Specification for the new Solidity Language

The Argot Collective

April 7, 2025

1 Introduction

In this report we describe the abstract syntax, the type system and desugaring steps for a new version of the new Solidity programming language.

2 Language overview

Before diving into the formalization details, it is useful to explain how the new language will look like and the steps which will be formally defined in this report. Let's consider the following simple code piece which defines three functions inside a contract:

```
contract Ex {
  function id () {
    return lam(x) {return x ;};
  }
  function compose(f,g) {
    return lam(x){ return f(g(x));};
  }
  function main () {
    let f = compose(id,id);
    return f(42);
  }
}
```

In this short example we can see some features of the new Solidity language: it will have anonymous functions and function arguments and return types can be omitted because the language will have type inference. The same code example can be written using type annotations as follows:

```
contract Ex {
  forall a . function id () -> a -> a {
    return lam(x) {return x ;};
  }
  forall a b c . function compose(f : b -> c, g : a -> b) : a -> c {
    return lam(x){ return f(g(x));};
  }
  function main1 () -> word {
    let f : a -> a = id();
    return f(42);
  }

  function main () -> word {
    let f : a -> a = compose(id,id);
    return f(42);
  }
}
```

While we can understand the typing of functions as in the previous code piece, the new Solidity compiler will produce an equivalent code using the primitive `Invokable` type class.

```
class self : Invokable (argsTy, retTy) {
  function invoke (a : self, args : argsTy) -> retTy ;
}
```

which call a function using the arguments *args* : *argsTy*. The compiler uses the `Invokable` class to deal with high-order functions. The strategy is to generate for each defined function a unique type and an instance of the `Invokable` class for it (using the generated *unique type* as the main type argument). Unique types always have two type arguments representing the argument and result types of the original function, and have just one constructor with no arguments. For the functions `id`, `compose` and `main`, the compiler will generate the following unique type definitions:

```
data t_id0(args,ret) = t_id0
data t_compose1 (args, ret) = t_compose1
data t_main2 (args, ret) = t_main2
```

Another transformation step is to perform closure conversion on anonymous functions. The function `id` is compiled to the following equivalent code snippet:

```
forall a . function id () -> t_lambda34(a,a) {
  return t_lambda34;
}
data t_lambda34(args, ret) = t_lambda34
forall a . instance t_id0((),t_lambda34(a,a)) : Invokable((),t_lambda34(a,a)) {
  function invoke (self : t_id((),t_lambda34(a,a)), x : ()) -> t_lambda34(a,a) {
    return id() ;
  }
}
forall a . function lambda3 (x : a) -> a {
  return x ;
}
forall a. instance t_lambda34(a,a) : Invokable (a, a) {
  function invoke (self : t_lambda34(a,a), x : a) -> a {
    return x;
  }
}
```

Since the λ-abstract present in `id` definition does not have captures, we can just lift its body into a new function and produce its corresponding unique type and `Invokable` instance. Notice, that `id`'s body now produces the constructor of its unique type which is used to guide instance search to find the correct to called by `Invokable.invoke`.

Next listing deals with the desugared version for function `compose`, which has free variables in its λ-expression. We then define a type for representing its closure and define a `Invokable` instance using the closure type as its main argument type. The resulting desugared code for compose is as follows:

```
// lam(x) { return f(g(x));}
data t_closure4(a,b,c,d,e) = t_closure4(d,e)
forall e : Invokable(b,c),
       d : Invokable(a,b) . t_closure4(a,b,c,d,e) : Invokable (a, c) {
  function lambda5(env : t_closure4(a,b,c,d,e), x : a) -> c {
    match env {
    | tclosure4(f,g) =>
      Invokable.invoke(f, Invokable.invoke(g, x));
    };
  }
}
forall a b c d e. function compose(f : d, g : e) -> t_closure4(a,b,c,d,e) {
  return t_closure4(f,g);
}
```

The last step of the desugared code is for the `main` function, which is presented next. Notice that the indirect call for function reference `f` is desugared to a call to `Invokable.invoke` in which `f` is passed as the argument of the called instance main type.

```
function main () -> word {
  let f = compose(t_id0, t_id0);
  return Invokable.invoke(f,42);
}
```

3 Notations used

We start by defining some notations used in the source language abstract syntax. As usual, all meta-variables can appear primed or subscripted. We let $[x]$ denote an optional occurrence of x and $\overline{x}$ a sequence of x's. Sometimes we use notation $x_1, ..., x_n$ to denote $\overline{x}$ and use Haskell list syntax to represent empty and head / tail sequences. We also allow ourselves a bit of informality by using set operations on sequences with their usual meaning.

Meta-variable	Meaning
Q	Qualified name
C	Type class name
X	Contract name
T	Type constructor name
D	Algebraic data type constructor name
f	Function name
I	EVM instruction
v	Variable name
w	Word literal

Table 1: Meta-variable usage.

In the following sections, we define inference rules to formalize typing and desugaring of source programs. The general form of inference judgements are

$$\mathit{environments} \overset{\text{judgment name}}{\vdash} \mathit{source} \rightsquigarrow \mathit{target} : \mathit{moreinfo}$$

where *environments* contain various contextual information, *target* is the translation result of *source* program and *moreinfo* is some additional result produce by the rule (e.g. if the *source* is an expression, *moreinfo* is its type). An environment is a finite map between names and some relevant information. We use different meta-variables to denote different types of environments. We allow ourselves a bit of informality and use set operations on finite maps with their obvious meanings. Additionally, we use the following notations over finite maps:

- Notation $v \mapsto \mathit{value}$ denotes a map entry with key v and associated *value* and $E[v \mapsto \mathit{value}]$ denotes the insert / update operation of key v for the map E.
- $\texttt{dom}(E_1) = \{\mathit{name} \,|\, \mathit{name} : \mathit{info} \in E_1\}$ defines the domain, i.e. set of names, defined on the environment.

- $E(v) = r$ denotes the finite map lookup operation.

$$E(v) \quad = \quad \begin{cases} value & \texttt{when}\ [v \mapsto value] \in E \\ \bot & \texttt{otherwise} \end{cases}$$

3.1 Contexts

Our formalization will use the following environments:

- Θ: global definition environment, which is formed by the following components:
  - Θ^U: environment associating function names to its respective unique data type definitions.
  - Θ^X: environment associating contract names to a environment containing its field names and its respective types.
  - Θ^{cls}: environment associating class names to the constraint associated with it. A class definition

    ```
    forall v1 ... vn . P => class vj : C(...) {
      ...
    }
    ```

    introduces the constraint $\forall \overline{\alpha}.P \Rightarrow \alpha_j : C(...)$ under the key C in Θ^{cls}.
  - Θ^{ins}: environment associating class names to the list of defined instances for this class. Each instance definition

    ```
    forall v1 ... vn . P => instance t : C(...) {
      ...
    }
    ```

    introduces the constraint $\forall \alpha.P \Rightarrow \tau : C(...)$ in the list of instances under the key C in Θ^{ins}.
- Γ: typing environment, which holds definitions of defined variables in scope, data constructors and function definitions. Following common practice, notation $\Gamma, x : \sigma$ denotes the inclusion of the assumption $x : \sigma$ in Γ.

