Definitional Interpreter and Type Checker for Scilla in JavaScript

HOANG Ngoc Tram
A0169622Y
KEE Chong Wei Ulysses
A0229389L

CONTENTS

1. INTRODUCTION

Scilla is an intermediate-level language smart contract language developed for the Zilliqa blockchain. In the following project, we will describe the denotational and static semantics of Scilla, along with our implementations of a definitional interpreter and a type checker.

2. USER LEVEL DOCUMENTATION

All user-level documentation of Scilla can be found in the given Scilla docs https://scilla.readthedocs.io/. This project implements a definitional interpreter that evaluates pure Scilla, and a type checker that type checks pure and impure Scilla. Additionally, we present an IDE that can interpret and type check Scilla for those who wish to learn how to program in Scilla.

A. Ways to access the application

A.1. Web-based Interface

To access the definitional interpreter and type checker from the internet, simply head to https://granada.ulysseskee.software/.

A.2. Command-Line Interface

- Evaluator: the code to run the evaluator is located inside `testingEvaluator.js`. To run the evaluator, navigate to the root of the application folder, and enter `node testingEvaluator.js` into the command line and run.

Some helper functions have been added in `general.js` such as the ability to log output from the evaluator at certain points in time. To do this, you have the ability to add print logs using `addLineToLogOutput` - by default, log outputs are disabled. Turn it off using `setPrintTrue`.

- Type Checker: run `node testingTC.js` to test the type checker. All the files tested will be printed.

B. Definitional Interpreter

B.1. Differences from Scilla

- Recursion: A major quirk is that recursion is disabled in Scilla, however, our definitional interpreter does not ban recursion. As recursion is not disabled for the definitional interpreter, standard libraries in Scilla such as `ListUtils.scilib` that rely on recursion are implemented directly in Scilla and parsed as a standard library file. You can see these functions written in `PrimRec.scilib`.
- Match Expressions: For match expressions to be considered well-formed in Scilla, a pattern-match has to be exhaustive and every pattern must be reachable. We found this pattern-matching checks to be non-trivial to implement once recursive data types such as Lists or the Peano Numbers are considered. Instead, a run-time check is made using the value of the bounded variable provided for the match expression to check the reachability of the patterns given.

B.2. How it works

For the definitional interpreter, it is able to evaluate pure Scilla, namely the expressions that someone is able to write in Scilla. This means that the evaluator covers:

- Let expressions
- Function Declarations
- Function Applications
- Atomic expressions
- Builtin functions
- Messages
- Match expressions
- Constructors
- Type Functions

- Type Applications

Detailed denotational semantics for these expressions are described in the Developer Level Documentation.

B.3. Examples

```
(* Expected result: 42 *)
let x = Int64 42 in
let f = fun (z : Int64) =>
    let b = x in
    fun (c : Int64) => b
in
let a = Int64 1 in
let d = Int64 2 in
f a d
```

Here is an example of a simple program in Scilla and an overview of how the definitional interpreter would evaluate it.

This program begins with a let expression. In this case, the variable x is bound to the value `Int64 42` in the environment. The next expression to be evaluated is another let expression. However, note that the expression being evaluated here is a function declaration.

Function declarations are evaluated and saved as closures. Importantly, the environment at the time of the function declaration is saved within the closure to ensure lexical scoping. This means that if the function is applied later on in the program, it retains the information of the environment at the time it was declared. Closures are treated as values and the variable f is bound to this `closure`.

We then have two let expressions, where we bind a to `Int64 1` and bind d to `Int64 2`. f is then applied to a and the result of f to a is applied to d. z is bound to the value of a in the environment, namely `Int64 1`, and then the function body is evaluated. This results in the binding of b to x, which as explained earlier, would exist in the function's environment as `Int64 42`. b is bound to `Int64 42`. c is bound to the value of d in the environment, namely `Int64 2`, and then the function body is evaluated to return b which is the value `Int64 42`.

C. Type Checker

The type checker can type check both Scilla expressions and full Scilla contracts. Examples of Scilla expressions can be found in the repository here. Examples of Scilla contracts can be found here. You can either run the files in the repository, on on the online IDE mentioned above.

Detailed static semantics of how Scilla programs are type checked are described in the Developer Level Documentation.

3. DEVELOPER LEVEL DOCUMENTATION

The following section will describe the Scilla's semantics in detail, and how they are implemented. The tagged release for the following description is https://github.com/tramhnt99/scilla_parse/releases/tag/scillaJS.

A. Parser

A.1. ANTLR Grammar

The ANTLR Grammar written for Scilla in `scilla.g4` mimic the grammar written using Menhir in the original Scilla repo.

Most notable feature of ANTLR that we exploited were the context tags. For each syntactic piece, we add a tag using a # which would initiates the parser to create `Context` classes which we would look for instances of later on.

Here we show a snippet of statements grammar and how each statement is tagged.

```
stmt
    : l=identifier FETCH r=sid #Load
    | r=remote_fetch_stmt #RemoteFetch
    | l=identifier ASSIGN r=sid #Store
    | l=identifier EQ r=exp #Bind
    ...
```

Once ANTLR parses the program, it builds its own abstract syntax tree (AST) which we can traverse to evaluate or type check. However, we quickly realised how the AST is dense with unnecessary information and methods, which encouraged us to first translate all of the AST into our own implemented syntax before implementing the interpreter and type checker.

A.2. Implementing Scilla Syntax

This section details classes written in JavaScript which represent the Scilla's expressions, statements, and contract elements. In general, each syntactic piece is implemented as a JS class which contains information such as variable name, type, or expression as its properties.

The following classes can be found in `syntax.js`. The file itself is a concise documentation of the syntax. This section merely serves to verbosely elaborate what each syntactic structure means.

Expressions All expressions extend the class `ScillaExpr`.

- `Literal`: Implementations of literals are described in the later section.
- `Var`: Variables contain a property `var` of type `String`.
- `Let`: Let-bindings contain a variable name `x : String`, optional type of the variable `ty: ScillaType` (`ScillaType` is a JS class implemented for Scilla Types), an expression `lhs: ScillaExpr` which, once evaluated, would be bound to `x`, and an expressions `rhs: ScillaExpr` which is run with the updated enviroment.
- `Fun`: Functions, or lambdas, contain an single parameter `id: String` (since Scilla function are curried), its type `ty: ScillaType`, and the function body `e: ScillaExpr`.
- `App`: Function applications contain the function being applied `f_var: String` and a list of its arguments `args: String[]`.
- `Builtin`: Builtin functions contain the name of the function `b: String`, a list of types `targs: ScillaType[]` (as many builtin functions are polymorphic), and a list of argument variables `xs: String[]`.
- `Message`: Messages contain a list of objects containing the tag name `id : String` and either literals `l: ScillaLiterals` or variable names `v: String`.
- `Match`: Match expressions contain the variable being matched `x: String` and a list of pattern clauses `clauses: ClauseExp[]`. A `ClauseExp` contains a pattern `pat: Pattern` and a corresponding expression `exp: ScillaExpr`. There are three kinds of patterns that extend the class `Pattern`:
  - `WildCard`: Wildcards do not contain any further information.
  - `Binder`: Binders contain one identifier `x: String`.
  - `ConstructorPat`: Constructor patterns contain the constructor's name `pat: String` as well as a list of patterns `ps: Pattern[]` which continue pattern matching the constructor.
- `DataConstructor`: A data constructor expression contains the name of the data constructor `c: String`, a list of types it expects `ts: ScillaType[]`, and a list of arguments `args: String[]`.
- `TFun`: Type functions contain the name of type parameter `i: String` and the function's body `e: ScillaExpr`.
- `TApp`: Type applications contain the function name `f: String` and the list of types being applied `targs: ScillaType[]`.

