5 Implementation

We will now discuss the implementation of the elm type system and then the implementation of the refinement types. We use this implementation for rapid prototyping.

Section 5.1 will discuss the implementation of the Elm type system in the research language K Framework. In section 5.2 we will go over the implementation of the refinement types in Elm. Sadly we could not do this implementation in K Framework, as it lacked a way to communicate to an SMT-Solver. Section 5.3 we will give a detailed walkthrough of the Elm code. In section 5.4 we will demonstrate the implemented algorithm by looking at an example code.

5.1 The Elm Type System in the K Framework

K Framework[RS14] was created in 2003 by Grigore Rosu. It is a research language and a system of tools for designing and formalizing programming languages. These include tools for parsing, execution, type checking and program verification[Ste+16]. Most of the features of the system are done by rewriting systems that are specified using its programming language called "K Language".

The main usage besides the creation and formalization of new languages is to create formal languages of existing programming languages. These include C[HER15], Java[BR15], JavaScript[PSR15], Php[FM14], Python[Gut13] and Rust[Kan+18].

The project was developed by the Formal Systems Laboratory Research Group and the University of Illinois, USA. The project itself is open source while the various more specialized tools are published under the Runtime Verification Inc. These include an analysing tool for C called RV-Match that is based on the formal C language written in K language[Gut+16] and more recently a tool for verifying smart contract written for the crypto-coin Ethereum [Hil+18].

We will be using K Framework to express small step semantics of the denotational semantics from a previous chapter. We can validate the semantic by letting K Framework apply the rewriting rules upon some examples.

```
require "unification.k"
require "elm-syntax.k"

module ELM-TYPESYSTEM
  imports DOMAINS
  imports ELM-SYNTAX

  configuration <k> $PGM:Exp </k>
                <tenv> .Map </tenv>
  //..

  syntax KResult ::= Type
endmodule
```

One can specify the realm upon which the rewriting system can be executed by using the `configuration` keyword. Here we specify two parts: `<k></k>` containing the expression and `<tenv></tenv>` containing the type context.

We also need to specify the end result using the `KResult` keyword. Once the rewriting system reaches such an expression, it will stop. If not specified the system might not terminate.

5.1.1 Implementing the Formal Language

To implement the formale Elm language in K Framework we need to translate the formal grammar into the K language.

```
syntax Type
  ::= "bool"
    | "int"
    | "{}Type"
    | "{" ListTypeFields "}Type" [strict]
    | Type "->" Type       [strict,right]
    | LowerVar
    | "(" Type ")"              [bracket]
    | ..
```

Additionally, we can include meta-information: `strict` to ensure the inner expression gets evaluated first, `right/left` to state in which direction the expressions should be evaluated and `bracket` for brackets that will be replaced with meta level brackets during paring.

Rules are written as rewriting rules instead of inference rules.

```
syntax Exp ::= Type
rule E1:Type E2:Type
  => E1 =Type (E2 -> ?T:Type)
    ~> ?T
syntax KResult ::= Type
```

The rule itself has the syntax `rule . => ..` The inner expressions need to be rewritten (into types) for the outer rule can be applied. We can include an additional `syntax` line before the rule and a `KResult` to ensure that rewriting system keeps on applying rules until a specific result has been reached. Only then it may continue.

Additionally, we have variables starting with an uppercase letter and existentially quantified variables starting with a question mark.

The system itself allows for a more untraditional imperative rewriting system using `~>`. This symbol has only one rule: `rule . ~> A => A` where `.` is the empty symbol. Thus,d the left part needs to be rewritten to `.` before the right part can be changed.

With all of this applied, the type system can infer types by applying rules as long as possible. But this only holds true for mono types. For poly types we need to implement the polymorphism, in particular instantiation and the generalization. The inference rules that we have presented in the section about type inference are not monomorphic and therefore can't be implemented. So in order to implement them we need to modify the slightly.

5.1.2 Implementing Algorithm J

In the original paper by Milner[Mil78] an optimized algorithm is presented for implementing polymorphism in a programming language. This algorithm is imperative but is typically presented as logical rules:

$$\frac{a : T_1 \quad T_2 = \mathit{inst}(T_1)}{\Gamma \vdash_J a : T_2} \qquad \text{[Variable]}$$

$$\frac{\Gamma \vdash_J e_0 : T_0 \quad \Gamma \vdash_J e_1 : T_1 \quad T_2 = \mathit{newvar} \quad \mathit{unify}(T_0, T_1 \to T_2)}{\Gamma \vdash_J e_0 e_1 : T_2} \qquad \text{[Call]}$$

$$\frac{T_1 = \mathit{newvar} \quad \Gamma, x : T_1 \vdash_J e : T_2}{\Gamma \vdash_J \texttt{\textbackslash}x\texttt{->}e : T_0 \to T_1} \qquad \text{[Lambda]}$$

$$\frac{\Delta_1 \vdash_J e_0 : T_1 \quad \Delta_1, a : \mathrm{insert}_{\Delta_1}(\{T_1\}) \vdash_J e_1 : T_2}{\Delta \vdash_J \texttt{let}x\texttt{=}e_0\texttt{in}e_1 : T_2} \qquad \text{[LetIn]}$$

So all we need to do, is to replace the rules of *let in*, *lambda*, *call* and *variable* with the rules above. The imperative functions are *newvar*, *unify* and *inst*:

- *newvar* creates a new variable.
- *inst* instantiates a type with new variables.
- *inify* checks whether two types can be unified.