4 Source language abstract syntax

$$
\begin{array}{rcl}
\mathit{mod} \in \text{Module} & \rightarrow & \overline{\mathit{imp}}\ ;\ \overline{d} \\
\mathit{imp} \in \text{Import} & \rightarrow & \texttt{import}\ Q; \\
d \in \text{Declaration} & \rightarrow & \mathit{classdecl} \mid \mathit{instdecl} \mid \mathit{datadecl} \mid \mathit{fundecl} \mid \mathit{contract} \\
\mathit{classdecl} \in \text{Class} & \rightarrow & \forall\ \overline{\alpha}\ .\ P \Rightarrow \texttt{class}\ \alpha : C(\overline{\alpha'})\ \overline{\mathit{sig}} \\
\mathit{instdecl} \in \text{Instance} & \rightarrow & \forall\ \overline{\alpha}\ .\ P \Rightarrow \texttt{instance}\ \tau : \mathrm{C}(\overline{\tau'})\ \overline{\mathit{fundecl}} \\
\mathit{datadecl} \in \text{Data Type} & \rightarrow & \texttt{data}\ T(\overline{\alpha})\ \texttt{=}\ \overline{\mathit{dc}} \\
\mathit{fundecl} \in \text{Function} & \rightarrow & \mathit{sig}\ \mathit{body} \\
\mathit{contract} \in \text{Contract} & \rightarrow & \texttt{contract}\ X(\overline{\alpha})\ \overline{\mathit{cdecl}} \\
P \in \text{Context} & \rightarrow & \overline{\pi} \\
\mathit{sig} \in \text{Signature} & \rightarrow & \forall\ \overline{\alpha}\ .\ P \Rightarrow \texttt{function}\ f\ (\overline{\mathit{arg}})\ [\rightarrow \tau] \\
\mathit{dc} \in \text{Data Constr.} & \rightarrow & D(\overline{\tau}) \\
\mathit{body} \in \text{Function body} & \rightarrow & \overline{s} \\
\mathit{cdecl} \in \text{Contract Decl.} & \rightarrow & \mathit{constr} \mid \mathit{fundecl} \mid \mathit{fielddecl} \\
\pi \in \text{Predicate} & \rightarrow & \tau : C(\overline{\tau'}) \\
\mathit{arg} \in \text{Argument} & \rightarrow & v[:\tau] \\
\mathit{constr} \in \text{Contract Constr.} & \rightarrow & \texttt{constructor}\ (\overline{\mathit{arg}})\ \mathit{body} \\
\mathit{fielddecl} \in \text{Field Decl.} & \rightarrow & \tau\ v\ [=\ e] \\
s \in \text{Statement} & \rightarrow & \texttt{let}\ [\tau]\ v\ [=\ e] \mid \mathit{lvalue} = e \mid e \mid \texttt{return}\ e \\
 & \mid & \mathit{match} \mid \texttt{assembly}\ \mathit{asm} \\
e \in \text{Expr.} & \rightarrow & w \mid v \mid D(\overline{e}) \mid e.v \mid \texttt{lam}(\overline{\mathit{arg}})\ \mathit{body} \\
 & \mid & f(\overline{e}) \mid e : \sigma \\
\mathit{match} \in \text{Match} & \rightarrow & \texttt{match}\ \overline{e}\ \texttt{with}\ \overline{\mathit{eqn}} \\
\mathit{asm} \in \text{Assembly} & \rightarrow & \{\overline{\mathit{ys}}\} \\
\mathit{eqn} \in \text{Equation} & \rightarrow & \overline{p} \Rightarrow \mathit{body} \\
p \in \text{Pattern} & \rightarrow & w \mid v \mid D(\overline{p}) \\
\mathit{ys} \in \text{Yul Stmt.} & \rightarrow & v = \mathit{ye} \mid \texttt{let}\ v\ [=\mathit{ye}] \mid \mathit{ye} \mid \{\overline{\mathit{ys}}\} \mid \texttt{if}\ e\ \mathit{asm} \\
 & \mid & \texttt{for}\ \mathit{asm}\ \mathit{yexp}\ \mathit{asm}\ \mathit{asm} \mid \texttt{switch}\ \mathit{ye}\ \{\overline{\mathit{case}}\ [\mathit{default}]\} \\
\mathit{ye} \in \text{Yul Exp.} & \rightarrow & w \mid v \mid I(\overline{\mathit{ye}}) \\
\mathit{case} \in \text{Case} & \rightarrow & w\ \texttt{=>}\ \mathit{asm} \\
\mathit{default} \in \text{Def. Case} & \rightarrow & \texttt{default}\ \mathit{asm}
\end{array}
$$

Figure 1: New Solidity surface abstract syntax

5 Type syntax

The type syntax is defined by Figure 2.

$$
\begin{array}{rcl}
\tau \in \text{Type} & \rightarrow & \alpha \mid T(\tau_1,...,\tau_n) \\
\rho \in \text{Qual. Type} & \rightarrow & P \Rightarrow \tau \\
\sigma \in \text{Scheme} & \rightarrow & \forall\ \overline{\alpha}\ .\ \rho
\end{array}
$$

Figure 2: Type syntax

Simple types (τ) are either type variables, represented by meta-variable α, or fully-applied type constructors[1]. We let meta-variable T denote an arbitrary type constructor and we assume the existence of the following primitive type constructors:

[1]We use this restriction on type constructors to avoid checking kinds.

- `word`: primitive type for UInt256 literals.
- `unit`: type which has a unique inhabitant. We denote both the unit type and its element by ().
- `arrow`: function type constructor, written as right associative infix operator as $\tau_1 \to \tau_2$. Sometimes we write n-ary arrows as $\overline{\tau}_a \to \tau_r$
- `tuples`: We assume the existence of n-ary tuples ($n \geq 2$), which are translated internally into right-nested pairs.

A qualified type (ρ) is formed by a set of type class predicates, denoted by P, and a simple type. We let meta-variable π be any predicate. Notation P, Q denote the union of the predicate sets, $P \cup Q$. A type scheme (σ) is qualified type which has a set of quantified type variables. Notation $\texttt{ftv}(\sigma)$ denotes the set of σ's free type variables. We slighly abuse notation by using `ftv` for typing context and environments with its obvious definition. We let $\texttt{fpv}(x)$ denote the set of x's free program variables. A *substitution* is a finite function associating type variables to simple types. We let meta-variable S denote an arbitrary substitution. Notation $[\overline{\alpha \mapsto \tau}]$ denotes the substitution that replaces type variables $\overline{\alpha}$ by the types $\overline{\tau}$, `id` denotes the identity substitution and $S_1 \circ S_2$ denotes the composition of S_1 and S_2. Application of a substition S to a type τ is denoted by $S\,\tau$ and application is extended to constraints and qualified types as expected. Given two types, τ_1 and τ_2, we let $\texttt{mgu}(\tau_1, \tau_2)$ denote the most general unifier for these types. We extend `mgu` to constraints and qualified types as usual. Finally, given a substitution S and a set of type variables V, notatation $S\,|_V$ denotes the restriction of S's domain to V as follows: $S\,|_V = \{[\alpha \mapsto \tau] \,|\, \alpha \in V\}$.

