Vt \vdash E_3 : \exp(S_3) \\ S_1 = \texttt{bool} \qquad S = S_2 = S_3 \end{array}}{Vt \vdash E_1\ \texttt{?}\ E_2\ \texttt{:}\ E_3 : \exp(S)}$$

Rules for expressions:

$$\frac{Ss = [\,]}{Vt \vdash \_ : \text{exps}(Ss)} \qquad \frac{Vt \vdash Es : \text{exps}(Ss') \qquad Vt \vdash E : \exp(S) \qquad Ss = Ss' \circ [S]}{Vt \vdash Es, E : \text{exps}(Ss)}$$

Variable V and Variables Vs

$$\frac{Ss = [\,]}{Vt \vdash \_ : \text{variables}(Ss)} \qquad \frac{Vt \vdash Vs : \text{variables}(Ss') \qquad \langle V, S \rangle \in Vt \qquad Ss = Ss' \circ [S]}{Vt \vdash Vs, V : \text{variables}(Ss)}$$

Implementation

Now we implement the above domains and type checking rules in Rust. The following sections comment on important sections of the code, for the full implementation please refer to the source files.

Grammar

In the `grammar.rs` file we define our syntactic domains as enums and a trait, which is a shared that is implemented for each type. This function returns a string being the linear representation of an expression.

Since they are not terribly verbose the following listing shows all the domain declarations. We derive the `Clone` standard trait for all types to create copy from a reference and for types used as keys in a hash map later we also derive a few other traits. Other than that all domains are built with an **`enum`** and Variants for all grammar rules. Non terminal symbols of the grammar have arguments. If there is a recursive declaration the compiler can't know the size of the type and we have to use `std::boxed::Box` which is a container that allocates memory on the heap for the contained item. For the domain of commands we also add a rule that is not formally defined above. We need it here because the enum parameters are not optional. For example we always need to fill a `Command::Sequence(cmd1, cmd2)` with two valid commands. Therefore we add the `Command::None` variant which basically is a no-op command without a linear representation and no type checking.

```
3  type Declarations = Vec<Declaration>;
4  type Expressions = Vec<Expression>;
5  type Variables = Vec<Variable>;
6
7  #[derive(Clone, PartialEq, Eq, Hash)]
8  pub enum Identifier {
9    New(String),
10 }
11
12 #[derive(Clone, PartialEq, Eq, Hash)]
13 pub enum Variable {
14   New(String),
15 }
16
17 #[derive(Clone, PartialEq)]
18 pub enum Sort {
19   Int,
20   Bool,
21 }
22
23 #[derive(Clone)]
24 pub enum Program {
25   New(Declarations, Identifier, Parameters, Command),
26 }
27
28 #[derive(Clone)]
29 pub enum Declaration {
30   Var(Variable, Sort),
31   Procedure(Identifier, Parameters, Parameters, Command),
32 }
33
34 #[derive(Clone)]
35 pub enum Parameters {
```

```
  None,
  List(Box<Parameters>, Variable, Sort),
}

#[derive(Clone)]
pub enum Command {
  None,
  VarAssignement(Variable, Expression),
  VarBlock(Variable, Sort, Box<Command>),
  Sequence(Box<Command>, Box<Command>),
  IfThenElse(Expression, Box<Command>, Box<Command>),
  IfThen(Expression, Box<Command>),
  WhileDo(Expression, Box<Command>),
  ProcedureCall(Identifier, Expressions, Variables),
}

#[derive(Clone)]
pub enum Expression {
  IntLiteral(i32),
  BoolLiteral(bool),
  Variable(Variable),
  Sum(Box<Expression>, Box<Expression>),
  Product(Box<Expression>, Box<Expression>),
  Difference(Box<Expression>, Box<Expression>),
  IntNegation(Box<Expression>),
  Quotient(Box<Expression>, Box<Expression>),
  LessThanEqual(Box<Expression>, Box<Expression>),
  Equality(Box<Expression>, Box<Expression>),
  BoolNegation(Box<Expression>),
  And(Box<Expression>, Box<Expression>),
  Or(Box<Expression>, Box<Expression>),
  TernaryConditional(Box<Expression>, Box<Expression>, Box<Expression>),
}
```

With `trait LinearRepresentation` we define the before mentaioned trait to be implemented for all domains.

```
pub trait LinearRepresentation {
  fn str(&self) -> String;
}
```