Statements All statements extend the class `ScillaStmt`.

- `Load`: Loading contains variable name being declared `x: String` and an existing variable's name `r: String`.
- `RemoteLoad`: Loading remotely takes a variable name being declared `x: String`, an address to a contract `adr: String`, and a field name `r: String`.
- `Store`: Storing takes the name of the field `x: String` and a variable name `r: String`.
- `Bind`: Binding takes a variable name being declared `x: String` and an expression `e: ScillaExpr`.
- `MapUpdate`: Updating the map takes a map name `m: String`, a list of keys `klist: String[]`, and an optional argument `ropt: String` which indicates whether the map value is being updated (if the argument is given) or deleted (if it is not).
- `MapGet`: Accessing the map requires a variable name where the result is stored `x: String`, the name of the map `m: String`, the list of names of the keys `klist: String[]`, and a boolean value `fetchval: Bool` which indicates whether the user wants the value from the map or to check whether it exists.
- `RemoteMapGet`: Accessing the map remotely contains all the properties `MapGet` does as well as an address name `adr: String`.
- `ReadFromBC`: Reading from the blockchain takes a variable name `x: String` and a query type `bf: BCInfoQuery`. There are three kinds of queries that extend the class `BCInfoQuery`:
  - `CurrBlockNum`: Contains no additional properties.
  - `ChainID`: Contains no additional properties.
  - `TimeStamp`: containing a variable name `x: String`.
- `TypeCast`: Type-casting takes a variable `x: String` where the type casting results are stored, a variable `r: String` which would be type-cast to type `t: ScillaType`.
- `MatchStmt`: The properties of a pattern matching statement class mimic the pattern matching expression. The only difference lies in the clauses containing a list of statements rather than expressions.
- `AcceptPayment`: Contains no properties.
- `SendMsgs`: Sending messages contains the variable name that contains the messages `ms: String`.
- `CreateEvnt`: Creating an event contains a variable name with the event `param: String`.
- `CallProc`: Calling a procedure contains the procedure name `p: String` and a list of arguments `actuals: String[]`.

- `Iterate`: Iteration takes the name of a list variable `l: String` and a procedure name `p: String`.
- `Throw`: Throw an error takes an optional argument exception `eopt: String`.

Contracts Contracts are translated to contract modules `Cmodule` which contain the following:

- `smver: String`: Scilla version
- `libs: Library:` Optional library module defined in the contract. The `Library` class contains the name of the library `lname: String` and a list of lirbary entries `lentries: LibEntry[]`. There are two kinds of library entries:
  - `LibVar`: Library variables which contain both variabl declarations and function declarations. A `LibVar` would contain the name of the variable `x: String`, its optional type `tyopt: ScillaType`, and an expression `e: ScillaExpr`.
  - `LibType`: Library Types declares abstract data types. A `LibType` contains the name of the data type `x: String`, and a list of arguments `c: ContractDef[]`. A `ContractDef` would contain the name of the constructor `cname: String` and a list of types `cArgTypes: ScillaType[]` it takes.
- `elibs: (String * String * Lmodule)[]`: List of library imports. The first `String` is the name of the library, followed by another `String` which is an optional name space. The `Lmodule` is a library module, whose properties mimic a `Cmodule` but without a contract.
- `contr: Contract`: A Scilla Contract. A `Contract` class contains the following properties:
  - `cname: String`: Name of the contract.
  - `cparams: (String * SType)[]`: Contract parameters which contain the parameter name and parameter type.
  - `cconstraint: ScillaExpr`: Contract constraint.
  - `cfields: Field[]`: Contract mutable fields. A `Field` class contains the name of the field `name: String`, the type of the field `type: ScillaType`, and the inital expression the field contains `e: ScillaExpr` .
  - `ccomps: Component[]`: A list of components. A `Component` class would contain a the type of component it is `compType: ComponentType` , where the possible types are either a transition `CompTrans` or a procedure `CompProc`. The component type is followed by the name of the component `compName: String`, a list of its parameters `compParams : (String * ScillaType)[]`, and a component body `compBody: ScillaStmt[]` which is a list of statements.

Types and Literals Types and Literals are implemented in the same manner: each Scilla type and literal has its own class which would contain necessary information. For example, a literal of type `Int32` (which is its own class extending `ScillaType`) would be `Int32L(i)` where `i` is some integer value. In this implementation, all integer values would be of the same JS integer and would not differ in bit lengths.

An example of a type that contains properties is type `Map`. `Map` contains two properties `t1: ScillaType` and `t2: ScillaType`, where `t1` is the type of the map's keys and `t2` is the type of map's values.

The implementation of Scilla types and literals are in `types.js` and `literals.js` respectively.

A.3. Translate ANTLR Abstract Syntax Tree

To obtain an abstract syntax tree (AST) that follows the syntax implementation above, we traverse ANTLR's generated AST and construct our own. For example, when translating expressions, we check if the context is an instance of a specific ANTLR context generated from the grammar, followed by handling said context based on the information it holds.

```
translateSimpleExp(ctx) {
    if (!ctx) {
      return;
    }
    return ctx instanceof SP.LetContext
      ? this.translateLet(ctx)
      : ctx instanceof SP.FunContext
      ? this.translateFun(ctx)
      : ctx instanceof SP.AppContext
      ? this.translateApp(ctx)
      ...
```

These contexts are generated as a result of tagging (with #) every expression instance in the grammar (`scilla.g4`).

```
simple_exp
    : LET x=identifier (COLON ty=typ)? EQ f=simple_exp
   IN e=exp #Let
    | f_var=sid (args+=sid)+ #App
    ...
```

Each translate function then parses through the identifiers given in ANTLR to build corresponding objects. Taking the let-binding expression as an example, we see how we generate the appropriate properties to generate a new `Let` object, followed by recursively translating the next expression.

```
 translateLet(ctx) {
```

```
    if (!ctx) {
      return;
    }
    const x = ctx.x.getText();
    const type = ctx.ty !== null ? generateSType(ctx.ty
  ) : null;
    const value = translateExp(ctx.f);
    return new SE.Let(x, type, value, translateExp(ctx.
  e));
  }
```

It is important to note that the translation classes act as `Visitor()` classes for ANTLR, where `visitChildren()` is the entry point of the AST translations.

As translating the rest of Scilla from ANTLR's AST is quite straightforward, we won't be elaborating how the rest of Scilla is translated. Translation of pure Scilla, of impure scilla, of types and literals can be found in `syntaxVisit.js`, `translate.js`, `types.js`, and `literals.js` respectively.