K Framework has these imperative functions implemented in the `Unification.k` module. In order to use them, we need to first properly define poly types.

```
syntax PolyType ::= "forall" Set "." Type
```

Next we tell the system that we want to use the unification algorithm on types.

```
syntax Type ::= MetaVariable
```

Once this is set up, we can use the function `#renameMetaKVariables` for *inst* and ?T for *newvar*.

```
rule <k> variable X:Id => #renameMetaKVariables(T, Tvs) ...</k>
    <tenv>... X |-> forall Tvs . T
    ...</tenv>

rule <k> fun A:Id -> E:Type => ?T:Type -> E ~> setTenv(TEnv) ...</k>
    <tenv> TEnv:Map => TEnv [ A <- ?T ] </tenv>

syntax KItem ::= setTenv(Map)
  rule <k> T:Type ~> (setTenv(TEnv) => .) ...</k>
   <tenv> _ => TEnv </tenv>
```

Note that the `setTenv` function ensures that ?T is instantiated before its inserted into the environment.

For implementing unification we use `#metaKVariables` for getting all bound variables and `#freezeKVariables` to ensure that variables in the environment needs to be newly instantiated whenever they get used.

```
rule <k> let X = T:Type in E => E ~> setTenv(TEnv)
    ...</k>
    <tenv> TEnv
      => TEnv[ X
        <- forall
          (#metaKVariables(T) -Set #metaKVariables(setTenv(TEnv))) .
          ( #freezeKVariables(T, setTenv(TEnv)):>Type
          )
      ]
    </tenv>
```

As for *unify*, we can take advantage of the build-in pattern matching capabilities:

```
syntax KItem ::= Type "=Type" Type
rule T =Type T => .
```

By using a new function `=Type` with the rewriting rule `rule T =Type T => .` we can force the system to pattern match when ever we need to. Note that if we do not use this trick, the system will think that all existentially quantified variables are type variables and will therefore stop midway.

5.1.3 Example

We will now showcase how K-Framework infers types using the following example:

```
let
  model = []
in
(::) 1 model
```

We first need to write the example into a form that K-Framework can parse. Using the following syntax:

```
syntax Exp
    ::= "let" LowerVar "=" Exp "in" Exp               [strict(2)]
      | Exp Exp                                       [left,strict]
      | "[]Exp"
      | "intExp" Int
      | "(::)"
      | "variable" LowerVar
      | ..
```

Translating the program into our K-Framework syntax, this results in the following file.

```
<k>
let
  model = []Exp
in
((::) (intExp 1)) (variable model)
</k>
<tenv> .Map </tenv>
```

Here `.Map` denotes the empty type context. Also note that we have already applied the `left` rule. K-Framework uses this rule in parse-time, so this is just syntax sugar.

K-Framework will now walk through the abstract syntax tree to find the first term it can match. By specifying `strict(2)` we tell the system that `let in` can only be matched once `[]Exp` is rewritten. By appling the rule

```
rule []Exp => list ?A:Type
```

K-Framework obtains the following result.

```
<k>
let
  model = list ?A0:Type
in
((::) (intExp 1)) (variable model)
</k>
<tenv> .Map </tenv>
```

The system remembers the type hole `?A0` and will fill it in as soon as it finds a candidate for it. By using the rule

```
rule <k> let X = T:Type in E => E ~> setTenv(TEnv)
    ...</k>
    <tenv> TEnv
      => TEnv[ X
        <- forall
          (#metaKVariables(T) -Set #metaKVariables(setTenv(TEnv)))
          .
          ( #freezeKVariables(T, setTenv(TEnv)):>Type
          )
      ]
    </tenv>
```

the system rewrites the `let in` expression.

```
<k>
((::) (intExp 1)) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Note that we have just witnessed generalization: The free variable `?A` of the type got bound resulting in a poly type. These poly types only exist inside the type inference system.

The rule `Exp Exp` is strict, we therefore need to first rewrite `(::) (intExp 1)` and `variable model`. By appling the rules

```
rule (::) => ?A:Type -> ( list ?A ) -> ( list ?A )
rule intExp I:Int => int
```

the left expression can be rewritten.

```
<k>
((?A1:Type -> ( list ?A1 ) -> ( list ?A1 )) int) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

We can apply the expression using the rule

```
rule E1:Type E2:Type => E1 =Type (E2 -> ?T:Type) ~> ?T
```

and by pattern matching we fill in the type hole `?A1` with `int`.

```
<k>
(( list int ) -> ( list int )) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Next we need to get `model` out of the type context. By the rule

```
rule <k> variable X:Id => #renameMetaKVariables(T, Tvs) ...</k>
    <tenv>... X |-> forall Tvs . T
    ...</tenv>
```

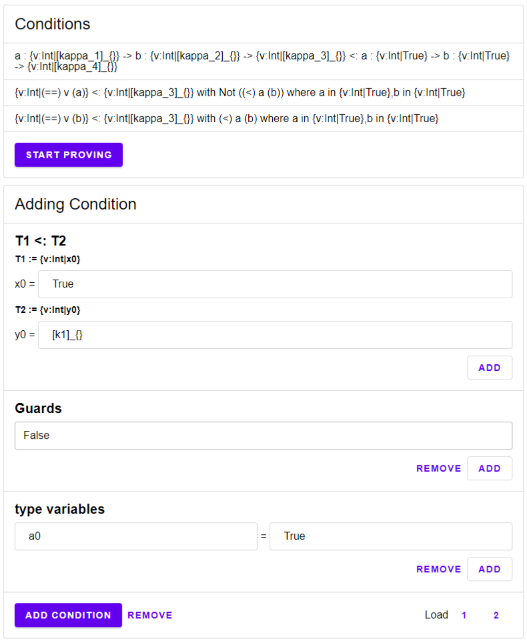
we obtain the following expression.