Linear Representation — Command

Following listing shows how this trait is implemented for the domain of commands. First the Command is matched to one of the Variants and then the respective string is generated by recursively invoking `str()` on sub expressions. To make the linear representation more readable we add curly braces and line breaks to the strings. In other domains we also slightly alter the linear representatio. For example wrapping binary int and bool expressions in parenthesis.

```
impl LinearRepresentation for Command {
  fn str(&self) -> String {
    match self {
```

```
      Command::None ⇒ String::from(""),
      Command::VarAssignement(var, exp) ⇒ {
        format!("{0} := {1}", var.str(), exp.str())
      }
      Command::VarBlock(var, sort, cmd) ⇒ {
        format!("var {0}: {1}\n{2}", var.str(), sort.str(), cmd.str())
      }
      Command::Sequence(cmd_1, cmd_2) ⇒ {
        format!("{0}\n{1}", cmd_1.str(), cmd_2.str())
      }
      Command::IfThenElse(exp, cmd_1, cmd_2) ⇒ format!(
        "if {0} then \n{1} \nelse \n{2}",
        exp.str(),
        cmd_1.str(),
        cmd_2.str()
      ),
      Command::IfThen(exp, cmd) ⇒ {
        format!("if {0} then {{\n{1}\n}}", exp.str(), cmd.str())
      }
      Command::WhileDo(exp, cmd) ⇒ {
        format!("while {0} do {{\n{1}\n}}", exp.str(), cmd.str())
      }
      Command::ProcedureCall(id, exps, vars) ⇒ format!(
        "call {0}({1}; {2})",
        id.str(),
        exps
          .iter()
          .map(|exp| exp.str().to_owned())
          .collect::<Vec<String>>()
          .join(", "),
        vars
          .iter()
          .map(|var| var.str().to_owned())
          .collect::<Vec<String>>()
          .join(", "),
      ),
    }
  }
}
```

Linear Representation — Parameters

The domain of Parameters is the only list where the implementation is not solved by using a vector data type like `Vec<Parameter>` but rather a one to one translation of the formal rules. Because at the end of the day it doesn't really matter which of the two we choose we opt for the shorter vector based implementation for other list-like domains like declarations Ds or expressions Es (see listing for domain definitions on page 6). The biggest difference is that we need implement the `str()` function recursively.

```
impl LinearRepresentation for Parameters {
  fn str(&self) → String {
    match self {
      Parameters::None ⇒ String::from(""),
      Parameters::List(params, var, sort) ⇒ {
        let p_str = params.str();
```

```
        format!(
          "{0}{1} {2}: {3}",
          p_str,
          if p_str.is_empty() { "" } else { "," },
          var.str(),
          sort.str()
        )
      }
    }
  }
}
```

Just to also show how we would get the linear representation for a list of expressions Es we look at a part of the `str()` implementation for a command. Namely the one for a procedure call where we have a list of expressions and a list of variables as arguments to the procedure call.

```
      Command::ProcedureCall(id, exps, vars) => format!(
        "call {0}({1}; {2})",
        id.str(),
        exps
          .iter()
          .map(|exp| exp.str().to_owned())
          .collect::<Vec<String>>()
          .join(", "),
        vars
          .iter()
          .map(|var| var.str().to_owned())
          .collect::<Vec<String>>()
          .join(", "),
      ),
```

Typesystem

The implementation of the type system is in `type.rs`. This implementation doesn't tag the whole syntax tree but still recursively type checks all expressions and passes on a result type. In Rust `Result<T, E>` is used for exception handling. We define a custom result type with `type TypeResult<Tag> = std::result::Result<Tag, ()>`. If a function now returns this type it can either be a tag or an anonymous error, meaning there is no additional exception information since all we care about is if it fails or not. To implement the type systems rules for all our domains we define another trait. This trait is again implemented for all domains and returns `Ok(tag)` if an expression is well formed or `Err(())` if not.

```
pub trait TypeCheck {
  fn check(&self, context: &mut TypeContext) -> TypeResult<Tag>;
}
```

Tags are defined as enum. Some variants, namely `Parameters` and `Expression`, have arguments which act as return value.

```rust
pub enum Tag {
  Program,
  Declaration,
  Parameters(Vec<Sort>),
  Command,
  Expression(Sort),
  Variable,
  Identifier,
}
```