B. Evaluator

B.1. Evaluating Expressions

We set Σ to be the representation of an environment that is a finite function that maps variables to value.

$\Sigma[x \mapsto v]$ is the function that is the same as Σ, excepts it maps x to v.

The evaluation relation that we use is $\Sigma \vdash e \Rightarrow v$. This is interpreted as under the environment Σ, we evaluate the expression e to a value v.

$$\frac{}{\Sigma \vdash i \Rightarrow i}\ \text{Literal} \qquad \frac{\Sigma(x) = v}{\Sigma \vdash x \Rightarrow v}\ \text{Var} \qquad \frac{\Sigma \vdash e_1 \Rightarrow v \quad \Sigma[x \mapsto v] \vdash e_2 \Rightarrow v_2}{\Sigma \vdash \text{ let } x = e_1 \text{ in } e_2 \Rightarrow v_2}\ \text{Let}$$

$$\frac{}{\Sigma \vdash \text{ fun}(x : T) \to e \Rightarrow (\text{fun}(x : T) \to e,\ \Sigma)}\ \text{Fun}$$

$$\frac{\Sigma \vdash e_1 \Rightarrow (\text{fun}(x : T) \to e,\ \Sigma') \quad \Sigma \vdash e_2 \Rightarrow v_2 \quad \Sigma'[x \mapsto v_2] \vdash e \Rightarrow v}{\Sigma \vdash e_1\ e_2 \Rightarrow v}\ \text{App}$$

$$\frac{\Sigma \vdash x \Rightarrow v \quad \Sigma^{(*)} \vdash e_i \Rightarrow v'}{\text{match } (x,\ [p_1 * e_1, \ldots, p_n * e_n]) \Rightarrow v'}\ \text{Match}$$

$$\frac{}{\Sigma \vdash \text{tfun } x \to e \Rightarrow (\text{tfun } x \to e,\ \Sigma)}\ \text{TFun}$$

$$\frac{\Sigma \vdash e \Rightarrow (\text{tfun } x \to e', \Sigma') \quad \Sigma' \vdash e'[x \mapsto T] \Rightarrow v}{\Sigma \vdash @e\ T \Rightarrow v}\ \text{TApp}$$

$$\frac{\Sigma \vdash x_1 \Rightarrow v_1 \quad \ldots \quad \Sigma \vdash x_n \Rightarrow v_n \quad c\ v_1 \ldots v_n \Rightarrow v}{\Sigma \vdash c\ x_1 \ldots x_n \Rightarrow v}\ \text{Constr}$$

$$\frac{\Sigma \vdash e_1 \Rightarrow v_1 \quad \ldots \quad \Sigma \vdash e_n \Rightarrow v_n \quad f\ v_1 \ldots v_n \Rightarrow v}{\Sigma \vdash \text{builtin } f\ e_1 \ldots e_n \Rightarrow v}\ \text{Builtin}$$

$$\frac{\Sigma \vdash t_1 \Rightarrow v_1 \quad \ldots \quad \Sigma \vdash t_n \Rightarrow v_n}{\Sigma \vdash \text{message } ([s_1 * t_1 \ldots s_n * t_n]) \Rightarrow \text{message } ([s_1 * v_1 \ldots s_n * v_n])}\ \text{Message}$$

The rules above are the denotational semantics of pure Scilla. Using these rules, we are able to evaluate Scilla expressions. This section below will elaborate on how each rule is implemented, detailing the implementation of evaluating expressions.

The function `evalExp()` takes in an expression and an evaluation environment and returns the value of the expression, as specified in the denotational semantics. Detailed here is how expressions are evaluated:

- **Literals**: As specified in the *Literal* rule above, literals evaluate to themselves.

- **Var**: For variables, we check the evaluation environment for a previous binding of the variable to a value, and we return the value.
- **Let**: For a let expression `let x = e1 in e2`, we first evaluate `e1`. `e1` evaluates to some value `v`, then the variable `x` is mapped to the value `v` in the evaluation environment. `e2` is then evaluated using this updated evaluation environment to some value `v2`. The result of the let expression is `v2`.
- **Fun**: Functions are evaluated to closures as specified in the denotational semantics. As such, we create a lambda function that takes in a single argument `x`, and save the current environment within the scope of the closure. This closure is the result of evaluating the function.
- **App**: For a function application, we first evaluate `e1` to a closure, this provides us with a lambda function and the environment Σ'. `e2` is then evaluated to a value `v2` in the current evaluation environment Σ. The argument `x` is then mapped to the value `v2` in the evaluation environment Σ' previously saved in the closure, which was done to ensure lexical scoping. `e` is then evaluated to a value `v` and is the result of the application.
- **Match**: For a match expression, match $(x, [p_1 * e_1, \ldots, p_n * e_n])$ its variable x is first evaluated to a value v. This value v is then matched against the patterns provided in the match expression.
  - In the case where the pattern is a `Wildcard`, no new bindings to the environment Σ is introduced.
  - In the case where the pattern is a `Binder`, then we extend the environment Σ to a new environment $\Sigma^{(*)}$ with the binding of the variable, specifically $\Sigma^{(*)} = \Sigma[x \mapsto v]$.
  - In the case where the pattern is a `Constructor`, no new bindings are introduced, however it may have nested patterns or no nested patterns.
    - In the case where the constructor has no nested patterns, no new bindings are introduced.
    - In the case where the constructor has nested patterns, these patterns are recursively evaluated.

  Once a pattern is matched, its corresponding expression e_i is evaluated under the environment $\Sigma^{(*)}$ to a value v'. This value v' is the result of the match expression.
- **Constr**: Given a constructor c, its arguments x_i are each evaluated to some value v_i under the current evaluation environment Σ. $c\ v_1 \ldots v_n$ is then interpreted as the creation of the Abstract Data Type (ADT) using these values v_i and the constructor c. The result is this ADT value. Note that c, which is a constructor, is itself looked up and its arity checked to match the number of arguments x_i provided.

- **TFun**: Type functions are evaluated to closures, similar to the `Fun` rule. As such, we create a lambda function that takes in a type variable `x`, and save the current environment within the scope of the closure. This closure is the result of evaluating the type function.
- **TApp**: First, we evaluate `e` to a closure, similar to a function application. This provides us with a lambda type function and the environment Σ'. $e'[x \mapsto T]$ is interpreted as: we substitute all free occurrences of x for T in e'. `e'` is the evaluated to some value `v` and is the result of the type application.
- **Builtin**: For a builtin expression, we first evaluate each e_i to its corresponding value v_i. The interpretation of $f\ v_1 \ldots v_n \Rightarrow v$ is interpreted as: we apply the builtin function f to the corresponding arguments vi.
- **Message**: For a message, $([s_1 * t_1 \ldots s_n * t_n])$ is interpreted as a message with a set of pairings of identifiers and variables, where s_i denotes the identifier and t_i denotes the variable. Each variable t_i is evaluated to some value v_i under the current evaluation environment Σ and the entry, composed of a pair $(identifier * variable)$ is added to the message. This message is then the result of the message expression.