We let notation $\sigma \sqsubseteq \rho$ denote that ρ is an *instance* of type scheme σ, and we define this relation as:

$$\frac{}{\forall\,\overline{\alpha}\,.\,\rho \sqsubseteq [\overline{\alpha \mapsto \tau}]\rho}\ \{\textit{Inst}\}$$

Figure 3: Type instantiation relation.

Another relation over types used in our formalization is *subsumption* [5], written as $\sigma_{off} \leq \sigma_{req}$, which means that type σ_{off} is at least as polymorphic as σ_{req}. This relation is necessary to deal correctly with type annotations.

$$\frac{\overline{\alpha} \subseteq \texttt{ftv}(\sigma) \quad \sigma \leq \rho}{\sigma \leq \forall\,\overline{\alpha}.\rho}\ \{\textit{Skol}\} \qquad \frac{\overline{[\alpha \mapsto \tau]}\rho_1 \leq \rho_2}{\forall\,\overline{\alpha}\,.\,\rho_1 \leq \rho_2}\ \{\textit{Spec}\} \qquad \frac{}{\tau \leq \tau}\ \{\textit{Mono}\}$$

Figure 4: Subsumption judgement.

6 Ambiguity and type improvement

Type inference proceeds by collecting constraints generated by the use of defined function symbols and variables which are later solved. However, some constraints cannot be solved using defined instances due to have have what we call "unreachable type variables".

Definition 1 (Unreachable type variables)**.** Given a qualified type $P \Rightarrow \tau$, we define the set of unreachable variables as $\texttt{ftv}(P) - \texttt{ftv}(\tau)$, i.e. variables occuring in predicates which do not appear in τ.

Using this notion, we can define the notion of ambiguity in types.

Definition 2 (Ambiguity)**.** A type $P \Rightarrow \tau$ is ambiguous if it contains unreachable variables.

Haskell consider types containing unreachable variables as ambiguous and this was one of the main reasons behind the design of extensions like functional dependencies [4] and type families [1] which provides some additional constraint solving based on these additional specifications on class / instance definitions.

In this section, we review constraint entailment and reduction to define a type improvement strategy.

6.1 Constraint entailment

A central problem in type systems of qualified types is determining when a restriction present on a type holds. The notion of constraint entailment is formalized by a relationship between sets of type constraints. In this report, we consider a definition of provability similar to that presented by Mark Jones [3]. We say that a set of constraints Q is provable from another set of constraints P using the information contained in the context Θ if it is possible to prove $\Theta; P \Vdash Q$. When $P = \emptyset$, we use notation $\Theta \Vdash Q$.

$$\frac{Q \subseteq P}{\Theta; P \Vdash Q}\,\{Mono\} \qquad \frac{\Theta; P \Vdash Q' \quad \Theta; Q' \Vdash Q}{\Theta; P \Vdash Q}\,\{Trans\} \qquad \frac{\Theta; P \Vdash Q}{\Theta; S\,P \Vdash S\,Q}\,\{Subst\} \qquad \frac{\Theta; P \Vdash P' \quad \Theta; Q \Vdash Q'}{\Theta; P, Q \Vdash P', Q'}\,\{Conj\}$$

$$\frac{\forall\overline{\alpha}.P \Rightarrow \tau : C(\overline{\tau'}) \in \Theta^{ins}(C)}{\Theta; P \Vdash \tau : C(\overline{\tau'})}\,\{Inst\}$$

$$\frac{\Theta; P \Vdash Q' \quad \pi \in Q' \quad \pi = \alpha_1 : C(\overline{\alpha'_1}) \quad \forall\overline{\alpha}.P \Rightarrow \alpha : C(\overline{\alpha}) \in \Theta^{cls}(C)}{\Theta; P \Vdash \pi}\,\{Super\}$$

Figure 5: Constraint entailment.

The three first rules shows the requirements for an entailment relation: monotonicity, transitivity and closure under substitution [3]. Rule *Conj* conjuncts two sets of constraints. Rule *Inst* specifies that a constraint $\tau : C(\overline{\tau'})$ is entailed by a set P if there's an instance $\forall\overline{\alpha}.P \Rightarrow \tau : C(\overline{\tau'})$. Finally, rule *Super* allows the entailment of a predicate by using super class information.

Example 1. As an example an entailment proof, let's consider the following class and instance constraints:

- $\Theta^{ins} = \{\texttt{word} : \texttt{Eq}, \forall\alpha.\{\alpha : \texttt{Eq}\} \Rightarrow \texttt{Foo}(\alpha) : \texttt{Eq}\}$.
- $\Theta^{cls} = \{\forall\alpha.\alpha : \texttt{Eq}\}$

Using rules in Figure 5 we can show that $\Theta \Vdash \texttt{Foo}(\texttt{word}) : \texttt{Eq}$ by the following derivation:

$$\dfrac{\dfrac{\texttt{word} : \texttt{Eq} \in \Theta^{ins}(\texttt{Eq})}{\Theta \Vdash \texttt{word} : \texttt{Eq}}\,\{\texttt{Inst}\} \qquad \dfrac{\dfrac{\forall\,\alpha.\,\alpha : \texttt{Eq} \Rightarrow \texttt{Foo}(\alpha) : \texttt{Eq} \in \Theta^{ins}(\texttt{Eq})}{\Theta,\,\alpha : \texttt{Eq} \Vdash \texttt{Foo}(\alpha) : \texttt{Eq}}\,\{\texttt{Inst}\} \qquad S = [\alpha \mapsto \texttt{Int}]}{\Theta,\,\texttt{word} : \texttt{Eq} \Vdash \texttt{Foo}(\texttt{word}) : \texttt{Eq}}\,\{\texttt{Subst}\}}{\Theta \Vdash \texttt{Foo}(\texttt{word}) : \texttt{Eq}}\,\{\texttt{Trans}\}$$

□

6.2 Context reduction

We name *context reduction* the process of transforming each constraint π to a new context P introduced by the instance that has a head which matches with π. The reducing process continues until either $P = \emptyset$ or there's no further matching instances. Similarly to constraint satisfiability, we define a function to get information of about matching instances for a given input constraint.

$$\begin{array}{rcl}\texttt{matches}(\pi, \Theta) & = & \{(S\,P, \pi') \mid (S, P, \pi') \in \texttt{sat}(\Theta, [\overline{\alpha \mapsto K}]\,\pi)\} \\ & & \texttt{where} \\ & & \quad \overline{\alpha} = \texttt{ftv}(\pi) \\ & & \quad \overline{K}\ \texttt{are new Skolem constants}\end{array}$$