```
<k>
(( list int ) -> ( list int )) (list ?A2)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Note how the poly type was only used to store the variables that have been frozen. As we take a copy out of the type context, we instantiate the poly type resulting in a new type hole `?A1`.

Finally, we apply the expressions and again fill the type hole `?A2 = int` resulting in in our final expression.

```
<k>
list int
```


Conditions

a : {v:Int|[kappa_1]_{}} -> b : {v:Int|[kappa_2]_{}} -> {v:Int|[kappa_3]_{}} <: a : {v:Int|True} -> b : {v:Int|True} -> {v:Int|[kappa_4]_{}}

{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}

{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}

Adding Condition

T1 <: T2

T1 := {v:Int|x0}

x0 = True

T2 := {v:Int|y0}

y0 = [k1]_{}

Guards

False

type variables

a0 = True

Figure 1: A GUI for writing a set of input conditions.

```
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Here the rewriting system terminates, and the inferred type is `list int`.

5.2 Implementation in Elm

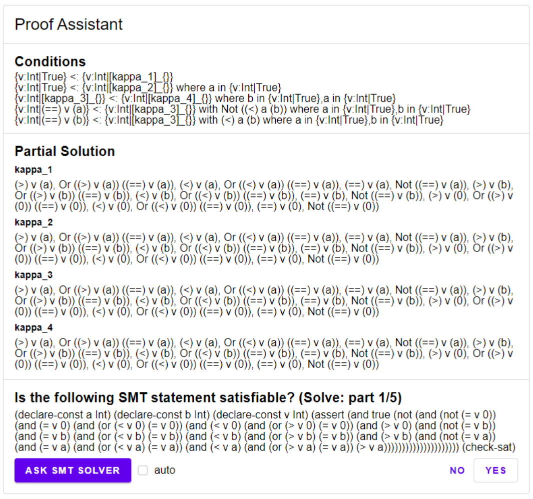
We will now turn to the implementation of the core of the type inference algorithm discussed in a previous chapter.

In particular, we will present the `split`, `solve` and `weaken` functions for computing the strongest refinements for a set of given subtyping conditions.

We have implemented these functions in Elm itself; to simplify testing, we have equipped the implementation with a GUI by using an Em package written by the author called Elm-Action [Pay20]. See Figure 1.

Elm is an immutable pure functional language. The architecture of a typical elm program is similar to that of a state machine: First a `init` function is called to define the initial state (in Elm typically called `Model`). The state is then passed to the `view` function that displays the state as HTML on the screen. The user can now interact with the elements on screen(like pressing a button). Once the user has done an interaction, a message describing the action will be passed to an `update` function, updating the current state (and the with that also the HTML on screen).

Our implementation has three different programs called `Setup`, `Assistant` and `Done`.



Proof Assistant

Conditions

```
{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}
```

Partial Solution

kappa_1

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

kappa_2

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

kappa_3

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

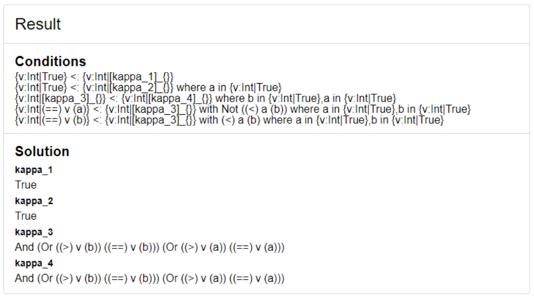
kappa_4

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

Is the following SMT statement satisfiable? (Solve: part 1/5)

```
(declare-const a Int) (declare-const b Int) (declare-const v Int) (assert (and true (not (and (not (= v 0))
(and (= v 0) (and (or (< v 0) (= v 0)) (and (< v 0) (and (or (> v 0) (= v 0)) (and (> v 0) (and (not (= v b))
(and (= v b) (and (or (< v b) (= v b)) (and (< v b) (and (or (> v b) (= v b)) (and (> v b) (and (not (= v a))
(and (= v a) (and (or (< v a) (= v a)) (and (< v a) (and (or (> v a) (= v a)) (> v a))))))))))))))))))))) (check-sat)
```

Figure 2: Proof assistant displaying the current SMT statement



Result

Conditions

```
{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}
```

Solution

kappa_1

True

kappa_2

True

kappa_3

```
And (Or ((>) v (b)) ((==) v (b))) (Or ((>) v (a)) ((==) v (a)))
```

kappa_4

```
And (Or ((>) v (b)) ((==) v (b))) (Or ((>) v (a)) ((==) v (a)))
```

Figure 3: The end result

The `Setup` program as seen in Figure 1 handles the creation of our conditions. The `Assistant` program as seen in Figure 2 applies the `split`, `solve` and `weaken` functions to the conditions. The `Done` program as seen in Figure 3 shows the solution.

Our library Elm-Action simplifies the wiring to combine multiple Elm programs into one. To do so, the library models the different Elm programs as different states of a meta-level state machine: Each state is its own state machine. To transition from one program into another we define a transition function that takes some transition data as an input and returns the initial state of the new elm program.

We will only discuss the `Assistant` program, as it is the most interesting. In this program our state describes a satisfiability problem. This SAT problem needs to be solved by either the SMT solver or a human. We are using the SMT solver called Z3. To talk to Z3 we use a small JavaScript code that communicates between Z3 and Elm. Elm will send the problem in question through JavaScript to Z3 and then awaits a response. Once the response has been received, it will then be sent to the `update` function, resulting in a new satisfiability problem. This new problem can be again sent to either Z3 or displayed on the screen. If this process stops, then the program ends and transitions into the `Done` program.

5.3 Details of the Elm implementation

We will now go over the Elm code in more detail.

5.3.1 Types

For Liquid Types we use the following representation:

```
type alias LiquidType a b =
    ( List
        { name : String
        , type : a
        }
    , b
    )
```

A function $a : \{Int|r1\} \to b : \{Int|r2\} \to \{Int|r3\}$ would be represented as `([{name=a,refinement=r1},{name=b,refinement=r2}],r3)`. We allow different types for `a` and `b`:

```
type SimpleLiquidType
    = IntType Refinement
    | LiquidTypeVariable Template
```

Possible types for `a` and `b` are either the most general `SimpleLiquidType` or the more specific types `Refinement` and `Template`. Note on the naming: `SimpleLiquidType` is “simple” in the sense that it is not a function type.

In respect to conditions we have two types:

```
type alias Condition =
    { smaller : LiquidType Template SimpleLiquidType
    , bigger : LiquidType Refinement Template
    , guards : List Refinement
    , typeVariables : List ( String, Refinement )
    }


type alias SimpleCondition =
    { smaller : SimpleLiquidType
    , bigger : Template
    , guards : List Refinement
    , typeVariables : List ( String, Refinement )
    }
```

`SimpleCondition` is the implementation of $\mathcal{C}^-$.

5.3.2 Transition

The `Assistant` program starts by obtaining some transition data from the `Setup` program. This transition data will then be used to initiate the state.

```
type alias Transition =
    List SimpleCondition
```

We obtain simple conditions from the `split` function. This is a one-to-one implementation of the *split* function previously described. We will now go through its definition.

—

```
split : Condition -> Result () (List SimpleCondition)
split =
  let
    rec : Int -> Condition -> Result () (List SimpleCondition)
    rec offset condition =
      case ( condition.smaller, condition.bigger ) of
        ( ( q1 :: t2, t2end ), ( q3 :: t4, t4end ) ) ->
          if q1.name == q3.name then
            rec (offset + 1)
              { condition
              | smaller = ( t2, t2end )
              , bigger = ( t4, t4end )
              , typeVariables =
                ( q3.name, q3.refinement )
                  :: condition.typeVariables
              }
              |> Result.map
                ((::)
                  { smaller = IntType q3.refinement
                  , bigger = q1.refinement
                  , guards = condition.guards
                  , typeVariables = condition.typeVariables
                  }
                )

          else
            Err ()
```

This first case is equivalent to the following.

$$\begin{aligned}&\mathrm{Split}(a:\{\nu:\mathit{Int}|q_1\}\to\hat{T}_2 <:_{\Theta,\Lambda} a:\{\nu:\mathit{Int}|q_3\}\to\hat{T}_4) = \\ &\qquad \{\{\nu:\mathit{Int}|q_3\} <:_{\Theta,\Lambda} \{\nu:\mathit{Int}|q_1\}\}\cup\mathrm{Split}(\hat{T}_2 <:_{\Theta\cup\{(a,q_3)\},\Lambda} \hat{T}_4\})\end{aligned}$$

—

```
        ( ( [], q1 ), ( [], q2 ) ) ->
          [ { smaller = q1
            , bigger = q2
            , guards = condition.guards
            , typeVariables = condition.typeVariables
            }
        ]
          |> Ok
```

The second case is a direct transformation from a `Condition` into a `SimpleCondition`. For our formal definition of the second case, this is equivalent to the identity.

$$\begin{aligned}&\text{Split}(\{\nu : \mathit{Int}|q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int}|q_2\}) = \\ &\quad \{\{\nu : \mathit{Int}|q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int}|q_2\}\}\end{aligned}$$

—

```
        _ ->
          Err ()
  in
  rec 0
```

The *split* function is a partial function, therefore we will return an error if neither case could be applied. If so, the `Setup` program will throw an error and the user would need to correct the given conditions. For a valid condition, the *split* function will always be successful. Once successful the new list of `SimpleConditions` will be passed as transition data to the `Assistant` program.

```
case model.conditions |> List.map function.split |> Result.combine of
    Ok conds ->
        conds |> List.concat |> Action.transitioning
    Err () ->
        ...
```

5.3.3 Init

After we have split the conditions, we initiate the Elm program. Note that this program will be implementing the `solve` and `weaken` functions.

```
init : Transition -> ( Model, Cmd Msg )
init conditions =
    let
        initList =
            (conditions
                |> List.map
                    (\{ typeVariables } ->
                        typeVariables
                            |> List.map (\( name, _ ) -> name)
                    )
                |> List.concat
            )
                |> Refinement.init
    in
    ( { conditions = conditions |> Array.fromList
      , predicates =
            conditions
```

```
                |> List.concatMap Condition.liquidTypeVariables
                |> List.map (\v -> ( v, initList |> Array.fromList ))
                |> Dict.fromList
      , index = 0
      , weaken = Nothing
      , auto = False
      , error = Nothing
      }
    , Cmd.none
    )
```

We now go through all fields of our model.

- `conditions` contains a copy of the conditions.
- `predicates` contains a dictionary, mapping every liquid type variable to the initial set of predicates $\mathit{Init}(V)$. (Equivalent to `Refinement.init`)
- `index` contains the index of the current condition. Keep in mind, that the loop from the Solve function is actually modelled as state transitions. Therefore, we can assume that we are always investigating one specific condition at a time. If not, then the program would have already stopped.
- `weaken` says if we are currently weakening a condition. If this is set to `Nothing` then we are in the `solve` function, else its `Just i` where `i` is the index of the predicate that we are currently investigating.
- `auto` is a boolean expression that says if the SMT solver should be asked directly. If set to `False`, then the user may decide the satisfiability of the current SMT statement.
- `error` contains any error message that should be displayed to the user. These errors come directly from the SMT solver.

5.3.4 Update