B.2. Evaluating Builtins

For Scilla's builtin functions such as `eq`, `pow` and more, are all implemented internally. This means that it is not possible to interpret a Scilla program without first implementing these builtin functions as primitives in the interpreter. For our implementation of the definitional interpreter, we have written these builtin functions as primitives inside the `builtins.js` file.

Here is an example of the builtin function `add`:

```
add = (x) => (y) => {
    if (x instanceof IntLit && y instanceof IntLit) {
      if (x instanceof Int256L || y instanceof Int256L)
    {
        return new Int256L(x.i + y.i);
      } else if (x instanceof Int128L || y instanceof
   Int128L) {
        return new Int128L(x.i + y.i);
      } else if (x instanceof Int64L || y instanceof
   Int64L) {
        return new Int64L(x.i + y.i);
      } else if (x instanceof Int32L || y instanceof
   Int32L) {
        return new Int32L(x.i + y.i);
      }
    } else if (x instanceof UintLit && y instanceof
   UintLit) {
```

```
13      if (x instanceof Uint256L || y instanceof
    Uint256L) {
14        return new Uint256L(x.i + y.i);
15      } else if (x instanceof Uint128L || y instanceof
    Uint128L) {
16        return new Uint128L(x.i + y.i);
17      } else if (x instanceof Uint64L || y instanceof
    Uint64L) {
18        return new Uint64L(x.i + y.i);
19      } else if (x instanceof Uint32L || y instanceof
    Uint32L) {
20        return new Uint32L(x.i + y.i);
21      }
22     } else {
23      setError(new Error(`Error: builtin add of ${x}
    and ${y}`));
24      return;
25     }
26   };
```

We have curried functions for the builtin functions as these run on our definitional interpreter, which tries to be as faithful to the polymorphic lambda calculus of Scilla as possible. As such, the `add` function takes in two arguments sequentially, namely `x` and `y`. We then match these arguments that are passed in with the types that they belong to. For example, in the case of two `Int32L` literals that were passed in to `add`, they would be added and returned as a `Int32L`.

Depending on the builtin function that is being implemented internally, certain checks are made, and if they fail, an error is flagged.

C. Type Checker

C.1. Type Checking Expressions

Let Γ be our type environment and Δ be our type variables in scope. Additionally, let $:>$ be the subtyping notation, where $\mathtt{T}_1 :> \mathtt{T}_2$ indicates that $\mathtt{T}_1$ sub-types $\mathtt{T}_2$.

$$\frac{}{\Delta;\Gamma \vdash \mathtt{i} : \mathtt{Uint32}} \qquad \frac{}{\Delta;\Gamma \vdash \mathtt{i} : \mathtt{Int32}} \qquad \frac{}{\Delta;\Gamma \vdash \mathtt{i} : \mathtt{BNum}} \quad ... \quad \text{(Literals)}$$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Delta;\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(Var)} \qquad \frac{\Delta \vdash \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \quad \mathtt{T} :> \mathtt{T}_1 \quad \Delta;\Gamma, \mathtt{x} \vdash \mathtt{e}_2 : \mathtt{T}_2}{\Delta;\Gamma \vdash \mathtt{let\ x : T = e_1 ine_2} : \mathtt{T}_2} \quad \text{(Let)}$$

$$\frac{\Delta \vdash \mathtt{T} \quad \Delta;\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{e} : \mathtt{S}}{\Delta;\Gamma \vdash \mathtt{fun}(\mathtt{x} : \mathtt{T}) \Rightarrow \mathtt{e} : \mathtt{T} \to \mathtt{S}} \quad \text{(Fun)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}_1 \to \mathtt{S} \quad \Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{U}_1 \quad \mathtt{T}_1 :> \mathtt{U}_1}{\Delta;\Gamma \vdash \mathtt{ee}_1 : \mathtt{S}} \quad \text{(App)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}_1 \quad \Delta;\Gamma^{(*)} \vdash \mathtt{e}_1 : \mathtt{T}_2 \ \ ... \ \ \Delta;\Gamma^{(*)} \vdash \mathtt{e}_n : \mathtt{T}_2}{\Delta;\Gamma \vdash \mathtt{match}(\mathtt{e}, [\mathtt{pat}_1 * \mathtt{e}_1, ..., \mathtt{pat}_n * \mathtt{e}_n]) : \mathtt{T}_2} \quad \text{(Match)}$$

$\Gamma^{(*)}$ denotes updated type environment. How the environment updates depends on the kind of pattern matched (rules below).

$$\frac{\Delta;\Gamma \vdash \mathtt{Binder\,x} : \mathtt{T}_1 \quad \Delta;\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{e} : \mathtt{T}_2}{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}_2[\mathtt{x} := \mathtt{T}_1]} \quad \text{(PatBinder)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{Binder\,x} : \mathtt{T}_1 \quad \Delta;\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{e} : \mathtt{T}_2}{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}_2[\mathtt{x} := \mathtt{T}_1]} \quad \text{(PatWildCard)}$$

Finally, constructor patterns would have nester Binder or Wildcard patterns. The environment would update appropriately.

$$\frac{\Delta;\Gamma \vdash \mathtt{l} : \mathtt{T}_1 \to \mathtt{T}_n \to \mathtt{Constr}(\mathtt{T}_1 * ... * \mathtt{T}_n) \quad \Delta;\Gamma \vdash \mathtt{x}_1 : \mathtt{T}_1 ... \ \Delta;\Gamma \vdash \mathtt{x}_n : \mathtt{T}_n}{\Delta;\Gamma \vdash \mathtt{lx}_1 ... \mathtt{x}_n : \mathtt{Constr}(\mathtt{T}_1 * ... * \mathtt{T}_n)} \quad \text{(Constr)}$$

$$\frac{\Delta, \mathtt{X};\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{tfunX} \Rightarrow \mathtt{e} : \forall \mathtt{X}.\mathtt{T}} \quad \text{(TFun)} \qquad \frac{\Delta \vdash \mathtt{T}_2 \quad \Delta;\Gamma \vdash \mathtt{e} : \forall \mathtt{X}.\mathtt{T}_1}{\Delta;\Gamma \vdash @\mathtt{e} : \mathtt{T}_2 : \mathtt{T}_1[\mathtt{X} := \mathtt{T}_2]} \quad \text{(TApp)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{x}_1 : \mathtt{T}_1 ... \Delta;\Gamma \vdash \mathtt{x}_n : \mathtt{T}_n \quad \mathtt{s}_1 : \mathtt{T}_1, ..., \mathtt{s}_m : \mathtt{T}_m \in \mathtt{MF} \quad \mathtt{s}_{m+1} ... \mathtt{s}_n}{\Delta;\Gamma \vdash \mathtt{Message}([\mathtt{s}_1 * \mathtt{x}_1, ..., \mathtt{s}_m * \mathtt{x}_m, ..., \mathtt{s}_n * \mathtt{x}_n]) : \mathtt{Message}} \quad \text{(Message)}$$

MF is a set of mandatory fields with predetermined types.
All other labels $\mathtt{s}_{m+1} ... \mathtt{s}_n$ do not have predetermined types.

The rules above are the static semantics of pure Scilla. Using these rules, we were able to type check Scilla expressions. In this section, we will be elaborating how each rule is implemented, detailing the implementation of type checking expressions.

