Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively

Robert Atkey robert.atkey@strath.ac.uk

with
Matthew L. Daggitt and Ekaterina Komendantskaya (Heriot Watt University)
and Wen Kokke (University of Strathclyde)
and Luca Arnaboldi (University of Edinburgh)

Certified Programs and Proofs
17th January 2023

Goal: to translate high level specifications into SMT solvers with good error messages and guaranteed semantic preservation.

Vehicle: a language for specifying neural networks

We are developing a language called **Vehicle** for specifying neural networks.

It is a high-level language with higher-order functions, dependent types, arbitrary properties.

Multiple backends:

1. SMT(-like) solvers such as Marabou for verifying properties
2. ITPs such as Agda for using properties in larger proofs
3. Loss functions for training with built-in constraints

`https://github.com/vehicle-lang/vehicle`

Translating to SMT

Translating into SMT solver from a high-level language requires some work:

- Properties need to be within the subset handled by the solver.
- Higher order functions need to be reduced away.
- Features like **uninterpreted functions** (used by Marabou to stand for the neural net being analysed) and **if-then-else** need to be translated specially.
- We'd like to be confident that the translation is correct (equi-satisfiable).

A high-level specification language

```
type Input = Vec Real 5

fun f : Input -> Real

equalExceptAt : Index 5 -> Input -> Input -> Bool
equalExceptAt i x y = forall j. i != j => x!j == y!j

monotonic : Bool
monotonic =
    forall x, y. equalExceptAt 2 x y and x!2 <= y!2 => f x <= f y
```

$$\texttt{x0 == y0} \wedge \texttt{x1 == y1} \wedge \texttt{x2 <= y2} \wedge \texttt{x3 == y3} \wedge \texttt{x4 == y4} \wedge$$
$$\texttt{z1 == f [x0, x1, x2, x3, x4]} \wedge$$
$$\texttt{z2 == f [y0, y1, y2, y3, y4]} \wedge$$
$$\texttt{z1 > z2}$$

```
inRange : Real -> Bool
inRange y = exists x. f x == y

-- Modular use:
zeroAndOneInRange : Bool
zeroAndOneInRange = inRange 0 and inRange 1

-- "Hidden" Mixed Quantifiers:
surjective : Bool
surjective = forall y. inRange y
```

Translation to SMT

To translate to a solver.

Avoid

1. Non-linear constraints
2. Mixed quantifiers

Translate away

1. Higher-order functions
2. Nested function applications and if-then-elses

Analysis target

Intermediate Language: Annotated Types

Annotate types with information about what kinds of properties they describe.

```
Real : Linearity -> Type
Bool : Linearity -> Polarity -> Type
```

```
C, L, N : Linearity
U, ∀, ∃, P, A : Polarity
```

Intermediate Language: Annotated Types

Annotate types with information about what kinds of properties they describe.

```
Real : Linearity -> Type
Bool : Linearity -> Polarity -> Type
```

```
C, L, N : Linearity
U, ∀, ∃, P, A : Polarity
```

We do not disallow non-linear or mixed quantifiers; we note that they are used.

Intermediate Language: Annotated Operations

Arithmetic

```
+ : ∀ {l1 l2 l3}. {{MaxLin l1 l2 l3}} -> Real l1 -> Real l2 -> Real l3
* : ∀ {l1 l2 l3}. {{MulLin l1 l2 l3}} -> Real l1 -> Real l2 -> Real l3
```

Constraints

```
<= : ∀ {l1 l2 l3}. {{MaxLin l1 l2 l3}} ->
                   Real l1 -> Real l2 -> Bool l1 U
```

Logic

```
and : ∀ {l1 l2 l3 p1 p2 p3}.
    {{MaxLin l1 l2 l3}} -> {{MaxPol p1 p2 p3}} ->
    Bool l1 p1 -> Bool l2 p3 -> Bool l3 p3

forall : ∀ {t1 t2}. {{HasForall t1 t2}} -> t1 -> t2
```

Type class resolution rules provide evidence

Linear combination

`MaxLin l1 l2 l3` exactly when `l3` is the max of `l1` and `l2`. `C < L < N`

Multiplicative combination

```
MulLin C C C      MulLin C L L      MulLin L C L
                MulLin L L N              ...
```

Quantification

$$\frac{\texttt{ForallPol p1 p2}}{\texttt{HasForall (Real L -> Bool l p1) (Bool p2)}}$$

$$\frac{}{\texttt{HasForall (Index n -> Bool l p) (Bool p)}}$$

Analysis by Elaboration

Elaborating Types

Inserting extra type variables:

```
type Input = Vec Real 5

fun f : Input -> Real
```

Elaborating Types

Inserting extra type variables:

```
type Input = Vec Real 5

fun f : Input -> Real
```

becomes

```
type Input l = Vec (Real l) 5

fun f : ∀ {l1 l2}. {{MaxLin L l1 l2}} -> Input l1 -> Real l2
```

