Specification of the Amy language

Mutable Variables & Constant Propagation

Computer Language Processing

LARA

Autumn 2019

1 Syntax

The syntax of Amy is given formally by the context-free grammar of Figure 1. Everything spelled in *italic* is a nonterminal symbol of the grammar, whereas the terminal symbols are spelled in `monospace` font. * is the Kleene star, s^+ stands for one or more repetitions of s, and ? stands for optional presence of a symbol (zero or one repetitions). The square brackets [] are not symbols of the grammar, they merely group symbols together.

Before parsing an Amy program, the Amy *lexer* generates a sequence of terminal symbols (*tokens*) from the source files. Some notnerminal symbols mentioned, but not specified, in Figure 1 are also represented as a single token by the lexer. They are lexed according to the rules in Figure 2. In Figure 2, we denote the range between characters α and β (included) with $[\alpha - \beta]$.

The syntax in Figure 1 is an *overapproximation* of the real syntax of Amy. This means that it allows some programs that should not be allowed in Amy. To get the real syntax of Amy, there are some additional restrictions presented (among other things) in the following notes:

- The reserved words of Amy are the following: `abstract`, `Boolean`, `case`, `class`, `def`, `else`, `error`, `extends`, `false`, `if`, `Int`, `match`, `object`, `String`, `true`, `Unit`, `val`,`var`, `_` (the wildcard pattern).

  Identifiers are not allowed to coincide with a reserved word.

- The operators and language constructs of Amy have the following precedence, starting from the *lowest*:

  (1) `val`, `var`; (2) `if`, `match` (3) `||` (4) `&&` (5) `==` (6) `<`, `<=` (7) `+`, `-`, `++` (8) `*`, `/`, `%` (9) Unary `-`, `!` (10) `error`, calls, variables, literals, parenthesized expressions.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Module}^* \\
\mathit{Module} & ::= & \texttt{object}\ \mathit{Id}\ \{\ \mathit{Definition}^*\ \mathit{Expr}?\ \} \\
\mathit{Definition} & ::= & \mathit{AbstractClassDef} \mid \mathit{CaseClassDef} \mid \mathit{FunDef} \\
\mathit{AbstractClassDef} & ::= & \texttt{abstract class}\ \mathit{Id} \\
\mathit{CaseClassDef} & ::= & \texttt{case class}\ \mathit{Id}\ (\ \mathit{Params}\ )\ \texttt{extends}\ \mathit{Id} \\
\mathit{FunDef} & ::= & \texttt{def}\ \mathit{Id}\ (\ \mathit{Params}\ )\ :\ \mathit{Type}\ =\ \{\ \mathit{Expr}\ \} \\
\mathit{Params} & ::= & \epsilon \mid \mathit{ParamDef}\ [\ ,\ \mathit{ParamDef}\ ]^* \\
\mathit{ParamDef} & ::= & \mathit{Id}\ :\ \mathit{Type} \\
\mathit{Type} & ::= & \texttt{Int} \mid \texttt{String} \mid \texttt{Boolean} \mid \texttt{Unit} \mid [\ \mathit{Id}\ .\ ]?\ \mathit{Id} \\
\mathit{Expr} & ::= & \mathit{Id} \\
 & \mid & \mathit{Id}\ =\ \mathit{Expr} \\
 & \mid & \mathit{Literal} \\
 & \mid & \mathit{Expr}\ \ \mathit{BinOp}\ \ \mathit{Expr} \\
 & \mid & \mathit{UnaryOp}\ \mathit{Expr} \\
 & \mid & [\ \mathit{Id}\ .\ ]?\ \mathit{Id}\ (\ \mathit{Args}\ ) \\
 & \mid & \mathit{Expr}\ ;\ \mathit{Expr} \\
 & \mid & (\ \texttt{val} \mid \texttt{var}\ )\ \mathit{ParamDef}\ =\ \mathit{Expr}\ ;\ \mathit{Expr} \\
 & \mid & \texttt{if}\ (\ \mathit{Expr}\ )\ \{\ \mathit{Expr}\ \}\ \texttt{else}\ \{\ \mathit{Expr}\ \} \\
 & \mid & \mathit{Expr}\ \texttt{match}\ \{\ \mathit{MatchCase}^+\ \} \\
 & \mid & \texttt{error}\ (\ \mathit{Expr}\ ) \\
 & \mid & (\ \mathit{Expr}\ ) \\
\mathit{Literal} & ::= & \texttt{true} \mid \texttt{false} \mid (\ ) \\
 & \mid & \mathit{IntLiteral} \mid \mathit{StringLiteral} \\
\mathit{BinOp} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<} \mid \texttt{<=} \\
 & \mid & \texttt{\&\&} \mid \texttt{||} \mid \texttt{==} \mid \texttt{++} \\
\mathit{UnaryOp} & ::= & \texttt{-} \mid \texttt{!} \\
\mathit{MatchCase} & ::= & \texttt{case}\ \mathit{Pattern}\ \texttt{=>}\ \mathit{Expr} \\
\mathit{Pattern} & ::= & [\ \mathit{Id}\ .\ ]?\ \mathit{Id}\ (\ \mathit{Patterns}\ ) \mid \mathit{Id} \mid \mathit{Literal} \mid \_ \\
\mathit{Patterns} & ::= & \epsilon \mid \mathit{Pattern}\ [\ ,\ \mathit{Pattern}\ ]^* \\
\mathit{Args} & ::= & \epsilon \mid \mathit{Expr}\ [\ ,\ \mathit{Expr}\ ]^*
\end{array}
$$