The function `typeExpr()` takes an expression and a type environment, and returns the same expression and its type. We know the program type checks if the function returns successfully. `typeExpr()`, which can be found in `typechecker.js`, type-checks expressions in the following way:

- **Literals**: For literals, we use the function `literalType()` to generate the type of the literal.

```
export function literalType(l) {
  return l instanceof Int32L
  ? new ST.Int32()
  : l instanceof Int64L
  ? new ST.Int64()
  ...

```

- **Var**: We check that the variable name is indeed in the type environment, and return the said type.
- **Let**: Given a let binding `let x = e1 in e2`, we first type check `e1`. Given `e1` is of type `T1`, if the let-binding declares `x`'s type to be some `T1'`, we check if `T1'` is a sub-type of `T1`, thus making `T1'` assignable to `T1`. Finally, we extend the environment with `x` bound to `T1` to type check `e2`. The whole expression is then of type of `e2`.
- **Fun**: Given a function that takes an argument of type `T`, we check that `T` is well-formed using `isWellFormedType()` (elaborated on in Section C.4). Knowing `T` is well-formed, we extend the environment with `x` bound to `T` with which we type check the body of the function. Given the type of the function body is `S`, we return a function type `FunType(T, S)`. The procedure of checking whether the function type applies to the given arguments is encapsulated in `functionTypeApplies()`.
- **App**: First we check whether the function and the argument variables exist in the type environment. Given they do, we check that the argument type checks with the function type, ie. whether the argument type is assignable to the function's parameter type. A type is considered assignable to another type if it is sub-typed by said type. Finally, we check whether the resulting application's type is well-formed.
- **Match**: First, we check that list of pattern matching clauses is not empty. Then, we type check each expression at each pattern matching clause to check that they must all be of the same type. If the pattern given binds a new variable (Binder pattern), then we extend the environment before type checking its corresponding expression. Otherwise, we type check without extending the environment. The type of the match expression is the same as the type of all expressions the match clauses.
- **Constr**: We begin type checking data constructors by making sure all arguments are of well-formed types. We then look up the data constructor,

and check if its arity matches the number of arguments given. Since our implementation of abstract data types (ADTs) requires them to be initiated with type variables (elaborated on in Section C.6), we instantiate the ADTs, followed by accessing the relevant constructor's type and type check whether the arguments passed are assignable. Finally, we return with an ADT type with the ADT's name and its type arguments.

- **TFun**: Due to the shadowing issue (elaborated on in C.7), we ensure that the type variable being declared is currently not in use. After extending the environment with the type variable, we type check the body of the function. The type of the type function is then `forall 'X. T` where `'X` is the type variable and `T` is the type of the body given the extended environment.

- **TApp**: We begin is ensuring all type arguments are well-formed. We then find the type of the type function either from the environment or from our dictionary of built-ins. We refresh the polymorphic function using `refreshPolyFun` to ensure now shadowing of type variables, before applying the type arguments to the polymorphic function type. In the process of applying type arguments, we remove the `forall` equivalent for each type parameter we resolve, followed by substituting the type varibles in the function type using `substTypeinType()`. The resulting type is the type of the expression.

- **Builtin**: If the builtin include type applications, then we check whether all type arguments are well-formed. We then check whether all arguments passed exist in the environment, and access them. Given the types of the arguments, we look up the built in function and resolve its type using `resolveBIFunType()` by polymorphic built-ins monomorphic. We then check if the resolved function type applies to the given arguments using `functionTypeApplies()`. Finally, we check if the resulting type is well-formed, which we then return.

- **Message**: For each message entry, if its tag is in the mandatory field, then it would have a predetermined type and we check whether the value its given is assignable to its predetermined type. Otherwise, any tags can contain values of any field. The type of the message depends on what mandatory field tag did it contain. For example, if the message contains a `_eventname` tag, then the message of type `EventTyp`.

C.2. Type Checking Statements

In this section, we will be elaborating on how the static semantics of Scilla statements shown below are implemented. Below we have hand-written (due to the lack of time) static semantics for Scilla statements.

STATEMENTS

<: Subtyping notation

LOAD

$$\frac{\Delta;\Gamma \vdash y:t \quad \Delta;\Gamma\{x \to t\} \vdash s_2}{\Delta;\Gamma \vdash \text{var } x = y; s_2}$$

STORE

$$\frac{\Delta;\Gamma \vdash x:T \quad \Gamma(f) <: T}{\Delta;\Gamma \vdash f := x}$$

REMOTE LOAD

$$\frac{\Delta;\Gamma \vdash c:\text{Address} \quad \Delta;\Gamma \vdash c.f:t \quad \Delta;\Gamma\{x \to t\} \vdash s_2}{\Delta;\Gamma \vdash \text{var } x = c.f; s_2}$$

BIND

$$\frac{\Delta;\Gamma \vdash e:t \quad \Delta;\Gamma\{x \to t\} \vdash s_2}{\Delta;\Gamma \vdash \text{var } x = e; s_2}$$

MAP UPDATE

$$\frac{\Delta;\Gamma \vdash m: \text{Map}(T_1 * \text{Map}(T_2 * \ldots T_n)) \quad \Delta;\Gamma \vdash k_1:T_1 \ldots \Delta;\Gamma \vdash k_{n-1}:T_{n-1} \quad \Delta;\Gamma \vdash v:T \quad T <: T_n}{\Delta;\Gamma \vdash m[k_1]\ldots[k_{n-1}] := v}$$

MAPUPDATE DEL

$$\frac{\Delta; \Gamma \vdash m : \mathrm{Map}(T_1 * \mathrm{Map}(T_2 * \ldots T_n)) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1}}{\Delta; \Gamma \vdash m[k_1] \ldots [k_{n-1}] := \text{undefined}}$$

MAPGET

$$\frac{\Delta; \Gamma \vdash m : \mathrm{Map}(T_1 * \mathrm{Map}(T_2 * \ldots T_n)) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1} \quad \Delta; \Gamma\{x \to T_n\} \vdash s_2}{\Delta; \Gamma \vdash \text{var } x = m[k_1] \ldots [k_{n-1}]; s_2}$$

MAPEXISTS_TRUE

$$\frac{\Delta; \Gamma \vdash m : \mathrm{Map}(T_1 * \mathrm{Map}(T_2 * \ldots T_n)) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1} \quad \Delta; \Gamma\{x \to \mathrm{Bool}\} \vdash s_2 \quad \Delta; \Gamma \vdash m[k_1] \ldots [k_{n-1}] : T_n}{\Delta; \Gamma \vdash \text{var } x = \text{True}; s_2}$$

MAPEXIST_FALSE

$$\frac{\Delta; \Gamma \vdash m : \mathrm{Map}(T_1 * \mathrm{Map}(T_2 * \ldots T_n)) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1} \quad \Delta; \Gamma\{x \to \mathrm{Bool}\} \vdash s_2 \quad \Delta; \Gamma \vdash m[k_1] \ldots [k_{n-1}] : \text{undefined}}{\Delta; \Gamma \vdash \text{var } x = \text{False}; s_2}$$