```
update : (String -> Cmd msg) -> Msg -> Model -> Update msg
update sendMsg msg model =
    case msg of
        GotResponse bool ->
            handleResponse sendMsg bool { model | error = Nothing }
        ...

handleResponse : (String -> Cmd msg) -> Bool -> Model -> Update msg
handleResponse sendMsg bool model =
    case model.weaken of
        Just weaken ->
            handleWeaken weaken sendMsg bool model

        Nothing ->
            handleSolve sendMsg bool model
```

We have stored the additional information needed for the `weaken` function in `model.weaken`. We therefore check the content of `model.weaken`. We check the content of `model.weaken`, If it is `Nothing` we know that we are in the `solve` function, else we know that we are currently in the `weaken` function.

THE SOLVE FUNCTION

```
handleSolve : (String -> Cmd msg) -> Bool -> Model -> Update msg
handleSolve sendMsg bool model =
    if bool then
        --Start weaking
        case
            model.conditions
                |> Array.get model.index
        of
            Just { bigger } ->
                { model
                    | weaken =
                        Just
                            { index = 0
                            , liquidTypeVariable = bigger |> Tuple.first
                            }
                }
                    |> handleAuto sendMsg

            Nothing ->
                Action.updating ( model, Cmd.none )
```

If the incoming result is `True` it means that the SMT statement is satisfiable. Therefore, we start the `weaken` function. To do so, we initiate the weakening index at `0` and also store the liquid type variable whose corresponding refinement we want to weaken.

—

```
    else
        --Continue
        let
            index =
                model.index + 1
        in
        if index >= (model.conditions |> Array.length) then
            Action.transitioning
                { conditions = model.conditions
                , predicates =
                    model.predicates
                        |> Dict.map
                          (\_ -> Array.toList >> Refinement.conjunction)
                }

        else
```

```
            { model
                | index = index
            }
                |> handleAuto sendMsg
```

If the incoming result is `False`, then we check out the next condition. If there exists no following condition, then the function is done. We end the Elm program by transitioning into the `Done` program.

THE WEAKEN FUNCTION

```
handleWeaken :
    { index : Int
    , liquidTypeVariable : Int
    }
    -> (String -> Cmd msg)
    -> Bool
    -> Model
    -> Update msg
handleWeaken weaken sendMsg bool model =
    if bool then
        --Remove
        let
            predicates =
                model.predicates
                    |> Dict.update weaken.liquidTypeVariable
                        (Maybe.map
                            (Array.removeAt weaken.index)
                        )
        in
        if
            weaken.index
                >= (predicates
                        |> Dict.get weaken.liquidTypeVariable
                        |> Maybe.map Array.length
                        |> Maybe.withDefault 0
                   )
        then
            { model
                | predicates = predicates
                , weaken = Nothing
                , index = 0
            }
                |> handleAuto sendMsg

        else
            { model
                | predicates = predicates
            }
                |> handleAuto sendMsg
```

If the incoming result is `False` then the SMT statement is unsatifiable. Thus, we remove the predicate. If no predicate exists, we finish the `weaken` function by setting `model.weaken` to `Nothing`.

—

```
    else
        --Continue
        let
            index =
                weaken.index + 1
        in
        if
            index
                >= (model.predicates
                        |> Dict.get weaken.liquidTypeVariable
                        |> Maybe.map Array.length
                        |> Maybe.withDefault 0
                   )
        then
            { model
                | weaken = Nothing
                , index = 0
            }
                |> handleAuto sendMsg

        else
            { model
                | weaken =
                    Just
                        { liquidTypeVariable = weaken.liquidTypeVariable
                        , index = index
                        }
            }
                |> handleAuto sendMsg
```

If the incoming result is `True` then the SMT statement is satisfiable. We therefore check out the next predicate. We finish the function if no following predicate exists. To do so we again set `model.weaken` to `Nothing`.

5.3.5 SMT Statement

After every update we check if the SMT statement should be automatically sent to the SMT solver.

```
handleAuto : (String -> Cmd msg) -> Model -> Update msg
handleAuto sendMsg model =
    if model.auto then
        ( model
        , model
            |> smtStatement
            |> Maybe.map sendMsg
            |> Maybe.withDefault Cmd.none
        )
            |> Action.updating

    else
        Action.updating
            ( model, Cmd.none )
```

If not, it will be displayed on the screen. Either way we need to compute the SMT statement for the given model.

```
smtStatement : Model -> Maybe String
smtStatement model =
    let
        toString : SimpleCondition -> String
        toString condition =
            case model.weaken of
                Just weaken ->
                    statementForWeaken weaken model condition

                Nothing ->
                    statementForSolve model condition
    in
    model.conditions
        |> Array.get model.index
        |> Maybe.map toString
```

The statement differes between the `solve` and the `weaken` function.

SMT STATEMENT FOR SOLVE

For the `solve` function we translate the condition directly into the SMT statement.

