Formal Specification with Statecharts

Karthika Venkatesan, IIITB
Sujit Kumar Chakrabarti, IIITB

Abstract

UML Statecharts have enjoyed widespread popularity in software development community as a design and specification notation. It has primarily been employed for specification of object life-cycles. In this paper, we explore the option of using Statecharts in specifying GUI interactions, typically in web applications. A notation like Statechart is very apt to be used as a formal specification language, provided the following features: Firstly, it should have well-defined semantics, so that specifications written in it can not be inherently ambiguous. Secondly, it should have tool support for automatic detection of specification bugs. In this work, we present enhancements and customisations to the Statechart syntax and semantics for specification of modern web applications. We also demonstrate how automatic detection of specification bugs can be performed through well-known model-checking and program analysis. We have used our specification approach in specifying a number of web applications. Our approach enabled early detection of several specification bugs and hence facilitated higher quality specification.

Chapter 1

Specification Language

1.1 Introduction

The importance of formal specification is well-known. They come with the benefits of succinctness and unambiguous semantics, which makes them appropriate to act like formal contract between the user and maker of a software system. This also provides the added – and yet critical – benefit of automatic verification, wherein specification defects can be detected very early in the development life-cycle.

Adoption of formal languages and notations as de facto method for requirement specification has faced a number of hurdles. One of the main hurdles is the very mathematical nature of these languages: practising software engineers often consider these languages too hard to use. Though many of these specification languages are extremely powerful and have all the abstraction mechanisms to express the ideas of a domain, they need to be built up ground-up in the problem context using the language primitive, which is often considered too much of an effort.

Efforts to ameliorate these issues have been made primarily in two directions: *semi-formal languages* and *domain-specific languages*. Semi-formal languages, e.g. UML, come with user-friendly outlook which make them easy and usable. They make compromises on the formal semantics part. Domain specific languages (cite) give-up on the generality to gain on expressiveness in a given domain. The primitives in a DSL would also include the vocabulary of the domain to facilitate quick progress in the specification process.

Statechart was proposed a visual formalism for specification of complex systems (cite Harel). In particular, the 'system' here is an object with a complex life-cycle going through a finite number of discrete *states*. The notation has enjoyed a wide acceptance in software engineering community.

It is a part of the UML suite for modelling object oriented system. Statechart is a semi-formal language. This is the main hurdle in researchers developing automation support for verifying Statechart models. Consequently, there has been a lot of work done in defining its formal semantics (cite).

There have been attempts to use Statechart as a domain specific language (cite). An example of this is (cite Ricca-Tonella paper) in which application of Statecharts as a specification notation for web-based systems was explored. Features of web-based systems have evolved significantly since the time of this paper. Predominantly plain HTML forms have given place to highly interactive user-interfaces, e.g. tabbed panels, with rich client side validations as well as AJAX which permits partial page updates via asynchronous server interactions. This calls for a further look at how Statechart notation could be adapted to allow specification of richly featured modern web applications.

Another angle to look at is the growing complexity of web applications. This calls for increased levels of automation in engineering these systems, in particular, requirement analysis and verification. To achieve this goal, it is more important for specification formalisms to have a formally defined semantics, than to have rich syntactic features. A formal specification language would open doors to a variety of automated approaches to analysis, design, generation of both code and tests for modern web based systems.

In this paper, we introduce a specification language for GUI-intensive applications, in particular, web applications. This language takes inspiration from well-known imperative languages like C and Python (mutable variables, instructions, lexical scoping etc.), but borrows significantly some important aspect from Statecharts: *viz.*, states, transitions, hierarchy and action language. We present our language through examples as well as formal description of its abstract syntax and semantics. Through examples, we demonstrate how formal specifications for GUI applications can be written simply and intuitively using our language: the specification engineer uses her familiar programmatic constructs with little or no need to learn a very different looking language. The nitty-gritty implementation details can be conveniently abstracted away while maintaining the formal nature of the specification.

We also present a set of defects that are possible in a specification written in this language. We present an approach to detect a particular class of defects, namely *undefined variable access*, in the specifications. We use a well-known program analysis method (i.e. *reaching definition* [cite Dragon book] to detect undefined variable access. We present our experience in using our method to analyse the specifications we have developed. As it has turned out for us, our approach helped in early detection of many such defects, proving very helpful in creating a correct specification.