Figure 1: Syntax of Amy

$$
\begin{array}{rcl}
\mathit{IntLiteral} & ::= & \mathit{Digit}^+ \\
\mathit{Id} & ::= & \mathit{Alpha}\ \ \mathit{AlphaNum}^* \text{ (and not a reserved word)} \\
\mathit{AlphaNum} & ::= & \mathit{Alpha} \mid \mathit{Digit} \mid \_ \\
\mathit{Alpha} & ::= & [\texttt{a} - \texttt{z}] \mid [\texttt{A} - \texttt{Z}] \\
\mathit{Digit} & ::= & [\texttt{0} - \texttt{9}] \\
\mathit{StringLiteral} & ::= & \texttt{"}\ \mathit{StringChar}^*\ \texttt{"} \\
\mathit{StringChar} & ::= & \text{Any character except newline and } \texttt{"}
\end{array}
$$

Figure 2: Lexical rules for Amy

For example,
`1 + 2 * 3` means `1 + (2 * 3)` and
`1 + 2 match {...}` means `(1 + 2) match {...}`.

A little more complicated is the interaction between `;` and `val`: the definition part of the `val` extends only as little as the first semicolon, but then the variable defined is visible through any number of semicolons. Thus `(val x: Int = y; z; x)` means `(val x: Int = y; (z; x))` and not `(val x: Int = (y; z); x)` or `((val x: Int = y; z); x)` (i.e. `x` takes the value of y and is visible until the end of the expression).

All operators are left-associative. That means that within the same precedence category, the leftmost application of an operator takes precedence. An exception is the sequence operator, which for ease of the implementation (you will understand during parsing) can be considered right-associative (it is an associative operator so it does not really matter).

- A `val` (or `var`) definition is not allowed directly in the value assigned by an enclosing `val` definition. E.g. `(val x: Int = val y: Int = 0; 1; 2)` is not allowed. On the other hand, `(val x: Int = 0; val y: Int = 1; 2)` is allowed.
- It is not allowed to use a `val` (or `var`) as a (second) operand to an operator. E.g. `(1 + val x: Int = 2; x)` is not allowed.
- A unary operator is not allowed as a direct argument of another unary operator. E.g. `--x` is not allowed.
- It is not allowed to use `match` as a first operand of any binary operator, except `;`. E.g. `(x match { ... } + 1)` is not allowed. On the other hand `(x match { ... }; x)` is allowed.
- The syntax [*Id* .]? *Id* refers to an optionally qualified name, for example either `MyModule.foo` or `foo`. If the qualifier is included, the qualified name refers to a definition `foo` in another module `MyModule`; otherwise, `foo` should be defined in the current module. Since Amy does not have the import statement of Scala or Java, this is the only way to refer to definitions in other modules.
- One line comments are introduced with "`//`": *`//This is a comment`*. Everything until the end of the line is a comment and should be ignored by the lexer.
- Multiline comments can be used as follows: *`/*This is a comment */`*. Everything between the delimiters is a comment, notably including newline characters and *`/*`*. Nested comments are not allowed.
- Escaped characters are not recognised inside string literals. I.e. `"\n"` stands for a string literal which contains a backspace and an "n".
- Variable assignments are to return the value of the variable after the assignation.

1.1 Implementation of changes

I added the keyword var and added the to the grammar the modifications that appear red in the lexical rules. For that I had to create an auxiliary class to obey the Scalion rules that prevent logical Or's from returning non-matching types.