REMOTEMAPGET

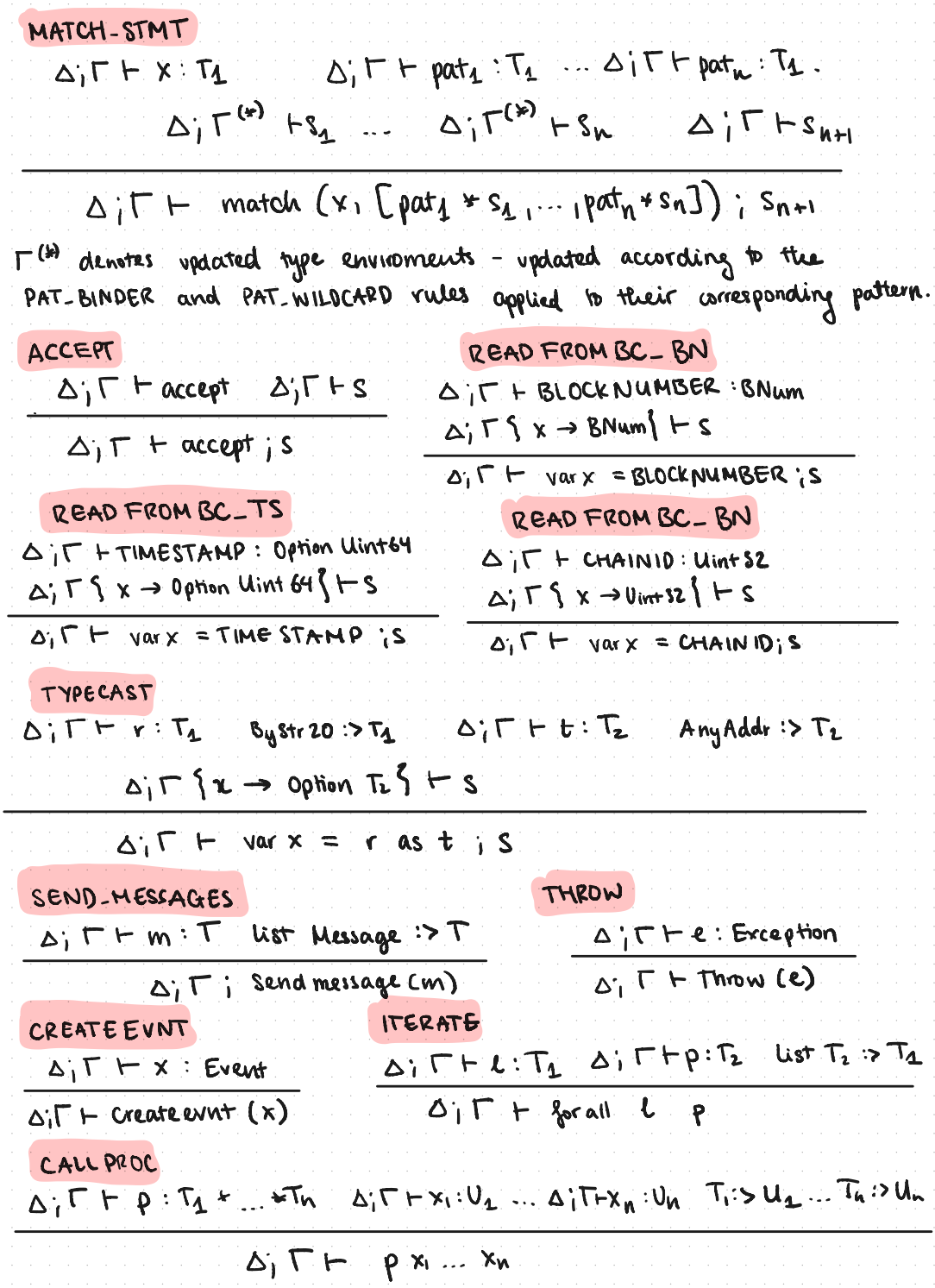
$$\frac{\Delta; \Gamma \vdash c : \text{Address} \quad \Delta; \Gamma \vdash c.m : \mathrm{Map}(T_1 * \ldots T_n) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1} \quad \Delta; \Gamma\{x \to T_n\} \vdash s_2}{\Delta; \Gamma \vdash \text{var } x = c.m[k_1] \ldots [k_{n-1}]; s_2}$$

REMOTE MAP EXISTS_TRUE

$$\frac{\Delta; \Gamma \vdash c : \text{Address} \quad \Delta; \Gamma \vdash c.m : \mathrm{Map}(T_1 * \ldots T_n) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1} \quad \Delta; \Gamma\{x \to \mathrm{Bool}\} \vdash s_2 \quad \Delta; \Gamma \vdash c.m[k_1] \ldots [k_{n-1}] : T_n}{\Delta; \Gamma \vdash \text{var } x = \text{True}; s_2}$$

REMOTE MAP EXISTS_FALSE

$$\frac{\Delta; \Gamma \vdash c : \text{Address} \quad \Delta; \Gamma \vdash c.m : \mathrm{Map}(T_1 * \ldots T_n) \quad \Delta; \Gamma \vdash k_1 : T_1 \ldots \Delta; \Gamma \vdash k_{n-1} : T_{n-1} \quad \Delta; \Gamma\{x \to \mathrm{Bool}\} \vdash s_2 \quad \Delta; \Gamma \vdash c.m[k_1] \ldots [k_{n-1}] : \text{undefined}}{\Delta; \Gamma \vdash \text{var } x = \text{False}; s_2}$$



Function `typeStmts()` type checks statements in the following manner:

- **Load**: For loading a variable, we check if the variable we are loading is bound in the type environment. We then extend the type environment is the new variable bound its the type we found, and run the consecutive statements.
- **RemoteLoad**: For remotely loading a variable, we check if the address variable is bound in the type environment. If it is, we look for the type of the field we are loading in the declared fields of the address's type. Finally, we extend the type environment with the new variable and the field's type, and run the consecutive statements.
- **Store**: Let us store variable `x` in a field `f`. We check that both types of `x` and `f` are stored in the type environment. Given `x: T` and `f: T'`, we check that `T` is a sub-type of `T'` in order for `x` to be assignable to `f`.
- **Bind**: Let us bind an expression `e` with a newly decalred variable `x`. We first type check `e`. Assuming `e` type-checks successfully, then we would know that `e` is of some type `T`. We extend the environment with `x` bound to type `T` before running consecutive statements.
- **MapUpdate**: Let us update a map `m` at a value accessed by a list of keys `kl` with a value `v`. We first check if the type of the map and the keys are bound in the environment. Then, we type check the map access by using the function `typeMapAccess()`. `typeMapAccess()` iterates through the list of keys and check whether the type of the key is assignable to the type corresponding type in the map. Given all key types type-check, we check if the value `v`'s type is assignable to the map. If `v` is not given (MapUdpateDelete rule), i.e., the user is trying to remove the value at the certain key, then we can move onto type checking the rest of the statements.
- **MapGet**: `MapGet` is implemented to represent both accessing the value and checking if it exists. Considering both options:
  - If the user is getting a value from the map, we begin with checking with the type of the map and keys is bound in the type environment. If they do, we type check the key access against the type of the map using `typeMapAccess()`. Given the type of the value being some `T`, we extend the environment with a newly declared variable bound to `Option T` before type-checking the rest of the statements.
  - If the user is checking if the value exists in the map, we follow all the same steps mentioned when getting the value. The difference lies in extending the environment with a newly declared variable bound to type `Bool` before type-checking the rest of the statements.
- **RemoteMapGet**: Getting a value from a remote map mimics the type checking implementations to `MapGet`. However, instead of checking if the map variable exists, we check if the address variable exists, and then check what is the type of the map field in the address.
- **TypeCast**: Given the user wants to bind to a variable `x` variable `r` type-casted as type `T`, we first begin with checking whether type `t` is assignable

to type `AnyAddr`, as Scilla only allow casts to address types. We then check whether the type of `r` is assignable to type `ByStr20`, as all address types are a sub-type of `ByStr20`. Finally, we extend the enviroment with `x` bound to `Option T` before running the rest of the statemetns.