1.2 Statechart Language

Statechart provides a pictorial notation to describe the life-cycle of a system, usually object-oriented. It is an adaptation of the classical finite state automation with enhancement like hierarchical states, concurrent states, action language, communication etc., all aimed to increase its expressive power as a specification language of industrial strength software systems.

Statechart has been used widely in industry and academia as a part of various UML modelling tools [Rational Rose, Rhapsody, Telelogic, Statemate, Stateflow]. In all industrial tools, the explicitly specified semantics of the language, if at all given, are semi-formal to the best of our knowledge. There has been a lot of research in formal specification of semantics for Statechart(-like) languages [J. Rushby paper in FASE'04 on Stateflow operational semantics etc.]. They mostly target specific subsets of the Statechart language. We take a similar approach, defining formal syntax and semantics of the Statechart language for parts which we have found useful in formal specification of web applications. Mapping various existing features of the Statechart language to specification use-cases and formulating their formal description and building verification support for the same is a part of our ongoing research agenda.

1.3 Example

In this section, we informally introduce, through an example, the specific features that have been added or modified to the existing Statechart notation to enable better modelling of web applications.

1.3.1 Syntax

```
type statechart =
  state      *    (* initial state *)
  state list *    (* other states *)
  transition list (* transitions *)

type state =
    AtomicState of
      name *  (* State name *)
    | block * (* Entry actions *)
    | block   (* Exit actions *)
  | NonAtomicState of statechart
  | ShallowHistory
  | DeepHistory
  | Branch

type transition = Transition of
    id * (* source state name *)
```

```
    id * (* destination state name *)
    expr  * (* trigger *)
    expr  * (* guard  *)
    block   (* action *)

block = stmt list (* statements *)

stmt = Assignment of
        expr * (* LHS *)
        expr   (* RHS *)
  | Expr of expr (* independent expressions like
    arithmetic expressions or function calls *)
  | Return of expr (* Returned value *)
  | If of expr  * (* boolean condition *)
          block * (* then block *)
          block   (* else block *)
  | Loop of expr * (* boolean condition *)
            block  (* loop body *)

expr =
    BinaryExpr of expr   * (* LHS *)
                  binop * (* Operator *)
                  expr     (* RHS *)
  | UnaryExpr of unaryop * (* Unary operator *)
                 expr
  | FunctionCall of expr * (* Function *)
                    expr list (* actual parameters *)
  | Id of string
  | string_literal of string
  | num_const of string
  | PHI (* empty set *)
  | Set of expr list
```

Statechart specifications can be partitioned into their core Statechart part, and the trigger/guard/action part, typically abbreviated as the *action* part. The core Statechart part of the specification gives the pictorial form and finite-state nature to the specification. The action part, essentially comprised of code fragments in an appropriate imperative programming language, adds arbitrary expressiveness to the Statechart, allowing it to be infinite state in reality. In our adaptation, we closely follow the Statechart language as in Harel's notation, except – for the time being – postponing the treatment of some of its advanced features. However, we have provided more detailed specification of the action language part, some of them in variance with existing norms. These specifications have been accommodated to align the notation more closely to the needs of specifying GUI intensive applications in general, and web apps in particular.

The salient points of our syntax are:

1. *Set theoretic operations.*

2. *Local variables.*

3. *Input and output variables of states*

1.3.2 Semantics

Our action language has the following general semantic features.

1. *Lexical scope.* Events and variables declared in a state are visible within it, including the substates. Internal bindings shadow the external bindings.

2. *Types.* In terms of types, our language is a statically and explicitly typed language; and these should be considered the primary aspects of the type system followed by our language. As added features (subject to evolution), primitive types are provided to represent integers and booleans. Composites like lists, sets, tuples and maps are natively provided – all polymorphic (i.e. parameterised over types). Structures (as in C) are provided to create user defined types.

3. *Data-flow between states.* Action on a transition are allowed to use local events and variables of the source state and define any variables of the destination state. This feature allows data-flow between successively visited states without recourse to global variables.

In the following, we present a more formal description of the language semantics.

Code may occur in the following sites of the specification:

- States: Entry, During and Exit clauses

- Transition: Event, Guard and Action clauses.

The language is imperative, static typed and lexically scoped. The variables receive their types from *explicit type declarations* (or just *declarations*). Declarations can occur in staties, i.e. each state may have its own set of declarations.

Variable names appearing in any piece of code is bound to one and only one declaration occuring in an appropriate environment as dictated by the lexical scoping rules of the language.