I added the new class Assignation to every file of the pipeline, as defined in the specification in red.

2 Type checking

2.1 Implementation of changes

In the AST, local definitions contain whether the variables are mutable or immutable. During type checking we must also store to the environment the mutability of the local variables. Down below you can find the set of rules followed, showing how types are considered as Cartesian products (Type, Mutability).

$$\textsc{Variable}\ \frac{v : (T_1, T_2) \in \Gamma}{\Gamma \vdash v : (T_1, T_2)} \qquad \textsc{Int Literal}\ \frac{i \text{ is an integer literal}}{\Gamma \vdash i : (\texttt{Int}, \texttt{I})} \qquad \textsc{String Literal}\ \frac{s \text{ is a string literal}}{\Gamma \vdash s : (\texttt{String}, \texttt{I})}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash \texttt{()} : (\texttt{Unit}, \texttt{I})} \qquad \textsc{Boolean Literal}\ \frac{b \in \{\texttt{true}, \texttt{false}\}}{(\Gamma \vdash b : \texttt{Boolean}, \texttt{I})}$$

$$\textsc{Arith. Bin. Operators}\ \frac{\Gamma \vdash e_1 : (\texttt{Int}, \texttt{Any}) \qquad \Gamma \vdash e_2 : (\texttt{Int}, \texttt{Any}) \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : (\texttt{Int}, \texttt{I})}$$

$$\textsc{Arith. Comp. Operators}\ \frac{\Gamma \vdash e_1 : (\texttt{Int}, \texttt{Any}) \qquad \Gamma \vdash e_2 : (\texttt{Int}, \texttt{Any}) \qquad op \in \{\texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1\ op\ e_2 : (\texttt{Boolean}, \texttt{I})}$$

$$\textsc{Arith. Negation}\ \frac{\Gamma \vdash e : (\texttt{Int}, \texttt{Any})}{\Gamma \vdash \texttt{-}e : (\texttt{Int}, \texttt{I})}$$

$$\textsc{Boolean Bin. Operators}\ \frac{\Gamma \vdash e_1 : (\texttt{Boolean}, \texttt{Any}) \qquad \Gamma \vdash e_2 : (\texttt{Boolean}, \texttt{Any}) \qquad op \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : (\texttt{Boolean}, \texttt{I})}$$

$$\textsc{Boolean Negation}\ \frac{\Gamma \vdash e : (\texttt{Boolean}, Any)}{\Gamma \vdash \texttt{!}e : (\texttt{Boolean}, \texttt{I})} \qquad \textsc{String Concatenation}\ \frac{\Gamma \vdash e_1 : (\texttt{String}, \texttt{I}) \qquad \Gamma \vdash e_2 : (\texttt{String}, \texttt{I})}{\Gamma \vdash e_1\ \texttt{++}\ e_2 : (\texttt{String}, \texttt{I})}$$

$$\textsc{Equality}\ \frac{\Gamma \vdash e_1 : (T, Any) \qquad \Gamma \vdash e_2 : (T, Any)}{\Gamma \vdash e_1\ \texttt{==}\ e_2 : (\texttt{Boolean}, I)}$$

$$\textsc{Sequence}\ \frac{\Gamma \vdash e_1 : (T_1, Any) \qquad \Gamma \vdash e_2 : (T_2, T)}{\Gamma \vdash e_1\ \texttt{;}\ e_2 : (T_2, T)} \qquad \textsc{Asignation}\ \frac{\Gamma \vdash e_1 : (T_1, \texttt{M}) \qquad \Gamma \vdash e_2 : (T_1, \texttt{Any})}{\Gamma \vdash e_1\ \texttt{=}\ e_2 : (T_1, \texttt{M})}$$

$$\textsc{Local Immutable Variable Definition}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, n : (T_1, \texttt{I}) \vdash e_2 : T_2}{\Gamma \vdash \texttt{val}\ n\ \texttt{:}\ T_1\ \texttt{=}\ e_1\ \texttt{;}\ e_2 : T_2}$$

$$\textsc{Local Mutable Variable Definition}\ \frac{\Gamma \vdash e_1 : (T_1, \texttt{Any}) \qquad \Gamma, n : (T_1, \texttt{M}) \vdash e_2 : T_2}{\Gamma \vdash \texttt{var}\ n\ \texttt{:}\ T_1\ \texttt{=}\ e_1\ \texttt{;}\ e_2 : T_2}$$