- **MatchStmt**: Type checking the pattern matching statement is almost identical to pattern matching the match expression. The main difference lies in, since statements themselves do not have types, we do not need to ensure that all statements of each pattern clause are of the same type (as that is not possible).
- **AcceptPayment**: There is not type checking done for the accept statement.
- **ReadFromBC**: Depending on the type of query, we extend the environment by binding a newly declared variable `x` with a corresponding type. If the query is a
  - **Current Block Number**: We bind `x` with `BNum`.
  - **Chain ID**: We bind `x` with `Uint32`.
  - **TimeStamp**: We bind `x` with `Option Uint64()`.
- **SendMsgs**: First we check whether the message variable is bound in the type environment. Finally, check if the mesage variable's type sub-types the type `List MessageTyp`.
- **CreateEvnt**: First, we check whether the event variable is bound in the type environment. Given the event variable is bound in the type environment, we check the type is assignable to the type `EventTyp`.
- **CallProc**: First, we check whether the procedure's parameters type as well as the argument's type are bound in the environment. If they are, we check that the arguments' type is assignable to their corresponding parameter's type.
- **Throw**: First, we check whether the exception variable is bound in the type environment. If it is, we check if its type is assignable to type `ExceptionTyp` .
- **Iterate**: First, we check whether the list of values we are iterating over and the procedure are bound in the type environments. If they are, assuming the procedure is unary and the parameter is of type `T`, we check whether the list of values applied to the procedure's type is assignable to type `List T`.

C.3. Type Checking Scilla Contracts

As described in section A.2, a contract module comprises of a contract library, important libraries, and the contract. To type-check a contract module, we type check all of the those components.

A contract library, similarly to an imported library, comprises of library variables `LibVar` and library abstract data types `LibTyp`. The rule for type-checking a library variable is

$$\frac{\Delta \vdash \mathtt{T} \qquad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}' \qquad \mathtt{T} :> \mathtt{T}'}{\Delta;\Gamma \vdash \mathtt{let\ x : T = e : T}} \qquad \text{(LibVar)}$$

The following rule is implemented by firstly checking if the type of the library variable `T` is well-formed. If it is, then we type check the expression to get some type `T'`. We then check if `T` is assignable to `T'` using `typeAssignable()` (elaborated on in Section C.4). Finally, we bind the library variable in the type environment.

On the other hand, type-checking a library abstract data type simply requires making sure that each data constructor's type is well-formed. The interesting part lies on recording the necessary information about the abstract data type in the data type dictionary (elaborated on in Section C.6).

Having type checked the contract library and imported libraries, we move onto type checking the contract. In order to type check a contract parameter, we check whether the type of the contract parameters are well-formed. Since a contract constraint is an expression, we type check the expression and check if its type is assignable to type `Bool`. Type checking the fields is done by type checking the expressions assigned to the fields, and checking is the types of the expression is assignable to the declared type of the field. Finally, type checking the contract component simply means checking if the component parameters are well-formed and type checking the statements which are the component's body. The rules described are shown here:

$$\frac{\Delta \vdash \mathtt{T}_1 ... \Delta \vdash \mathtt{T}_n}{\Delta;\Gamma \vdash \mathtt{x}_1 : \mathtt{T}_1 * ... * \mathtt{x}_n : \mathtt{T}_n} \qquad \text{(ContrParam)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{Bool} :> \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{constraint(e : T)}} \qquad \text{(ContrConstraint)}$$

$$\frac{\Delta \vdash \mathtt{T} \qquad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}' \qquad \mathtt{T} :> \mathtt{T}'}{\Delta;\Gamma \vdash \mathtt{fieldf : T = e}} \qquad \text{(ContrFields)}$$

$$\frac{\Delta \vdash \mathtt{T}_1 ... \Delta \vdash \mathtt{T}_n \qquad \Delta;\Gamma[\mathtt{x}_1 \leftarrow \mathtt{T}_1 ... \mathtt{x}_n \leftarrow \mathtt{Tn1}] \vdash \mathtt{ss}}{\Delta;\Gamma \vdash \mathtt{component(x}_1 \mathtt{: T1, ... x}_n \mathtt{: T}_n\mathtt{)} = \mathtt{ss}} \qquad \text{(Component)}$$

C.4. Type Checker Utility Functions

In the following section, we will describe a few utility functions that that type checker heavily relies on.

`isWellFormedType()` is a function that takes a type, a type environment, and the abstract data type dictionary (section C.6). The function iterates through the type and checks whether its well-formed in the following manner:

- **Primitive, Unit, Any Address, Code Address, and Library Address types**: All these types are already well-formed.
- **Map type**: Map types are well-formed if both the key and value types are well-formed.
- **Function type**: Function types are well-formed if the type of the argument and function body are well-formed.
- **ADT type**: ADT types are well-formed if:
  - The ADT is already defined in the internal ADT dictionary.
  - The number of type arguments applied matches number of arguments the ADT can accept.
  - Each type argument applied is well-formed.
- **Type variable**: A type variable is well-formed if it is bound locally (i.e., we are inside a body of a type abstraction) or bound in the type environment.
- **Type abstraction**: A type abstraction type is well-formed if its body's type is well-formed, taking note of the type variable its abstracting.
- **Contract Address type**: A contract address type is well-formed if the type of its fields are all well-formed.