1.3.3 Some Definitions

To formalise the semantics, we introduce the following definitions:

Definition 1 (Enclosing state of a state. $superstate : state \mapsto state$**)** *Given a state, returns the enclosing state if it exists. for the statechart (i.e. the outermost state), returns null.*

Definition 2 (Enclosing state of a transition. $state : transition \mapsto state$) *Given a transition, returns the enclosing state.*

Definition 3 (Source state. $src : transition \mapsto state$) *Given a transition, returns the source state.*

Definition 4 (Destination state. $dest : transition \mapsto state$) *Given a transition, returns the destination state.*

Definition 5 (Child state. $\prec_1$: $state \times state \mapsto boolean$) *Given two states s and s', $s \prec_1 s' = true$ if $s' = superstate(s)$. $s \prec_1 s' = false$ otherwise.*

Definition 6 (Proper descendent. $\prec$: $state \times state \mapsto boolean$) *Given two states s and s', $s \prec s' = true$ if for some integer $n \geq 0$, $\exists s_1, s_2, ..., s_n$ such that $s \prec_1 s_1$, $s_1 \prec_1 s_2$, ..., $s_{n-1} \prec_1 s_n$, $s_n \prec_1 s'$.*

$(s \prec_1 s') \Rightarrow (s \prec s)$. We may also be more specific and talk about the i-th descendent ($\prec_i$) with its intuitive meaning (e.g. $\prec_1$ is $\prec_i$ with $i = 1$). We also introduce the reflexive version of the descendent relation $\preceq$ (i.e. $s \preceq s'$ if $s = s' \vee s \prec s'$).

Definition 7 (Lowest upper bound (LUB). $\sqcap : state \times state \mapsto state$) *Given two states s_1 and s_2, a state s is defined as $s \sqcap s'$ (could be written as $\sqcap(s_1, s_2)$) iff: $(s_1 \preceq s \wedge s_2 \preceq s) \wedge$ (if $(\nexists s'.s' \neq s \wedge s_1 \preceq s' \wedge s_2 \preceq s')) \Rightarrow s \prec s'$*

In other words, the LUB of two states s_1 and s_2 is their common ancestor which is proper descendent of any other common ancestor of s_1 and s_2 if it exists.

Environment

All names are looked up in an environment. An environment can be visualised as a linked list of declaration lists, shown as:

$\sigma_1 = D_1 :: \sigma_2 = D_1 :: D_2 :: \sigma_3 = D_1 :: ... :: D_{n-1} :: \sigma_n = D_1 :: ... :: D_{n-1} :: D_n :: \phi$

where D_1, D_2, ... D_n are declaration lists, ϕ represents an empty declaration list, Operator '::' represents the list construction operator. For an environment $\sigma = D :: \sigma'$, we define $dec(\sigma) = D$ and $next(\sigma) = \sigma'$.

A lookup of a variable v in an environment σ, denoted by $\sigma[v]$, consists of searching for v's declaration in all the declaration lists in σ starting from the left end. v gets bound with the first matching declaration in this order.

Also, an environment σ is defined for a state s (σ^s) or a transition t (σ^t). States and transitions are referred to as *principle elements* for the rest of the discussion.

Finally, for an element, a number of environment types are defined as follows:

Definition 8 (Read environment σ_R) *A name which is* used *in an expression (i.e. used as r-value) is looked up in the read environment.*

Definition 9 (Write environment σ_W) *A name which is* defined *in an expression (i.e. used as l-value) is looked up in the write environment.*

Definition 10 (Read-Write environment σ_{RW}) *A name occuring in the read-write environment can be both used and defined in the context of a principle element.*

Definition 11 (Read-only environment σ_{RO}) *A name occuring in the read-only environment can be only used but not defined in the context of a principle element.*

Definition 12 (Write-only environment σ_{WO}) *A name occuring in the write-only environment can be only defined but not used in the context of a principle element.*

Directly, the following identities hold regarding the various environment types above:

1. $\sigma_R = \sigma_{RO} :: \sigma_{RW}$
2. $\sigma_W = \sigma_{WO} :: \sigma_{RW}$
3. $\sigma_{RO} \cap \sigma_{WO} = \phi$
4. $\sigma_R \cap \sigma_W = \sigma_{RW}$

In the above, we use intersection ($\cap$) in an overloaded sense by considering the environments as sets.