```
statementForSolve : Model -> SimpleCondition -> String
statementForSolve model condition =
    condition
        |> Condition.toSMTStatement
            (model.predicates
                |> Dict.map (\_ -> Array.toList >> Refinement.conjunction)
            )
```

The actual translation happens in `Condition.toSMTStatement`. The translation is taken directly from the described `solve` function. We therefore will now compare both with another.

—

```
toSMTStatement : Dict Int Refinement -> SimpleCondition -> String
toSMTStatement dict { smaller, bigger, guards, typeVariables } =
    let
        typeVariablesRefinements : List Refinement
        typeVariablesRefinements =
            typeVariables
                |> List.map
                    (\( b, r ) ->
                        r |> Refinement.rename
                            { find = "v"
                            , replaceWith = b
                            }
                    )
```

This equivalent to the following.

> Let
>
> $\Theta' := \{\ (a, r)$
>
> $\quad\mid r$ has the form $q \wedge (a,q) \in \Theta \wedge q \in \mathcal{Q}$
>
> $\quad\vee\ r$ has the form $[[k]_S]_{S_0} \wedge (a,q) \in \Theta$
>
> $\qquad\wedge\ q$ has the form $[k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}$
>
> $\{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta'$
>
> in $\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu, b_j)\}}$

—

```
        r1 : Refinement
        r1 =
            case smaller of
                IntType refinement ->
                    refinement

                LiquidTypeVariable ( int, list ) ->
                    list
                        |> List.foldl
                            (\( k, v ) ->
                                Refinement.substitute
                                    { find = k
                                    , replaceWith = v
                                    }
                            )
                            (dict
                                |> Dict.get int
                                |> Maybe.withDefault IsFalse
                    )
```

Here we have a case distinction between a refinement and a liquid type variable. We had the same distinction in our original definition of $r1$:

$$r_1 := \begin{cases} \bigwedge[S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and } S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases},$$

—

```
        r2 : Refinement
        r2 =
            bigger
                |> Tuple.second
                |> List.foldl
                    (\( k, v ) ->
                        Refinement.substitute
                            { find = k
                            , replaceWith = v
                            }
                    )
                    (dict
                        |> Dict.get (bigger |> Tuple.first)
                        |> Maybe.withDefault IsFalse
                    )
```

Here we see how we apply the lazy substitution (stored in `bigger |> Tuple.second`). In the original definition we assumed that we know how to apply a substitution on term level:

$$r_2 := \bigwedge[S(\kappa_2)]_{S_2}$$

—

```
        statement : Refinement
        statement =
            (r1
                :: typeVariablesRefinements
                ++ guards
            )
                |> List.foldl AndAlso (IsNot r2)
    in
    (statement
        |> Refinement.variables
        |> Set.toList
        |> List.map (\k -> "(declare-const " ++ k ++ " Int)\n")
        |> String.concat
    )
```

```
        ++ ("(assert "
               ++ (statement |> Refinement.toSMTStatement)
               ++ ")\n(check-sat)"
           )
```

The final statement is therefore

$$((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \wedge \neg r_2$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

SMT STATEMENT FOR WEAKEN

For the `weaken` function we modify the statement.

```
statementForWeaken :
    { index : Int, liquidTypeVariable : Int }
    -> Model
    -> SimpleCondition
    -> String
statementForWeaken weaken model condition =
  condition
    |> Condition.toSMTStatement
      (model.predicates
        |> Dict.map (\_ -> Array.toList >> Refinement.conjunction)
        |> Dict.update (condition.bigger |> Tuple.first)
          (Maybe.map
            (\_ ->
              model
                |> getLazySubstitute
                |> List.foldl
                  (\( find, replaceWith ) ->
                    Refinement.substitute
                      { find = find
                      , replaceWith = replaceWith
                      }
                  )
                  (model.predicates
                    |> Dict.get (condition.bigger |> Tuple.first)
                    |> Maybe.andThen (Array.get weaken.index)
                    |> Maybe.withDefault IsFalse
                  )
            )
          )
      )
```

We replace the value at the point `condition.bigger |> Tuple.first` with the predicate in question. The same happens in our formal definition. The resulting

SMT statement for the predicate q is therefore

$$((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \wedge \neg q$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

We therefore swap the result around: We keep the predicate if we SMT statement is unsatifiable. This is equivalent to saying we keep the predicate if the negated SMT statement is satifiable:

$$\neg((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \vee q$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

5.4 Demonstration

For this demonstration we will consider the following function.

```
max : a:{ v:Int|True } -> b:{ v:Int|True } -> { v:Int|k4 };
max =
  \a -> \b ->
    if
      (<) a b
    then
      b
    else
      a
```

To check the validity of the type signature, we will first infer the type of the function and then compare it with the type signature. Using the Inference rule, we obtain as a result the type

$$\{v : Int|\kappa_1\} \rightarrow \{v : Int|\kappa_2\} \rightarrow \{v : Int|\kappa_3\}$$

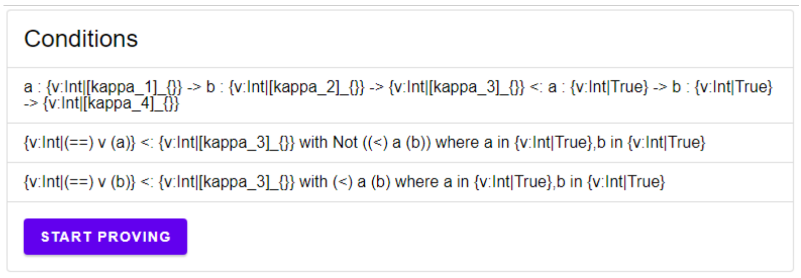
with the following conditions.

$$\{\nu : Int|\nu = b\} <:_{\{(a,\{Int|True\}),(b,\{Int|True\})\},\{a<b\}} \{\nu : Int|\kappa_3\},$$
$$\{\nu : Int|\nu = a\} <:_{\{(a,\{Int|True\}),(b,\{Int|True\})\},\{\neg(a<b)\}} \{\nu : Int|\kappa_3\},$$

We now write the validity check of the type signature as a condition.

$$\begin{aligned} &a : \{\nu : Int|\kappa_1\} \rightarrow b : \{\nu : Int|\kappa_2\} \rightarrow \{\nu : Int|\kappa_3\} \\ &\quad <:_{\{\},\{\}} a : \{\nu : Int|True\} \rightarrow b : \{\nu : Int|True\} \rightarrow \{\nu : Int|\kappa_4\} \end{aligned}$$

Figure 4 shows how the conditions can be inserted into the elm program.


Conditions