Function `sat` returns the information about instances that satisfies a constraint π: it selects instances which match the constraint main type argument. After determining a matching instance, the `sat` function unifies the constraint weak type arguments with the matching instance head corresponding arguments and then return the substitution together with the instance head, and remaining constraints to be satisfied.

$$\texttt{sat}(\Theta, \tau_1 : C(\overline{\tau})) \quad = \quad \left\{ (S' \circ S, P, \pi_0) \;\middle|\; \begin{array}{l} \forall\overline{\alpha}.P_0 \Rightarrow \pi_0 \in \Theta^{ins}(C) \\ S_1 = [\overline{\alpha \mapsto \beta}],\ \overline{\beta}\,\texttt{fresh} \\ P \Rightarrow \tau_1' : C(\overline{\tau'}) = S_1(P_0 \Rightarrow \pi_0) \\ S = \texttt{mgu}(\tau_1, \tau_1') \quad S' = \texttt{mgu}(S\,\overline{\tau}, S\,\overline{\tau'}) \end{array} \right\}$$

If we restrict ourselves to non-overlapping instances, `matches` always return a singleton or empty set for a given constraint. The context reduction for a given context $Q = \{\pi_1, ..., \pi_n\}$ is defined as:

$$\frac{\forall i.1 \leq i \leq n \to Q_i = \left\{ \begin{array}{ll} \pi_i & \Theta; n_0 \vdash^{\texttt{simp}} \pi_i \rightsquigarrow \bot \\ Q_i' & \Theta; n_0 \vdash^{\texttt{simp}} \pi_i \rightsquigarrow Q_i' \end{array} \right.}{\Theta \vdash^{\texttt{red}} \{\pi_1, ..., \pi_n\} \rightsquigarrow Q_1, ..., Q_n}\ \{Red\}$$

Figure 6: Context reduction.

Value n_0 denotes the initial "fuel" counter used to ensure termination of the reduction algorithm.

For each constraint in the input context, we use $\vdash^{\texttt{simp}}$ to simplify it, which could result in either in a failure (denoted by $\bot$) or in a new context Q'. Note that the only reason for failures in simplification is when its step counter reaches zero.

$$\frac{n > 0}{\Theta; n \vdash^{\texttt{simp}} \emptyset \rightsquigarrow \emptyset}\ \{\mathit{REmpty}\} \qquad \frac{}{\Theta; 0 \vdash^{\texttt{simp}} Q \rightsquigarrow \bot}\ \{\mathit{RStop0}\}$$

$$\frac{\begin{array}{l} n > 0 \\ \texttt{matches}(\pi, \Theta) = \emptyset \end{array}}{\Theta; n \vdash^{\texttt{simp}} \pi \rightsquigarrow \pi}\ \{\mathit{RStop}\} \qquad \frac{\begin{array}{l} n > 0 \\ \Theta; n-1 \vdash^{\texttt{simp}} \pi \rightsquigarrow P \\ \Theta; n-1 \vdash^{\texttt{simp}} Q \rightsquigarrow Q' \end{array}}{\Theta; n \vdash^{\texttt{simp}} \pi, Q \rightsquigarrow P, Q'}\ \{\mathit{RConj1}\}$$

$$\frac{\begin{array}{l} n > 0 \\ \Theta; n-1 \vdash^{\texttt{simp}} \pi \rightsquigarrow \bot \end{array}}{\Theta; n \vdash^{\texttt{simp}} \pi, Q \rightsquigarrow \bot}\ \{\mathit{RConj2}\} \qquad \frac{\begin{array}{l} n > 0 \\ \Theta; n-1 \vdash^{\texttt{simp}} \pi \rightsquigarrow P \\ \Theta; n-1 \vdash^{\texttt{simp}} Q \rightsquigarrow \bot \end{array}}{\Theta; n \vdash^{\texttt{simp}} \pi, Q \rightsquigarrow \bot}\ \{\mathit{RConj3}\}$$

$$\frac{\begin{array}{l} n > 0 \\ \{(P, \pi')\} = \texttt{matches}(\pi, \Theta) \\ \Theta; n-1 \vdash^{\texttt{simp}} P \rightsquigarrow Q \end{array}}{\Theta; n \vdash^{\texttt{simp}} \pi \rightsquigarrow Q}\ \{\mathit{RInst1}\} \qquad \frac{\begin{array}{l} n > 0 \\ \{(P, \pi')\} = \texttt{matches}(\pi, \Theta) \\ \Theta; n-1 \vdash^{\texttt{simp}} P \rightsquigarrow \bot \end{array}}{\Theta; n \vdash^{\texttt{simp}} \pi \rightsquigarrow \bot}\ \{\mathit{RInst2}\}$$

Figure 7: Reducing constraints.

6.3 Type improvement

Type improvement can be seen as a constraint simplification process which aims to eliminate ambiguities and infer more precise types. We follow [3] and use the concept of set of satisfying instances as a base for type improvement. Given a constraint set P, we define the set of satisfying instances for P as $\lfloor P \rfloor = \{S\,P \mid \Theta \Vdash S\,P\}$. If $\Theta \Vdash S\,P$ holds, we say that the substitution S satifies P in Θ. For any S, we have that $\lfloor S\,P \rfloor \subseteq \lfloor P \rfloor$. However, the inclusion $\lfloor P \rfloor \subseteq \lfloor S\,P \rfloor$ does not always hold but when it is true, we can specialize P to $S\,P$, which is a possibly simpler constraint set.

$$\frac{\begin{array}{cc} \sigma = \forall \overline{\alpha}.P \Rightarrow \tau & \Theta \vdash^{\texttt{red}} P \rightsquigarrow Q \\ \overline{\alpha}_1 = \texttt{ftv}(Q \Rightarrow \tau) & \sigma' = \forall \overline{\alpha}_1.Q \Rightarrow \tau \end{array}}{\Theta \vdash^{\texttt{impr}} \sigma \rhd \sigma'}\ \{\texttt{impr}\}$$

Figure 8: Type improvement.

7 Type system definition

In this section, we describe the typing rules for the new Solidity language.