The various environments for the principle elements are defined as summarised in table 1.1. In this, we use $\setminus$ in an overloaded sense to indicate the exclusion of the 'subtracted' set from the tail.

	State s	**Transition** t
σ_R	$\sigma(s)$	$\sigma(src(t))$
σ_W	$\sigma(s)$	$\sigma(dest(t))$ [1]
σ_{RW}	$\sigma(s)$	$\sigma(state(t))$
σ_{RO}	ϕ	$\sigma(src(t)) \setminus \sigma(state(t))$
σ_{WO}	ϕ	$\sigma(dest(t)) \setminus \sigma(state(t))$

Table 1.1: Environments for states and transitions

1.3.4 Typing

Our language is statically typed, i.e. all type-checking is completed at compile time. A specification typechecks if at the end of the typechecking process, all expression type nodes in the AST receive a valid type. Typechecking of all assignment instructions is considered successful if the RHS expression typechecks to a type matching the declared type of the name on the LHS; the assignment itself does not receive any type in such a case. As an example, we present *tAssignAdd* below, the typing rule for an assignment with an addition on RHS.

$$tAssignAdd \quad \frac{\begin{array}{c}\sigma_R^t \vdash e_1 : int \\ \sigma_R^t \vdash e_2 : int \\ \sigma_W^t[v] = int\end{array}}{\sigma_R^t, \sigma_W^t \vdash v = e_1 + e_2 : None}$$

Rule 1 (Placement of Transition) *Given a transition* t, $state(t) = src(t) \sqcap dest(t)$

1.4 Specification Bugs

Like programs, requirement specifications are subject to faults [2]. One of the most important goals of using formal methods is to detect such bugs automatically. Requirement bugs can be further classified as *implicit* and *explicit*.

[2] For our discussion, we use *faults*, *defects* and *bugs* as synonyms.

Figure 1.1: Undefined variable

Implicit bugs are violations of universal properties. In programming, universal properties abound. Examples of violations of universal properties are accessing undefined variables, null-pointer dereferencing etc. Such bugs would most likely lead to catastrophic failures, like a program crash. Hence, modern compilers come with built-in support to detect many of these errors; programmers do not need to specify them explicitly. Such analysis is often based on static program analysis methods.

Explicit bugs are violations of properties explicitly stated by the programmer, often arising out of the business domain, often specific to the system being modelled. For example, for a banking software, the `balance` attribute of a `BankAccount` class may never be allowed to hold a negative value. Such properties need additional analysis typically based on model-checking or theorem proving technology.

We have encountered a number of requirement bugs that could occur in a Statechart specification model which could be classified as implicit and explicit, in a sense very similar to the above. And interestingly, it turns out that the verification technologies applicable in detection of these bugs also correspond to the above: program analysis for implicit bugs, and model checking and theorem proving for explicit bugs.

Further subsection of this section present some of the properties – both implicit and explicit – which could get violated in a specification leading to requirement bugs. In section 1.5, we explain how they can be detected using static analysis techniques.

1.4.1 Undefined Variable Access

Consider the Statechart fragment shown in figure 1.1. On running static analysis on this model, several warnings are flagged, e.g. `roomNumber` for a student may turn out to be `nil` in a number of expressions where it is used. At several places, this corresponds to a serious flaw in the specification model. Room allocation can be done in two ways: either by first selecting a student and then going to the room allocation, or by first selecting the room and then going to the student selection. Once this process is completed, the composite state `AllocateRoom` terminates. Here, it is possible to complete the room allocation without actually completing the process.

1.4.2 Non-Determinism

Statechart permits non-determinism. For instance, when multiple outgoing transitions from the current state are enabled simultaneously, the system must make a non-deterministic choice between them. Different flavours resolve non-determinism in different ways (cite various Statechart semantics papers).

Non-determinism is an instance of incompleteness in specification. Whether it is desired in the specification or not depends on the specific case. In any case, it would be desirable to be able to detect their presence automatically. For any configuration, the set of *conflicting outgoing transitions* is nothing but the set of the outgoing transitions from all its states. To statically detect non-determinism, we need to check if for any given configuration, any two conflicting outgoing transitions can get enabled simultaneously.