```
a : {v:Int|[kappa_1]_{}} -> b : {v:Int|[kappa_2]_{}} -> {v:Int|[kappa_3]_{}} <: a : {v:Int|True} -> b : {v:Int|True}
-> {v:Int|[kappa_4]_{}}
```

```
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
```

```
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}
```

Figure 4: The conditions of the max-function

If we click on the "Start Proving" button, the `Assistant` program will start and get the list of conditions as the transition data. It now applies the `split` function to the conditions and computes the first SMT statement, as seen in Figure 5.

Here we see the conditions on top, displaying the conditions that are now split. Next we see that for each `kappa` the set of predicated have been initiated with all possible predicates for variables `a` and `b`. Below it presents the first SMT statement in the `Solve`-step, mainly if the first condition is not satisfiable for the current value of `kappa_1`. Therefore, the SMT statement is satisfiable.

Next program goes into the `Weaken`-step as starts checking each and every predicate currently associated with `kappa_1`, as seen in Figure 6.

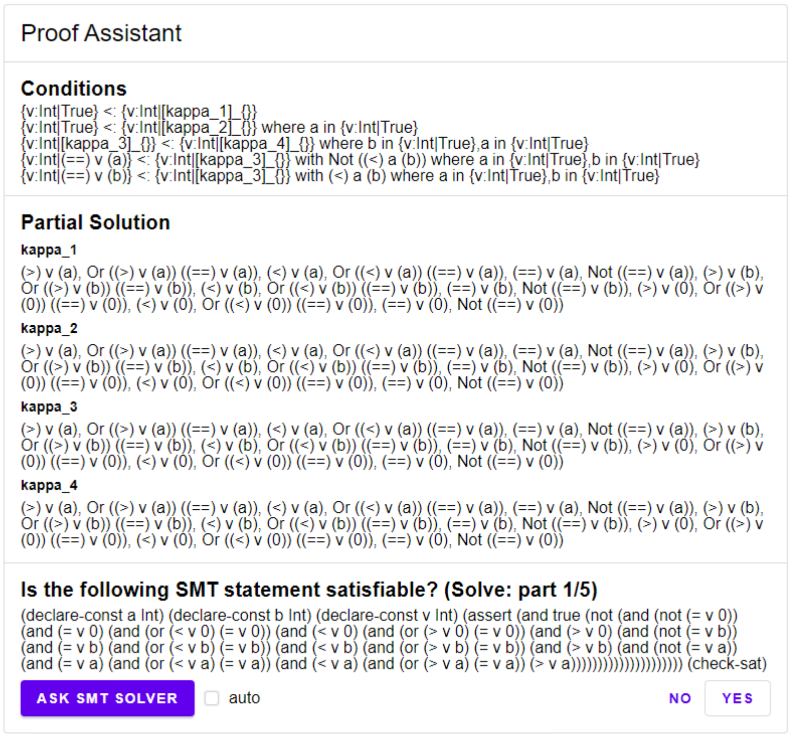
Once it has checked every predicate, it goes back to the `Solve`-step and repeats. In Figure 7 you can see the result after a few iterations.

Once every condition is valid (meaning that all SMT statements in the `Solve`-step are unsatifiable) the program holds and the result is displayed as seen in Figure 3.

References

[BR15] Denis Bogdanas and Grigore Roşu. "K-Java: A Complete Semantics of Java". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 445–456. ISSN: 0362-1340. DOI: 10.1145/2775051.2676982. URL: https://doi.org/10.1145/2775051.2676982.

[FM14] Daniele Filaretti and Sergio Maffeis. "An Executable Formal Semantics of PHP". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592. ISBN: 978-3-662-44202-9.


Proof Assistant

Conditions

```
{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}
```

Partial Solution

kappa_1

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

kappa_2

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

kappa_3

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

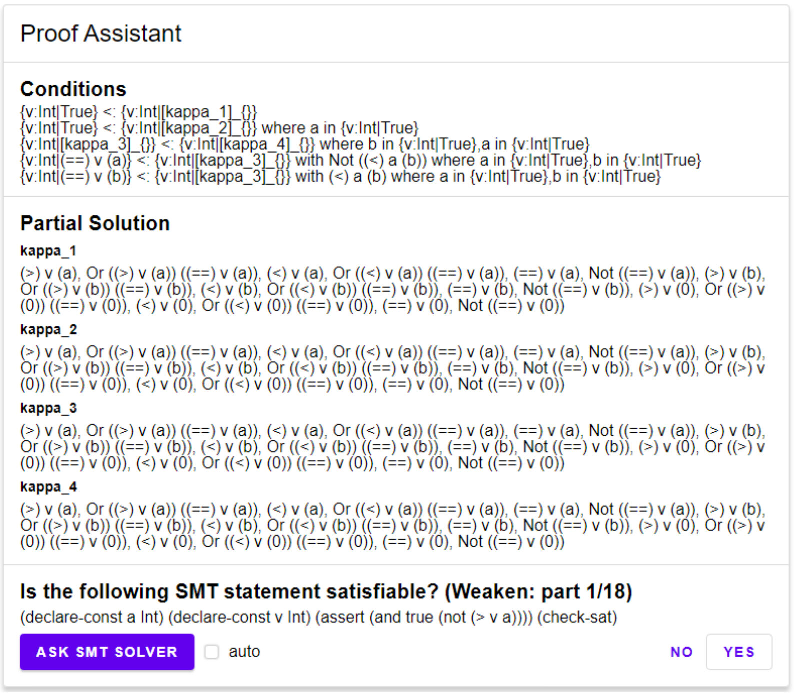
kappa_4

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

Is the following SMT statement satisfiable? (Solve: part 1/5)