`typeAssignable()` is a function that dictates whether a type is a sub-type of another type, and thus whether one type is assignable to another. The system of types is described in the following rules:

$$\frac{T_1 : \texttt{AnyAddress} \quad T_2 : \texttt{LibAddress}|\texttt{CodeAddress}|\texttt{ContrAddr}}{T_1 :> T_2}$$

$$\frac{T_1 : \texttt{LibAddress} \quad T_2 : \texttt{LibAddress}}{T_1 :> T_2}$$

$$\frac{T_1 : \texttt{CodeAddr} \quad T_2 : \texttt{LibAddress}|\texttt{CodeAddress}|\texttt{ContrAddr}}{T_1 :> T_2}$$

$$\frac{T_1 : \texttt{ContrAddr}(f_1 : M_1 * ... f_n : M_n) \quad T_2 : \texttt{ContrAddr}(f_1 : U_1 * ... f_m : U_m)}{T_1 :> T_2 \quad \text{if} \quad n \leq m \quad M_1 :> U1 ... M_n :> Un}$$

$$\frac{T_1 : \texttt{ByStr20} \quad T_2 : \texttt{AnyAddress}|\texttt{LibAddress}|\texttt{CodeAddress}|\texttt{ContrAddr}}{T_1 :> T_2}$$

$$\frac{T_1 : \texttt{Map}(U_1 * U_2) \quad T_2 : \texttt{Map}(M_1 * M_2) \quad U_1 :> M_1 \quad U_2 :> M_2}{T_1 :> T_2}$$

$$\frac{T_1 : U_1 \to U_2 \quad T_2 : M_1 \to M_2 \quad M_1 :> U_1 \quad U_2 :> M_2}{T_1 :> T_2}$$

$$\frac{T_1 : \texttt{ADT}(U_1 * ... U_n) \quad T_2 : \texttt{ADT}(M_1 * ... M_n) \quad U_1 :> M_1 ... U_n :> M_n}{T_1 :> T_2}$$

$$\frac{T_1 : \forall X.U \quad T_2 : \forall Y.M \quad X = Y \quad U :> M}{T_1 :> T_2}$$

If the two types are not of these types, then we check whether the two types are equal. Two primitive types are equal is they are of the same primitive type. Otherwise, two types are equal according to the following rules:

$$\frac{T_1 : \texttt{TypeVar}(X) \quad T_2 : \texttt{TypeVar}(X)}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{Unit} \quad T_2 : \texttt{Unit}}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{ADT}(M_1 * ... * M_n) \quad T_2 : \texttt{ADT}(U_1 * ... * U_n) \quad M_1 = U_1 ... M_n = U_n}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{Map}(M_1 * M_2) \quad T_2 : \texttt{Map}(U_1 * U_2) \quad M_1 = U_1 \quad M_2 = U_2}{T_1 = T_2}$$

$$\frac{T_1 : \forall X.U \quad T_2 : \forall X.M \quad U = M}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{AnyAddress} \quad T_2 : \texttt{AnyAddress}}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{CodeAddress} \quad T_2 : \texttt{CodeAddress}}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{LibAddress} \quad T_2 : \texttt{LibAddress}}{T_1 = T_2}$$

$$\frac{T_1 : \texttt{ContrAddress}(U_1 * ... * U_n) \quad T_2 : \texttt{ContrAddress}(M_1 * ... * M_n) \quad U_1 = M_1 ... U_n = M_n}{T_1 = T_2}$$

C.5. TypeChecking Builtins

Scilla's builtin functions such as `eq`, `pow`, `concat`, `to_bystr` and so on what implemented internally. In other words, there is not text for these functions. Thus, in order to type-check these functions, there are a few pieces of information that we must implement, namely the function's arity, what types it accepts (if the function is polymorphic), and the function's type. Here is an example of the information stored about the equality function `eq`:

```
export class BI_eq {
    constructor() {
        this.arity = 2;
        this.types = [new ST.String(), new ST.BNum()].
   concat(ST.allInts).concat(ST.allUints).concat(ST.
   allBystr).concat(ST.allAddr);
        this.funTyp =
            new ST.PolyFun("'A",
            new ST.FunType(new ST.TypeVar("'A"),
            new ST.FunType(new ST.TypeVar("'A"),
                new ST.ADT("Bool", [])));
    }
}
```

As we can see, the equality function has an arity of 2, accepts almost all types, and is a polymorphic function that takes two polymorphic parameters and return a boolean value. We then write a built in function dictionary `BuiltInDict` where we can access this information about each builtin function. Finally, each time a builtin function is evaluated, since type instantiation is done internally (and not by the user), we implement a function called `resolveBIFunType()` to resolve the type of the builtin function.

Depending on what is the builtin function, `resolveBIFunType()` will handle the function differently. For example, given a polymorphic builtin function with only one type abstraction, `resolveBIFunType()` will return a monomorphic function type according the to argument types given. On the other hand, `resolveBIFunType()` will handle a polymorphic map builtin function by instantiating the types of the map's keys and values. However, for all functions, there are a few basic checks that are done. We check whether the number of arguments passed to the function is equal to the arity of the function, as well as whether the type of parameters of the functions are allowed for this specific builtin function.

C.6. Abstract Data Types

In this section, we will describe how we implement abstract data types (ADTs) and their respective constructors to allow for type checking. ADTs are represented with the class `ScillaDataTypes` which contain the name `tname`, type variables `tparams`, constructors that belong to the ADT `tconstr`, and mapping from constructor to the constructor's argument types `tmap`. Additionally, each

constructor is represented by the class `Constructor` which contains the name of the constructor `cname` alone with its arity `arity`.

Let us work through an example of implementation of the `List` ADT.

```
export class ListDT extends ScillaDataTypes {
  constructor() {
    super();
    this.tname = "List";
    this.tparams = ["'A"];
    this.tconstr = [new Cons(), new Nil()];
    this.tmap = {
      Cons: [new ST.TypeVar("'A"),
             new ST.ADT("List", [new ST.TypeVar("'A")])
   ],
    };
  }
}

export class Cons extends Constructor {
  constructor() {
    super();
    this.cname = "Cons";
    this.arity = 2;
  }
}

export class Nil extends Constructor {
  constructor() {
    super();
    this.cname = "Nil";
    this.arity = 0;
  }
}
```

As shown above, the ADT `List` has a accepts one type variable `'A`. In other words, we can think of ADTs as polymorphic functions that contain type abstractions. Since `Nil` has an arity of 0, we do not include in the ADT's `tmap`. `Cons` constructor on the other hand has two arguments of type `'A` and `List 'A` respectively as described in `tmap`. Depending on what the ADT is instantiated with, we would know to instantiate the type of the constructor appropriately. This information is especially important for type checking pattern matching, as patterns access the arguments given to the constructors.

Finally, we build a dictionary `DataTypeDict` that contains a ADT dictionary as well as a constructor dictionary. Thus, when we need to type check constructors, we can easily access information `DataTypeDict`.

C.7. Polymorphism and Shadowing of Type Variables

Assume we have a polymorphic function of type $\forall$X.$\forall$Y.X $\rightarrow$ Y, if we instantiate the function with, say, Int64, we would get the type $\forall$Y.Int64 $\rightarrow$ Y. Implementation wise, we iterate through the type of the function and substitute any instance of type variable X with Int64.

The example above is valid. However, given a polymorphic function type $\forall$X.$\forall$X.X $\rightarrow$ X, instantiating with Int64 in the same manner would result in the type $\forall$X.Int64 $\rightarrow$ Int64.

Instead, at every evaluation of a type application, we make sure that type abstraction always introduce "fresh" type variables. In other words, at the point of applying type Int64, we would reconstruct $\forall$X.$\forall$X.X $\rightarrow$ X to $\forall$X.$\forall$X1.X1 $\rightarrow$ X1.

Although there exist other solutions, this solution was the easiest for us to generalise over other polymorphic components such as builtins and ADTs.

C.8. Testing the Type Checker

Running the file testingTC.js would type check all Scilla expression and Scilla contract files. Currently, the only programs failling are those that contain builtin primitives that have not been implemented, such as crypto-builtins.