7.1 Typing rules for expressions

We define expression typing by the following judgement

$$\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e_s \rightsquigarrow \langle e_t : \rho, \Theta', \Gamma' \rangle$$

where Θ denotes the global definition environment which holds information about class / instance declarations, data type and function definitions; and Γ is the typing environment, which holds type information for all variables, functions and data type constructor currently in scope. The expression judgement can modify the global definition and typing environments due to the desugaring of λ-expressions. When a rule does not change the environments, we use the following simplified version of the expression judgement:

$$\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e_s \rightsquigarrow e_t : \rho$$

We start presenting typing rules for basic expressions (Figure 9)

$$\frac{}{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} w \rightsquigarrow w : \texttt{word}}\{Lit\} \qquad \frac{\Gamma(v) = \sigma \quad \sigma \sqsubseteq \rho \quad \Theta_U(v) = \bot}{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} v \rightsquigarrow v : \rho}\{Var1\}$$

$$\frac{\begin{array}{l}\Theta_U(v) = \texttt{data}\, T(\alpha_{args}, \alpha_{ret}){=}D \\ \Gamma(D) = \sigma \qquad\qquad\qquad\quad \sigma \sqsubseteq \rho\end{array}}{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} v \rightsquigarrow D : \rho}\{Varf\}$$

$$\frac{\begin{array}{c}\Gamma(D) = \sigma \qquad \sigma \sqsubseteq \overline{\tau'} \to \tau \\ \Theta \mid \Gamma \overset{\text{args}}{\vdash} \overline{e} \rightsquigarrow \langle \overline{e'}, P, \overline{\tau'}, \Theta', \Gamma' \rangle\end{array}}{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} D(\overline{e}) \rightsquigarrow \langle D(\overline{e'}), P \Rightarrow \tau, \Theta', \Gamma' \rangle}\{Con\}$$

$$\frac{}{\Theta \mid \Gamma \overset{\text{args}}{\vdash} [] \rightsquigarrow \langle [], [], [], \Theta, \Gamma \rangle}\{ANil\} \qquad \frac{\begin{array}{c}\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow \langle e', Q, \tau, \Theta_1, \Gamma_1 \rangle \\ \Theta_1 \mid \Gamma_1 \overset{\text{args}}{\vdash} \overline{e}_s \rightsquigarrow \langle \overline{e'}_s, Q_s, \overline{\tau}_s, \Theta', \Gamma' \rangle\end{array}}{\Theta \mid \Gamma \overset{\text{args}}{\vdash} e \,:\, \overline{e}_s \rightsquigarrow \langle e' \,:\, \overline{e'}_s, Q \cup Q_s, \tau \,:\, \overline{\tau}_s, \Theta', \Gamma' \rangle}\{ACons\}$$

Figure 9: Expressions, part 01.

Rule $\{Lit\}$ specifies that literals have type `word`. For variables, we have two distinct cases: 1) for variables which are defined function references, which are desugared to the function unique type data constructor and 2) for normal variables which are desugared to themselves. Rule for data constructors just checks that each argument expression has the required argument type.

$$\frac{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow e' : T(\overline{\tau}_s) \quad \Theta_X(T) = \Gamma_f \quad \Gamma_f(v) = \tau}{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e.v \rightsquigarrow e'.v : \tau}\{FieldAcc\} \qquad \frac{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow e' : \sigma' \quad \sigma' \leq \sigma}{\Theta \mid \Gamma \overset{\text{exp}}{\vdash} e : \sigma \rightsquigarrow e' : \sigma}\{Ann\}$$

Figure 10: Expressions, part 02.

Rule $FieldAcc$ shows how to type and desugar a field access expression: we start by checking that expression e has a contract type $T(\overline{\tau}_s)$ and then we check that type T does have a field named v and return its type. We check type annotated expressions by infering its type, σ', and then check if it is as least as polymorphic as σ (Figure 4).

$$\frac{\begin{array}{c} f \in \texttt{dom}(\Theta_U) \qquad\qquad \Gamma(f) = \sigma \\ \sigma \sqsubseteq P \Rightarrow \overline{\tau}_s \to \tau \\ \Theta \,|\, \Gamma \overset{\text{args}}{\vdash} \overline{e} \rightsquigarrow \langle \overline{e'}, Q, \overline{\tau}, \Theta', \Gamma' \rangle \quad \rho = (P, Q) \Rightarrow \tau \end{array}}{\Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} f(\overline{e}) \rightsquigarrow \langle f(\overline{e'}), \rho, \Theta', \Gamma' \rangle}\ \{\mathit{CallD}\}$$

$$\frac{\begin{array}{c} f \notin \texttt{dom}(\Theta_U(f)) \qquad\qquad \Gamma(f) = T(\overline{\alpha}) \\ \Theta \,|\, \Gamma \overset{\text{args}}{\vdash} \overline{e} \rightsquigarrow \langle \overline{e'}, P, \overline{\tau'}, \Theta', \Gamma' \rangle \\ \pi = T(\tau_1, ..., \tau_n) \ : \ \texttt{Invokable}(\tau_1, ..., \tau_n) \quad \rho = (\{\pi\} \cup P) \Rightarrow \tau_{ret} \end{array}}{\Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} f(\overline{e}) \rightsquigarrow \langle \texttt{Invokable.invoke}(f, \overline{e'}), \rho, \Theta', \Gamma' \rangle}\ \{\mathit{CallI}\}$$

Figure 11: Expressions, part 03.

Next, we present the rules for dealing with function calls. The first rule deals with direct calls, which are typed as expected: first we get function type and instantiate it and check if its parameter types matches the arguments types. Finally, we say that the call type is the qualified type formed by set of predicates of arguments and the function type itself together with the function return's type. The desugaring of a direct call is just the same call with its desugared arguments. Typing indirect calls follows the same strategy for direct ones. The main difference is that it desugars to a call to `Invokable.invoke` function.

Now, we turn our attention to how to deal with λ-expressions: we type the body using as additional assumptions the λ-expression arguments with its types. Next, we apply a closure conversion step over the λ's type and body to produce:

- A value of the type representing the abstraction, which can be either a closure or the constructor of the unique type for the λ's generated function.
- Modified environments containing both the λ's generated function, unique type and `Invokable` instances.

$$\frac{\begin{array}{c} \Theta \,|\, \Gamma, \overline{x : \tau} \overset{\text{body}}{\vdash} \mathit{body} \rightsquigarrow \langle \mathit{body}', P' \Rightarrow \tau', \Theta_1, \Gamma_1 \rangle \\ \Theta; \Gamma; (P' \Rightarrow \overline{x : \tau} \to \tau'); \mathit{body}' \overset{cc}{\rightsquigarrow} \langle D, \rho, \Theta', \Gamma' \rangle \end{array}}{\Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} \texttt{lam}\,(\overline{x : \tau})\ \mathit{body} \rightsquigarrow \langle D, \rho, \Theta', \Gamma' \rangle}\ \{\mathit{Lam}\}$$

Figure 12: Expressions, part 04.