Figure 3: Typing rules for expressions

Function/Class Constructor Invocation

$$\frac{\Gamma \vdash e_1 : (T_1, \texttt{Any}) \qquad \ldots \qquad \Gamma \vdash e_n : (T_n, \texttt{Any}) \qquad \Gamma \vdash f : (T_1, \ldots, T_n) \Rightarrow T}{\Gamma \vdash f(e_1, \ldots, e_n) : (T, \texttt{I})}$$

If-Then-Else

$$\frac{\Gamma \vdash e_1 : (\texttt{Boolean}, \texttt{Any}) \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if (}e_1\texttt{) \{}e_2\texttt{\} else \{}e_3\texttt{\}} : T}$$

Error

$$\frac{\Gamma \vdash e : (\texttt{String}, \texttt{Any})}{\Gamma \vdash \texttt{error(}e\texttt{)} : T}$$

Pattern Matching

$$\frac{\Gamma \vdash e : (T_s, \texttt{Any}) \qquad \forall i \in [1, n].\ \Gamma \vdash p_i : (T_s, \texttt{I}) \qquad \forall i \in [1, n].\ \Gamma, \mathit{bindings}(p_i) \vdash e_i : T_c}{\Gamma \vdash e\ \texttt{match \{ case}\ p_1\ \texttt{=>}\ e_1\ \ldots\ \texttt{case}\ p_n\ \texttt{=>}\ e_n\ \texttt{\}} : T_c}$$

Figure 4: Typing rules for expressions

2.2 Constant propagation

I implemented the following simplifications.

$$\textsc{If-Then-Else True}\ \frac{e_1 \rightsquigarrow \texttt{True}}{\texttt{if (}e_1\texttt{) \{}e_2\texttt{\} else \{}e_3\texttt{\}} \rightsquigarrow e_2} \qquad \textsc{If-Then-Else False}\ \frac{e_1 \rightsquigarrow \texttt{False}}{\texttt{if (}e_1\texttt{) \{}e_2\texttt{\} else \{}e_3\texttt{\}} \rightsquigarrow e_3}$$

$$\textsc{Boolean symmetry}\ \frac{\texttt{op} \in \{||, \&\&\}}{e_1\ \texttt{op}\ e_2 \rightsquigarrow e_2\ \texttt{op}\ e_1} \qquad \textsc{And 1}\ \frac{e_1 \rightsquigarrow \texttt{False}}{e_1\ \texttt{\&\&}\ e_2 \rightsquigarrow \texttt{False}} \qquad \textsc{And 2}\ \frac{e_1 \rightsquigarrow \texttt{True}}{e_1\ \texttt{\&\&}\ e_2 \rightsquigarrow e_2}$$

$$\textsc{Or 1}\ \frac{e_1 \rightsquigarrow \texttt{False}}{e_1\ \texttt{||}\ e_2 \rightsquigarrow e_2} \qquad \textsc{Or 2}\ \frac{e_1 \rightsquigarrow \texttt{True}}{e_1\ \texttt{\&\&}\ e_2 \rightsquigarrow \texttt{True}} \qquad \textsc{If-Then-Else True}\ \frac{e_1 \rightsquigarrow \mathit{False}}{\texttt{if (}e_1\texttt{) \{}e_2\texttt{\} else \{}e_3\texttt{\}} \rightsquigarrow e_3}$$

$$\textsc{Int Binary Op}\ \frac{e_1 \rightsquigarrow \texttt{n} \qquad e_1 \rightsquigarrow \texttt{m}}{e_1\ \texttt{op}\ e_2 \rightsquigarrow \texttt{n op m}} \qquad \textsc{Definitions}\ \frac{\texttt{va*}\ x = e_1\texttt{;}\ e2 \qquad e_1 \rightsquigarrow e_1^*}{e_2 \rightsquigarrow \texttt{subs}(e_2, x, e_1^*)}$$

$$\textsc{Function}\ \frac{e_i \rightsquigarrow e_i^*}{f(e_1, \ldots, e_i, \ldots, e_n) \rightsquigarrow f(e_1, \ldots, e_i^*, \ldots, e_n)}$$

Figure 5: Constant propagation

2.3 Implementation of changes

We move through the code remembering the values of variables and in-lining them when possible. All is performed in the file ConstantPropagation.scala. There you can find how it is done following a scheme that closely resembles the other steps of the pipeline.

3 Further information

All further information is commented on each file. On the respective position of the change.