```
(declare-const a Int) (declare-const b Int) (declare-const v Int) (assert (and true (not (and (not (= v 0))
(and (= v 0) (and (or (< v 0) (= v 0)) (and (< v 0) (and (or (> v 0) (= v 0)) (and (> v 0) (and (not (= v b))
(and (= v b) (and (or (< v b) (= v b)) (and (< v b) (and (or (> v b) (= v b)) (and (> v b) (and (not (= v a))
(and (= v a) (and (or (< v a) (= v a)) (and (< v a) (and (or (> v a) (= v a)) (> v a))))))))))))))))))))) (check-sat)
```

Figure 5: The conditions of the max-function


Proof Assistant

Conditions

```
{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}
```

Partial Solution

kappa_1

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

kappa_2

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

kappa_3

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

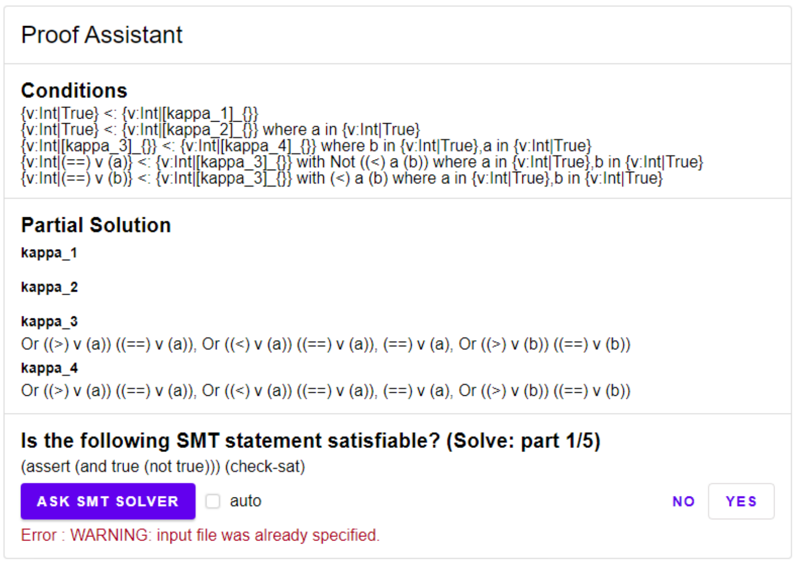
kappa_4

```
(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b),
Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v
(0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))
```

Is the following SMT statement satisfiable? (Weaken: part 1/18)

```
(declare-const a Int) (declare-const v Int) (assert (and true (not (> v a)))) (check-sat)
```

Figure 6: Weakening the predicates.


Proof Assistant

Conditions

```
{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}
```

Partial Solution

kappa_1

kappa_2

kappa_3

```
Or ((>) v (a)) ((==) v (a)), Or ((<) v (a)) ((==) v (a)), (==) v (a), Or ((>) v (b)) ((==) v (b))
```

kappa_4

```
Or ((>) v (a)) ((==) v (a)), Or ((<) v (a)) ((==) v (a)), (==) v (a), Or ((>) v (b)) ((==) v (b))
```

Is the following SMT statement satisfiable? (Solve: part 1/5)

```
(assert (and true (not true))) (check-sat)
```

```
Error : WARNING: input file was already specified.
```

Figure 7: Weakening the predicates.

[Gut+16] Dwight Guth et al. “RV-Match: Practical Semantics-Based Program Analysis”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I.* Vol. 9779. LNCS. Springer, July 2016, pp. 447–453. DOI: http://dx.doi.org/10.1007/978-3-319-41528-4_24.

[Gut13] Dwight Guth. “A formal semantics of Python 3.3”. In: 2013.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. “Defining the Undefinedness of C”. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 336–345. ISSN: 0362-1340. DOI: 10.1145/2813885.2737979. URL: https://doi.org/10.1145/2813885.2737979.

[Hil+18] Everett Hildenbrandt et al. “KEVM: A Complete Semantics of the Ethereum Virtual Machine”. In: *2018 IEEE 31st Computer Security Foundations Symposium.* IEEE, 2018, pp. 204–217.

[Kan+18] Shuanglong Kan et al. “K-Rust: An Executable Formal Semantics for Rust”. In: *CoRR* abs/1804.07608 (2018). arXiv: 1804.07608. URL: http://arxiv.org/abs/1804.07608.

[Mil78] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[Pay20] Lucas Payr. *Elm-Action.* https://github.com/Orasund/elm-action. 2020.

[PSR15] Daejun Park, Andrei Stefănescu, and Grigore Roşu. “KJS: A Complete Formal Semantics of JavaScript”. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 346–356. ISSN: 0362-1340. DOI: 10.1145/2813885.2737991. URL: https://doi.org/10.1145/2813885.2737991.

[RS14] Grigore Rosu and Traian-Florin Serbanuta. “K Overview and SIMPLE Case Study”. In: *Electr. Notes Theor. Comput. Sci.* 304 (2014), pp. 3–56. DOI: 10.1016/j.entcs.2014.05.002. URL: https://doi.org/10.1016/j.entcs.2014.05.002.

[Ste+16] Andrei Stefănescu et al. “Semantics-Based Program Verifiers for All Languages”. In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 74–91. ISSN: 0362-1340. DOI: 10.1145/3022671.2984027. URL: https://doi.org/10.1145/3022671.2984027.