7.2 Typing rules for statements

We define statement typing by the following judgement

$$\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} s \rightsquigarrow \langle s', P, \tau, \Theta', \Gamma' \rangle$$

where Θ denotes the global definition environment which holds information about class / instance declarations, data type and function definitions; and Γ is the typing environment, which holds type information for all variables, functions and data type constructor currently in scope.

$$\frac{\Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow \langle e', P, \tau, \Theta', \Gamma_1 \rangle \quad \Gamma' = \Gamma_1, v : \tau}{\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} \texttt{let}\,\tau\, v \ = \ e \rightsquigarrow \langle \texttt{let}\,\tau\, v \ = \ e', P, \texttt{unit}, \Theta', \Gamma' \rangle}\ \{SLet\}$$

$$\frac{\Gamma(v) = \tau \quad \Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow \langle e', P, \tau, \Theta', \Gamma' \rangle}{\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} v \ = \ e \rightsquigarrow \langle v \ = \ e', P, \texttt{unit}, \Theta', \Gamma' \rangle}\ \{SAssign\}$$

$$\frac{\Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow \langle e', P, \tau, \Theta', \Gamma' \rangle}{\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} e \rightsquigarrow \langle e', P, \texttt{unit}, \Theta', \Gamma' \rangle}\ \{SExp\}$$

$$\frac{\Theta \,|\, \Gamma \overset{\text{exp}}{\vdash} e \rightsquigarrow \langle e', P, \tau, \Theta', \Gamma' \rangle}{\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} \texttt{return}\, e \rightsquigarrow \langle \texttt{return}\, e', P, \tau, \Theta', \Gamma' \rangle}\ \{SRet\}$$

Figure 13: Statements, part 01

The first set of typing rule for statements are immediate: typing the subexpressions and return the modified environments. The interesting cases of statement typing are matching and assembly blocks. Typing a match statement consist of typing the discriminee expressions and then use its type to check each equation. Typing assembly blocks consists in typing the assembly code and returning its newly defined names into context.

$$\frac{\begin{array}{c}\Theta \,|\, \Gamma \overset{\text{args}}{\vdash} \overline{e} \rightsquigarrow \langle \overline{e}_s, Q, \overline{\tau}_s, \Theta_1, \Gamma_1 \rangle \\ \Theta_1 \,|\, \Gamma_1; \overline{\tau}_s \overset{\text{eqns}}{\vdash} \overline{eqn} \rightsquigarrow \langle \overline{eqn}_s, Q', \Theta', \Gamma' \rangle\end{array}}{\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} \texttt{match}\, \overline{e}\, \texttt{with}\, \overline{eqn} \rightsquigarrow \langle \texttt{match}\, \overline{e}_s\, \texttt{with}\, \overline{eqn}_s, (Q, Q'), \texttt{unit}, \Theta', \Gamma' \rangle}\ \{SMatch\}$$

$$\frac{\Theta \,|\, \Gamma \overset{\text{asm}}{\vdash} \overline{ys} \rightsquigarrow \langle \overline{ys}_s, \Gamma_n \rangle}{\Theta \,|\, \Gamma \overset{\text{stmt}}{\vdash} \texttt{assembly}\, \overline{ys} \rightsquigarrow \langle \texttt{assembly}\, \overline{ys}_s, [], \texttt{unit}, \Theta, \Gamma \cup \Gamma_n \rangle}\ \{SAsm\}$$

Figure 14: Statements, part 02

Blocks of statements are typed by recursing over each statement. The unique exception is on variable declarations which we need to remove after typing the whole block.

$$\dfrac{}{\Theta \mid \Gamma \overset{\text{body}}{\vdash} [] \rightsquigarrow \langle [], [], \Theta, \Gamma \rangle}\{BNil\} \qquad \dfrac{\begin{array}{c}\Theta \mid \Gamma \overset{\text{stmt}}{\vdash} \tau\, v = e \rightsquigarrow \langle s', P, \tau, \Theta_1, \Gamma_1 \rangle \\ \Theta_1 \mid \Gamma_1, v : \tau \overset{\text{body}}{\vdash} \overline{s} \rightsquigarrow \langle \overline{s}', P', \tau', \Theta', \Gamma' \rangle\end{array}}{\Theta \mid \Gamma \overset{\text{body}}{\vdash} \tau v = e : \overline{s} \rightsquigarrow \langle s' : \overline{s}', (P \cup P'), \Theta', \Gamma' - \{v : \tau\} \rangle}\{BLet\}$$

$$\dfrac{\begin{array}{c}\Theta \mid \Gamma \overset{\text{stmt}}{\vdash} s \rightsquigarrow \langle s', P, \tau, \Theta_1, \Gamma_1 \rangle \\ \Theta_1 \mid \Gamma_1 \overset{\text{body}}{\vdash} \overline{s} \rightsquigarrow \langle \overline{s}', P', \tau', \Theta', \Gamma' \rangle\end{array}}{\Theta \mid \Gamma \overset{\text{body}}{\vdash} s : \overline{s} \rightsquigarrow \langle s' : \overline{s}', (P \cup P'), \Theta', \Gamma' \rangle}\{BCons\}$$

Figure 15: Typing blocks

Typing equations takes as input the type of the discriminees used to check patterns.

$$\dfrac{}{\Theta \mid \Gamma; \overline{\tau} \overset{\text{eqns}}{\vdash} [] \rightsquigarrow \langle [], [], \Theta, \Gamma \rangle}\{EQNil\} \qquad \dfrac{\begin{array}{c}\Theta \mid \Gamma; \overline{\tau} \overset{\text{eqn}}{\vdash} eqn \rightsquigarrow \langle eqn', Q, \Theta_1, \Gamma_1 \rangle \\ \Theta_1 \mid \Gamma_1; \overline{\tau} \overset{\text{eqns}}{\vdash} \overline{eqn} \rightsquigarrow \langle \overline{eqn}', Q', \Theta', \Gamma' \rangle\end{array}}{\Theta \mid \Gamma; \overline{\tau} \overset{\text{eqns}}{\vdash} eqn : \overline{eqn} \rightsquigarrow \langle eqn' : \overline{eqn}', (Q, Q'), \Theta', \Gamma' \rangle}\{EQCons\}$$

$$\dfrac{\Theta \mid \Gamma; \overline{\tau} \overset{\text{pats}}{\vdash} \overline{p} \rightsquigarrow \langle \overline{p}', \Gamma_s \rangle \quad \Theta \mid \Gamma \cup \Gamma_s \overset{\text{body}}{\vdash} \overline{s} \rightsquigarrow \langle \overline{s}', P, \tau, \Theta', \Gamma' \rangle}{\Theta \mid \Gamma; \overline{\tau} \overset{\text{eqn}}{\vdash} \overline{p} \Rightarrow \overline{s} \rightsquigarrow \langle \overline{p}' \Rightarrow \overline{s}', P, \Theta', \Gamma' - \Gamma_s \rangle}\{Eqn\}$$

Figure 16: Typing equations

Rules to typing patterns just produce the extra assumptions to be added at typing context to check the body of a pattern matching case.

$$\dfrac{}{\Theta \mid \Gamma; [] \overset{\text{pats}}{\vdash} [] \rightsquigarrow []}\{Pnil\} \qquad \dfrac{\Theta \mid \Gamma; \tau \overset{\text{pat}}{\vdash} p \rightsquigarrow \Gamma_1 \quad \Theta \mid \Gamma; \overline{\tau}_s \overset{\text{pats}}{\vdash} \overline{p}_s \rightsquigarrow \langle \overline{p}'_s, \Gamma_s \rangle}{\Theta \mid \Gamma; \tau : \overline{\tau}_s \overset{\text{pats}}{\vdash} p : \overline{p}_s \rightsquigarrow \langle p' : \overline{p}'_s, \Gamma_1 \cup \Gamma_s \rangle}\{Pcons\}$$

$$\dfrac{}{\Theta \mid \Gamma; \texttt{word} \overset{\text{pat}}{\vdash} w \rightsquigarrow \langle w, [] \rangle}\{PLit\} \qquad \dfrac{}{\Theta \mid \Gamma; \tau \overset{\text{pat}}{\vdash} v \rightsquigarrow \langle v, [v : \tau] \rangle}\{PVar\}$$

$$\dfrac{\Gamma(D) = \overline{\tau}_b \to T(\overline{\tau}_a) \quad \Theta \mid \Gamma; \overline{\tau}_b \overset{\text{pats}}{\vdash} \overline{p} \rightsquigarrow \langle \overline{p}', \Gamma' \rangle}{\Theta \mid \Gamma; T\,\overline{\tau}_a \overset{\text{pat}}{\vdash} D(\overline{p}) \rightsquigarrow \langle D(\overline{p}'), \Gamma' \rangle}\{PCon\}$$