1.4.3 Stuck Specification

Another problem is that of *stuckness*: when there is no way left for the system to exit a particular configuration. Consider the simple Statechart fragement in figure 1.2

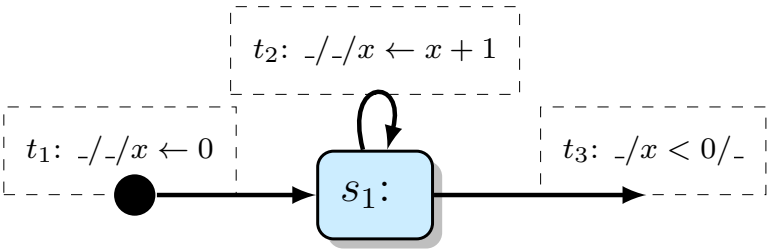


Figure 1.2: Stuck specification: The system can not exit s_1

1.4.4 Violation of External Specifications

Based on the specific domain, there may be any number of additional constraints which we may want our specification to follow.

1.5 Verification

In this section, we outline a number of analyses that can be used to detect the specification bugs presented in section 1.4.

1.6 Conclusion and Future Work

Concurrent states (And states) are a very important feature of Statechart notation which we have not considered so far. We realise that it is very relevant for specification of web applications which themselves have a large amount of concurrency built in. In the next stage of our research, we are working on incorporating concurrency into our adaptation of Statecharts. This feature – apart from adding expressiveness – will also make a number of additional specification bugs possible. Our next effort will be to augment our verification tool set for automatic detection of the above bugs.

Wherever there was a translation required for preparing an input to a verification tool, it was performed manually. This is tedious and error prone. There exists a wealth of prior work in this direction (cite Translation of Statechart to Model Checking). We are currently in the process of implementing some of these translators, so that the translation can be automated.

In this paper, we have focused on static verification techniques. However, Statechart specifications are an excellent starting point for automated test generation. In our earlier work, we have reported an end-to-end technique to generate test cases for web applications from Statechart specification (our ModSym 2016 paper). With addition of further features to the Statechart notation, it will be necessary to extend this test generation method to accommodate these.

Chapter 2

Design Notes

2.1 Types

We support a variety of *kinds* of types. There are the following kinds of types:

1. Basic types, e.g. int, boolean
2. Structures
3. ML style unions
4. Functions
5. Containers, e.g. lists, tuples, maps and sets For example:
   `registeredUsers : (string, User) map;`

Following points are to be noted about all the types in the language:

1. All basic types and containers are *native types*. All other types are *user defined types*.
2. All types are globally declared, whether native or user defined. That is, variables can be declared anywhere in the specification of all types.
3. Only containers are polymorphic. The varible declarations for such objects must substantiate at the time of declaration. For example:
   `registeredUsers : (string, User) map;`
4. Functions can't be declared independently. Rather, they are only added to support operations on container types,
   e.g. `requests.add( (loggedinUser, request_curfew.request) )` to add an element to a map `request`.

2.1.1 Implementation of Types

As mentioned above, all types are globally visible. They are implemented as elements of a type table in the `Statechart` class. All basic types are added to this during initialisation. As of now, we maintain the polymorphic types in a separate table because we don't intend to allow arbitrary mutual recursion between polymorphic types and non polymorphic types.

2.1.2 Structures

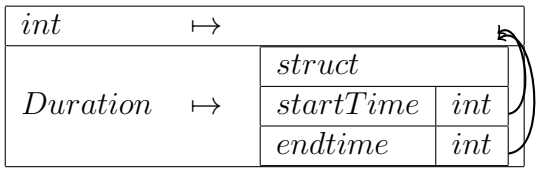
A `struct` type declaration will lead to a entry in the type table which will map a type name to a declaration list. For example:

```
struct Duration = {
  startTime : int;
  endTime : int
}
```

will create a type entry of the following form:

$$\textit{Duration} \quad \mapsto \quad \begin{array}{|c|c|}\hline \textit{struct} & \\ \hline \textit{startTime} & \textit{int} \\ \hline \textit{endtime} & \textit{int} \\ \hline \end{array}$$

The entry of a structure in the type table maps to a declaration list. All the type values appearing in this declaration list must map to already existing type entries in the type table which could be any – as of now non-polymorphic – types including structures themselves.



As of now, we don't allow forward references in the type table. We don't allow recursive types.

2.1.3 Functions

Functions map a list of input types to an output type. For example: $sum : fun(int \times int \rightarrow int)$ declares a function sum which takes two ints and returns an int.

As of now, we do not allow user defined function types. However, they are added to allow types like maps and lists with their properties declared as function types (see section 2.1.5).