Before showing the implementation of the rules we have to create a data structure to store variable typings Vt and procedure typings Pt. The `TypeContext` struct stores both as hash maps.

```rust
pub struct TypeContext {
  variable_typing: HashMap<Variable, Sort>,
  procedure_typing: HashMap<Identifier, Vec<Sort>>,
}
```

Program Rules

As first example we look at the typechecker for a program P with one rule. The function `check_vec` is a helper for checking a vector of expressions, declarations in this case. The function just invokes `check(context)` for all elements of the vector, which then in turn adds the procedure typing to the passed context.

The additional rule for an identifier checks if it is an empty string or not. Before checking the command we check declarations and arguments, which update variable and procedure typings in the context. The `?` after an invocation of a `check` function unwraps the returned result and short circuits the evaluation if the result is an error.

After checking the identifier we have to create a new context for the program checking. We pass this context to the parameters checker which adds all arguments to the variable typings. Then we check the command in the new context.

```rust
impl TypeCheck for Program {
  fn check(&self, context: &mut TypeContext) -> TypeResult<Tag> {
    match self {
      Program::New(declarations, id, args, cmd) => {
        check_vec(declarations, context)?;
        id.check(context)?;
        let mut program_context = TypeContext::new();
        args.check(&mut program_context)?;
        cmd.check(&mut program_context)?;
      }
    }
    Ok(Tag::Program)
  }
}
```

For further examples we look at individual rules from commands and expressions.

Command — IfThenElse

Here `check_exp` is an auxiliary function that checks an expression and returns the sort. Other than that the checking is identical to the rules defined above. First we check the expression to be of sort `bool` and then we check the commands.

```
145       Command::IfThenElse(exp, cmd_1, cmd_2) ⇒ {
146         if check_exp(exp, context)? ≠ Sort::Bool {
147           return Err(());
148         }
149         cmd_1.check(context)?;
150         cmd_2.check(context)?;
151       }
```

Expression — Binary Integer Operations

```
190       Expression::Sum(e_1, e_2)
191       | Expression::Difference(e_1, e_2)
192       | Expression::Product(e_1, e_2)
193       | Expression::Quotient(e_1, e_2) ⇒ {
194         let s: Sort = check_exp_same_sort(e_1, e_2, context)?;
195         if s == Sort::Int {
196           Ok(Tag::Expression(s))
197         } else {
198           Err(())
199         }
200       }
```

Examples

We'll now go through two examples, which are defined in `examples.rs`. The first one is a program that calculates the factorial of a given number. Following function was the reference for implementing it in our language.

```
1  fn factorial(mut value: u32) -> u32 {
2    let mut result = 1;
3
4    while value > 1 {
5      result *= value;
6      value -= 1;
7    }
8
9    return result;
10 }
```

Since the definition is rather verbose it's not listed here, so please refer to the source file for that.

Since our language doesn't directly support a `>` operator, we do it with `⩽` and negation. Just to also use a procedure, we define one that takes a number and a reference to a boolean variable as arguments and sets the reference to true if the given number is greater than one. The linear representation of our factorial program looks as follows.

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

program factorial( value: int ) {
var 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)
}
}
```

For the other example, which is a simple expression, the code defining it is listed below. Because of the recursive data structure we need to box the subexpressions.

```
pub fn simple_expression() → Expression {
  Expression::Sum(
    Box::new(Expression::TernaryConditional(
      Box::new(Expression::LessThanEqual(
        Box::new(Expression::IntLiteral(1)),
        Box::new(Expression::IntLiteral(5)),
      )),
      Box::new(Expression::IntLiteral(5)),
      Box::new(Expression::IntLiteral(15)),
    )),
    Box::new(Expression::IntLiteral(5)),
  )
}
```

This produces the following linear representation. Some of the parenthesis would not be necessary but as a general rule all expressions are wrapped in parenthesis.

```
(((1 ⩽ 5) ? 5 : 15) + 5)
```

There are also erroneous counterparts for the last two given examples defined in `examples.rs` which of course can be printed as linear representation but the type check will fail with an error. To print the linear representation and type check them we have to change the main function slightly. The outputs of both failing examples are also given in `examples_output.txt` with a short comment on why they fail the type check.