Figure 17: Typing patterns

7.3 Typing Yul blocks

We start by defining typing rules for Yul statements. Assignments are typed as follows: We need to check if the variable v has type `word` in context Γ and assignment right-hand side, ye, also has type `word`. Typing variable definitions follow a similar strategy.

$$\frac{\Gamma(v) = \texttt{word} \quad \Theta \,|\, \Gamma \overset{\text{yexp}}{\vdash} ye \rightsquigarrow \langle ye', \texttt{word}, \Theta, \Gamma\rangle}{\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} v = ye \rightsquigarrow \langle v = ye', \Gamma\rangle}\ \{YAssign\}$$

$$\frac{\Theta \,|\, \Gamma \overset{\text{yexp}}{\vdash} ye \rightsquigarrow \langle ye', \texttt{word}, \Theta, \Gamma\rangle \quad \Gamma' = \Gamma, v : \texttt{word}}{\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} \texttt{let}\, v = ye \rightsquigarrow \langle \texttt{let}\, v = ye', \texttt{unit}, \Theta, \Gamma'\rangle}\ \{YLet\}$$

$$\frac{\Theta \,|\, \Gamma \overset{\text{yexp}}{\vdash} ye \rightsquigarrow \langle ye', \tau, \Theta, \Gamma\rangle}{\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} ye \rightsquigarrow \langle ye', \texttt{unit}, \Theta, \Gamma\rangle}\ \{YExp\}$$

Figure 18: Yul Statements, part 01.

Next, we consider rules for Yul block statments: if and for.

$$\frac{\Theta \,|\, \Gamma \overset{\text{yexp}}{\vdash} ye \rightsquigarrow \langle ye', \texttt{word}, \Theta, \Gamma\rangle \quad \Theta \,|\, \Gamma \overset{\text{asm}}{\vdash} \overline{ys} \rightsquigarrow \langle \overline{ys}', \Theta, \Gamma'\rangle}{\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} \textbf{if}\, ye\, \overline{ys} \rightsquigarrow \langle \textbf{if}\, ye'\, \overline{ys}', \texttt{unit}, \Theta, \Gamma\rangle}\ \{YIf\}$$

$$\frac{\begin{array}{c}\Theta \,|\, \Gamma \overset{\text{asm}}{\vdash} ys_1 \rightsquigarrow \langle ys_1', \Theta, \Gamma'\rangle \\ \Theta \,|\, \Gamma' \overset{\text{yexp}}{\vdash} ye \rightsquigarrow \langle ye', \texttt{word}, \Theta, \Gamma'\rangle \\ \Theta \,|\, \Gamma' \overset{\text{asm}}{\vdash} ys_2 \rightsquigarrow \langle ys_2', \Theta, \Gamma''\rangle \\ \Theta \,|\, \Gamma' \overset{\text{asm}}{\vdash} ys \rightsquigarrow \langle ys', \Theta, \Gamma_1\rangle\end{array}}{\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} \textbf{for}\, ys_1\, ye\, ys_2\, ys \rightsquigarrow \langle \textbf{for}\, ys_1'\, ye'\, ys_2'\, ys', \Theta, \Gamma\rangle}\ \{YFor\}$$

Figure 19: Yul Statements, part 02.

Blocks of Yul statements are similar to usual statement typing, as can be seen in the next figure.

$$\frac{}{\Theta \,|\, \Gamma \overset{\text{asm}}{\vdash} [] \rightsquigarrow \langle [], \Theta, \Gamma\rangle}\ \{YBNil\} \qquad \frac{\begin{array}{c}\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} \tau\, v = e \rightsquigarrow \langle s', \tau, \Theta_1, \Gamma_1\rangle \\ \Theta_1 \,|\, \Gamma_1, v : \tau \overset{\text{asm}}{\vdash} \overline{s} \rightsquigarrow \langle \overline{s}', \tau', \Theta', \Gamma'\rangle\end{array}}{\Theta \,|\, \Gamma \overset{\text{asm}}{\vdash} \tau v = e : \overline{s} \rightsquigarrow \langle s' : \overline{s}', \Theta', \Gamma' - \{v : \tau\}\rangle}\ \{YBLet\}$$

$$\frac{\begin{array}{c}\Theta \,|\, \Gamma \overset{\text{ystmt}}{\vdash} s \rightsquigarrow \langle s', \tau, \Theta_1, \Gamma_1\rangle \\ \Theta_1 \,|\, \Gamma_1 \overset{\text{asm}}{\vdash} \overline{s} \rightsquigarrow \langle \overline{s}', \tau', \Theta', \Gamma'\rangle\end{array}}{\Theta \,|\, \Gamma \overset{\text{asm}}{\vdash} s : \overline{s} \rightsquigarrow \langle s' : \overline{s}', \Theta', \Gamma'\rangle}\ \{YBCons\}$$

Figure 20: Typing ASM blocks

7.4 Typing rules for declarations

Typing rules for declarations is defined by the following judgement

$$\Theta \mid \Gamma \overset{\text{decl}}{\vdash} d \rightsquigarrow \langle \Theta', \Gamma' \rangle$$

which produces new environments from the input declaration and environments. We start by defining the rule for new data type definitions.

$$\frac{}{\Theta \mid \Gamma ; T(\overline{\alpha}) \overset{\text{cons}}{\vdash} [] \rightsquigarrow \langle \Theta, \Gamma \rangle}\{DNil\} \qquad \frac{\begin{array}{c}\Gamma_1 = \Gamma, D : \forall \overline{\alpha}.\overline{\tau} \rightarrow T(\overline{\alpha}) \\ \Theta \mid \Gamma_1 \overset{\text{cons}}{\vdash} \overline{dc} \rightsquigarrow \langle \Theta, \Gamma' \rangle\end{array}}{\Theta \mid \Gamma ; T(\overline{\alpha}) \overset{\text{cons}}{\vdash} D(\overline{\tau}) : \overline{dc} \rightsquigarrow \langle \Theta, \Gamma' \rangle}\{DCons\}$$

$$\frac{\Theta \mid \Gamma ; T(\overline{\alpha}) \overset{\text{cons}}{\vdash} \overline{dc} \rightsquigarrow \langle \Theta, \Gamma' \rangle}{\Theta \mid \Gamma \overset{\text{data}}{\vdash} \texttt{data}\, T(\overline{\alpha}) = \overline{dc} \rightsquigarrow \langle \Theta, \Gamma' \rangle}\{Data\}$$

Figure 21: Typing data declarations.