2.1.4 Polymorphic Types

A polymorphic type can be thought of as a function (*type list* $\rightarrow$ *type*) which takes a list of types as parameters and returns a type. The type entry of a polymorphic type will have the following form:

$pair \quad \mapsto$

$polymorphic(\alpha, \beta)$:		
	$struct$	
	$first$	α
	$second$	β

A polymorphic type encapsulates a type expression inside it. The inner type expression is allowed to be any non-polymorphic type expression where type variables may take the place of types. For instance, above, $fun(\alpha \rightarrow None)$ is a function type which takes a type α as input and returns an *int* value.

As of now, we do not allow creation of new polymorphic types.

Substantiation of Polymorphic Types

When a variable is declared as a polymorphic, it's really a *substantiation* of a polymorphic type. By substantiation, we mean getting a concrete type from a polymorphic type by supplying the required type parameters. For example:

```
p : (int, boolean) pair;
```

will create a type entry from the polymorphic pair type:

$pair1 \quad \mapsto$

$struct$	
$first$	int
$second$	$boolean$

and, in the symbol table, bind `p` to $pair1$.

The semantic analyser must avoid duplicate substantiation by discovering structural equivalence between type declarations. For example, if another variable `q` is declared as:

```
q : (int, boolean) pair;
```

the semantic analyser should figure out that it has already created an entry $pair1$ in the type table which is structurally equivalent to this type, and bind `q` to $pair1$ rather than creating a new entry in the type table. This mayn't be a correctness issue, but may avoid proliferation of type entries in the type table.

2.1.5 Containers

Containers are really structures with functions being fields in them. For example, the entry for map type in the global type table will be something like:

$map \quad \mapsto$

$polymorphic(\alpha, \beta)$:	
$struct$	
add	$fun(\alpha \rightarrow None)$
$keys$	$fun(None \rightarrow \alpha\ set)$
get	$fun(\alpha \rightarrow \beta)$

In the above, *map* has been declared as a polymorphic type, which encapsulates a structure with three functions (*add*, *keys* and *get*). The function types have type variables as parameters, which bind to the formal parameters of the enclosing polymorphic type.

2.1.6 Constructs Involving Polymorphic Types

Declaration of Polymorphic Type

- **Atomic Types.**

  Examples:

```
map<| t1, t2 |>;
pair<| t1, t2 |>;
list<| t |>;
```

- **Composite Types.**

```
struct mypair<| t1, t2 |> {
  first  : t1;
  second : t2;
};
```

2.1.7 Declaration of Variables with Polymorphic Types

Variables can't be polymorphic, in the sense, they can be of non-polymorphic types created by substantiating a polymorphic type.

Examples:

```
map<| int, string |> rollnumbers;

mypair<| int, string |> student;
```

2.1.8 Declaration of Polymorphic Functions

Examples:

```
add_map<| t1, t2 |> (map<| t1, t2 |>, t1, t2) : map<| t1, t2 |>
```

2.1.9 Application of Polymorphic Functions

Application of polymorphic functions must be done using non-polymorphic types as type arguments to the function. The expressions sent as arguments to the function call should be of the same type as obtained of the formal parameters of the function when substantiated with the type arguments of the given function call.

Typing rule:

For a function call $f \triangleleft t_1, t_2, ..., t_k \triangleright (E_1, E_2, ..., E_n)$:

There exists a function $f \triangleleft tp_1, tp_2, ..., tp_k \triangleright (TE_1, TE_2, ..., TE_n) : T$ declared such that

1. it has k type parameters
2. it has n formal parameters
3. Let,

$$
\begin{aligned}
TE_1(tp_1 = t_1, tp_2 = t_2, ..., tp_k = t_k) &= T_1 \\
TE_2(tp_1 = t_1, tp_2 = t_2, ..., tp_k = t_k) &= T_2 \\
&... \\
TE_n(tp_1 = t_1, tp_2 = t_2, ..., tp_k = t_k) &= T_n
\end{aligned}
$$

Let $E_1 : T'_1$, $E_2 : T'_2$, ..., $E_n : T'_n$

Then $T_1 : T'_1$, $T_2 = T'_2$, ..., $T_n = T'_n$

If all the above work out, then the expression will type-check to $T(tp_1 = t_1, tp_2 = t_2, ..., tp_k = t_k)$ where T is the declared return type of f.

Examples:

```
n : int;
v : string;
map<| int, string |> rollnumbers;
...
...
add_map<| int, string |> (rollnumber, n, v)
```