The main work of typing rules for data declarations is to populate the environments with data constructors types.

8 Desugaring

In this section we describe the desugaring steps performed by new Solidity implementation. Some of these steps are done during type inference and others are performed later using the elaborated syntax produced by the type inference engine. We start by describing the desugaring of λ-abstractions, generation of unique function types and `Invokable` instance generation.

8.1 Desugaring of λ-abstractions

The desugaring of abstractions is done by $\overset{cc}{\rightsquigarrow}$ predicate (Figure 22), which has two rules. The first rule, NC, deal with the abstractions which does not have free program variables (no closure needed): we simply generate its function declaration, corresponding unique type and `Invokable` instance. Each of these tasks are done by a specific rule, presented next.

$$\frac{\begin{array}{c}\texttt{fpv}(\mathit{body}) - \overline{x} = \emptyset \\ P \Rightarrow \overline{\tau} \rightarrow \tau_r \overset{sig}{\rightsquigarrow} \langle sig, f, \forall\overline{\alpha}.\rho\rangle \\ f; \forall\overline{\alpha}.\rho \overset{unique}{\rightsquigarrow} \texttt{data}\, T(\overline{\alpha}) = D \\ sig; \texttt{data}\, T(\overline{\alpha}) = D \overset{gen}{\rightsquigarrow} instd \\ \Gamma' = \Gamma, f : \forall\overline{\alpha}.\rho, D : \forall\overline{\alpha}.T(\overline{\alpha}) \\ \Theta' = \Theta[\texttt{Invokable} \mapsto^i instdecl]\end{array}}{\Theta; \Gamma; (P \Rightarrow \overline{x : \tau} \rightarrow \tau_r); \mathit{body} \overset{\text{cc}}{\rightsquigarrow} \langle D, \rho, \Theta', \Gamma'\rangle} \; \{NC\}$$

$$\frac{\begin{array}{c}\texttt{fpv}(\mathit{body}) - \texttt{dom}(\Gamma) = \{x'_1 : \tau_1\} \\ \texttt{fpv}(\mathit{body}) - \texttt{dom}(\Gamma) \overset{ctype}{\rightsquigarrow} \texttt{data}\, T(\alpha_1, ..., \alpha_n) = D_{clo}(\tau_1, ..., \tau_m) \\ \{f\} \cup \{x_1, ..., x_n\} \text{ are fresh names}\end{array}}{\Theta; \Gamma; (P \Rightarrow \tau_1 \times ... \times \tau_n \rightarrow \tau); \mathit{body} \overset{\text{cc}}{\rightsquigarrow} \langle D, \rho, \Theta', \Gamma'\rangle} \; \{CC\}$$

Figure 22: Desugaring of λ-expressions.

The generation of function signatures for λ-abstraction are as follows: first we generate the quantified type, σ, for λ expression. Using the set of argument types, we create fresh argument variable names to create the newly defined function arguments. The signature is produced using the infered predicates, a fresh function name, the set of arguments and its return type. Note that we demand that predicates produced should be entailed by current global context Θ.

$$\frac{\begin{array}{c}\overline{\alpha} = ftv(P \Rightarrow \tau_1 \times ...\tau_n \rightarrow \tau_r) - ftv(\Gamma) \\ \text{f is a fresh function name} \\ x_1, ..., x_n \text{ are fresh argument names} \\ \sigma = \forall\overline{\alpha}.P \Rightarrow \tau_1 \times ... \times \tau_n \rightarrow \tau_r \qquad \Theta; P \Vdash P'\end{array}}{P \Rightarrow \tau_1 \times ... \times \tau_n \rightarrow \tau_r \overset{sig}{\rightsquigarrow} \langle \forall\overline{\alpha}\, P'.\texttt{function} f(x_1 : \tau_1, ..., x_n : \tau_n) \rightarrow \tau_r, \sigma\rangle} \; \{Sig\}$$

Figure 23: Generation of λ-function signatures.

8.2 Generation of unique function types

Generation of unique data type for a function using its type information. Basically, the judgement $\overset{unique}{\rightsquigarrow}$ just creates fresh names for the type and data constructor. The generated type will use the type variables for the function infered type as its arguments.

$$\frac{f, T, D \text{ are fresh names}}{f; \forall\overline{\alpha}.\rho \overset{unique}{\rightsquigarrow} \texttt{data}\, T(\overline{\alpha}) = D} \; \{unique\}$$

Figure 24: Generation of unique types for functions.

8.3 Generation of `Invokable` instances

Next, we turn our attention to `Invokable` instance generation, using rule $\overset{gen}{\rightsquigarrow}$. The rule starts by generating the `Invokable.invoke` function body which is formed by a call to function f, that triggered the instance creation, using a pattern matching over the unique / closure type and a tuple of f's input arguments. The main type for the instance create is the unique / closure type and the weak arguments are just the tuple formed by f's argument and return types.

$$\frac{\begin{array}{c} \mathit{body} = \texttt{match}\,(\mathit{self}, \mathit{args})\,\texttt{with}\,(D(\mathit{pat}_1, ..., \mathit{pat}_m), (\mathit{pat}'_1, ..., \mathit{pat}'_n)) \Rightarrow \\ \texttt{return}\, f((\mathit{pat}_1, ..., \mathit{pat}_m), \mathit{pat}'_1, ..., \mathit{pat}'_n) \\ \mathit{fundecl} = \texttt{function Invokable.invoke}(\mathit{self} : T(\overline{\tau}), \mathit{args} : \tau_1 \times ... \times \tau_n) \rightarrow \tau_r\ \mathit{body} \\ \mathit{instd} = \forall \overline{\alpha} P.\texttt{instance}\, T(\overline{\tau}) : \texttt{Invokable}(\overline{\tau})\ \mathit{fundecl} \end{array}}{\forall\,\overline{\alpha}\,P.\texttt{function}\, f(x_1 : \tau_1, ..., x_n : \tau_n) \rightarrow \tau_r; \texttt{data}\, T(\overline{\alpha'}) = D(\overline{\tau}) \overset{gen}{\rightsquigarrow} \mathit{instd}}\ \{Gen\}$$

Figure 25: Generation of `Invokable` instances for functions.

9 Conclusions and Future Work

From our experiments we can conclude that [2].

References

[1] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 1–13, New York, NY, USA, 2005. Association for Computing Machinery.

[2] K.-F. Faxén. A static semantics for haskell. *J. Funct. Program.*, 12(5):295–357, July 2002.

[3] M. P. Jones. *Qualified types: theory and practice.* Cambridge University Press, USA, 1995.

[4] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, page 230–244, Berlin, Heidelberg, 2000. Springer-Verlag.

[5] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, Jan. 2007.