Elaborating Definitions

For each user definition:

1. Insert linearity and polarity meta-variables into the type
2. Perform type checking and type class resolution to:
   2.1 Solve where possible
   2.2 Gather remaining constraints
3. Add constraints to track source-level function names
   (used for error messages)
4. Generalise (Hindley-Milner style) to put unsolved constraints into the type

```
equalExceptAt : Index 5 -> Input -> Input -> Bool
equalExceptAt i x y = forall j. i != j => x ! j == y ! j
```

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 ?3
equalExceptAt i x y = forall j. i != j => x ! j == y ! j
```

Type check (bidirectionally with meta-variable insertion for implicits)

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 ?3
```

$$\texttt{equalExceptAt i x y = }\underbrace{\texttt{forall j. }\underbrace{\underbrace{\texttt{i != j}}_{\texttt{Bool ?6 ?8}}\texttt{ => }\underbrace{\underbrace{\texttt{x ! j}}_{\texttt{Real ?0}}\texttt{ == }\underbrace{\texttt{y ! j}}_{\texttt{Real ?1}}}_{\texttt{Bool ?7 ?9}}}_{\texttt{Bool ?4 ?5}}}_{\texttt{Bool ?2 ?3}}$$

Gathered constraints:

```
HasEq (Index 5) (Index 5) (Bool ?6 ?8)
HasEq (Real ?0) (Real ?1) (Bool ?7 ?9)
MaxLin ?6 ?7 ?4, ImpliesPol ?8 ?9 ?5
HasForall (Index 5 -> Bool ?4 ?5) (Bool ?2 ?3)
```

From type checking:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 ?3
```

with constraints:

```
HasEq (Index 5) (Index 5) (Bool ?6 ?8)
HasEq (Real ?0) (Real ?1) (Bool ?7 ?9)
MaxLin ?6 ?7 ?4, ImpliesPol ?8 ?9 ?5
HasForall (Index 5 -> Bool ?4 ?5) (Bool ?2 ?3)
```

From type checking:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 ?3
```

with constraints:

```
HasEq (Index 5) (Index 5) (Bool ?6 ?8)
HasEq (Real ?0) (Real ?1) (Bool ?7 ?9)
MaxLin ?6 ?7 ?4, ImpliesPol ?8 ?9 ?5
HasForall (Index 5 -> Bool ?4 ?5) (Bool ?2 ?3)
```

Solve (via the type class resolution rules and unification) to get:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 U
```

with constraints: `MaxLin ?0 ?1 ?7`, `MaxLin C ?7 ?2`

We have:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 U
```

with constraints: `MaxLin ?0 ?1 ?7`, `MaxLin C ?7 ?2`

We have:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 U
```

with constraints: `MaxLin ?0 ?1 ?7`, `MaxLin C ?7 ?2`

We could now generalise the type to include these constraints:

```
equalExceptAt :
  ∀ {l1 l2 l3 l4}. {{MaxLin l1 l2 l3}} -> {{MaxLin C l3 l4}} ->
                   Index 5 -> Input l1 -> Input l2 -> Bool l4 U
```

We have:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 U
```

with constraints: `MaxLin ?0 ?1 ?7`, `MaxLin C ?7 ?2`

We could now generalise the type to include these constraints:

```
equalExceptAt :
  ∀ {l1 l2 l3 l4}. {{MaxLin l1 l2 l3}} -> {{MaxLin C l3 l4}} ->
                   Index 5 -> Input l1 -> Input l2 -> Bool l4 U
```

But this wouldn't give good error messages: when functions are applied, we unify variables and lose track of what functions are used.

We have:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 U
```

with constraints: `MaxLin ?0 ?1 ?7`, `MaxLin C ?7 ?2`

We have:

```
equalExceptAt : Index 5 -> Input ?0 -> Input ?1 -> Bool ?2 U
```

with constraints: `MaxLin ?0 ?1 ?7`, `MaxLin C ?7 ?2`

Function I/O constraints to track the function usage:

```
equalExceptAt : Index 5 -> Input ?10 -> Input ?11 -> Bool ?12 U
```

with constraints:

```
MaxLin ?0 ?1 ?7, MaxLin C ?7 ?2
InputLin "equalExceptAt" ?10 ?0
InputLin "equalExceptAt" ?11 ?1
OutputLin "equalExceptAt" ?2 ?12
```

These constraints will add provenance information (later...)

We have:

```
equalExceptAt : Index 5 -> Input ?10 -> Input ?11 -> Bool ?12 U
```

with constraints:

```
MaxLin ?0 ?1 ?7, MaxLin C ?7 ?2
InputLin "equalExceptAt" ?10 ?0
InputLin "equalExceptAt" ?11 ?1
OutputLin "equalExceptAt" ?2 ?12
```

We have:

```
equalExceptAt : Index 5 -> Input ?10 -> Input ?11 -> Bool ?12 U
```

with constraints:

```
MaxLin ?0 ?1 ?7, MaxLin C ?7 ?2
InputLin "equalExceptAt" ?10 ?0
InputLin "equalExceptAt" ?11 ?1
OutputLin "equalExceptAt" ?2 ?12
```

Generalise to get the final type:

```
equalExceptAt : ∀ {l1 l2 l3 l4 l5 l6 l7}.
  {{InputLin "equalExceptAt" l5 l1}} ->
  {{InputLin "equalExceptAt" l6 l2}} ->
  {{OutputLin "equalExceptAt" l3 l7}} ->
  {{MaxLin l1 l2 l4}} -> {{MaxLin C l4 l3}} ->
  Input 5 -> Input l5 -> Input l6 -> Bool l7 U
```

The rest of the specification elaborated:

```
equalExceptAt : ∀ {l1 l2 l3 l4 l5 l6 l7}.
  {{InputLin "equalExceptAt" l5 l1}} ->
  {{InputLin "equalExceptAt" l6 l2}} ->
  {{OutputLin "equalExceptAt" l3 l7}} ->
  {{MaxLin l1 l2 l4}} -> {{MaxLin C l4 l3}} ->
  Input 5 -> Input l5 -> Input l6 -> Bool l7 U
equalExceptAt i x y = forall j. i != j => x!j == y!j

monotonic : Bool L ∀
monotonic =
    forall x, y. equalExceptAt 2 x y and x!2 <= y!2 => f x <= f y
```

The rest of the specification elaborated:

```
equalExceptAt : ∀ {l1 l2 l3 l4 l5 l6 l7}.
  {{InputLin "equalExceptAt" l5 l1}} ->
  {{InputLin "equalExceptAt" l6 l2}} ->
  {{OutputLin "equalExceptAt" l3 l7}} ->
  {{MaxLin l1 l2 l4}} -> {{MaxLin C l4 l3}} ->
  Input 5 -> Input l5 -> Input l6 -> Bool l7 U
equalExceptAt i x y = forall j. i != j => x!j == y!j

monotonic : Bool L ∀
monotonic =
    forall x, y. equalExceptAt 2 x y and x!2 <= y!2 => f x <= f y
```

From the type: this specification is linear and only uses universal quantification.

What if it goes wrong?

What if we get a type we don't like?

- Linearity and polarity values are annotated with *provenance*
- The internal representation of the type

  $$\texttt{monotonic} : \texttt{Bool L } \forall$$

  stores the fact that the $\forall$ came from the use of `forall`, and the `L` came from the use of the unknown function `f`.
- Provenance information is used for returning error messages.

Error message example

```
inRange : Real -> Bool
inRange y = exists x. f x == y

surjective : Bool
surjective = forall y. inRange y
```

yields the type `Bool L A` (linear but alternating).

Error message example

```
inRange : Real -> Bool
inRange y = exists x. f x == y

surjective : Bool
surjective = forall y. inRange y
```

yields the type `Bool L A` (linear but alternating).

We can trace the provenance to get the error message:

Cannot verify specifications with alternating quantifiers. In particular:

1. *the inner quantifier is the 'exists' located at line 2, columns 12-18*
2. *which is returned as the output of the call to the function 'inRange' at line 1, columns 24-31*
3. *which alternates with the outer 'forall' quantifier at line 5, columns 13-19.*

Analysis Summary

Have defined an analysis to discover what kind of property a specification specifies:

1. *Compositional*: each definition is analysed once and summarised in its type.
2. *Produces good error messages*: provenance information is tracked to pinpoint errors.
3. *Evidence generating*: the output is a typed program with information on why each linearity and polarity decision holds.

Translating to SMT

Normalisation by Evaluation

We normalise a specification by evaluating it in a “non-standard” way as syntax.

Normalisation by Evaluation

We normalise a specification by evaluating it in a “non-standard” way as syntax.

- The `Real` type is interpreted as real-valued expressions
- The `Bool` type is interpreted as boolean-value expressions

Normalisation by Evaluation

We normalise a specification by evaluating it in a “non-standard” way as syntax.

- The `Real` type is interpreted as real-valued expressions
- The `Bool` type is interpreted as boolean-value expressions

Linearity and Polarity information refines these interpretations:

- `Real C` – constants
- `Real L` – linear expressions in some variables
- `Real N` – arbitrary expressions

Normalisation by Evaluation

We normalise a specification by evaluating it in a “non-standard” way as syntax.

- The `Real` type is interpreted as real-valued expressions
- The `Bool` type is interpreted as boolean-value expressions

Linearity and Polarity information refines these interpretations:

- `Real C` – constants
- `Real L` – linear expressions in some variables
- `Real N` – arbitrary expressions

Each primitive operation (e.g., `+`, `and`) is interpreted as a syntactic manipulation.

Normalisation by Evaluation

We normalise a specification by evaluating it in a “non-standard” way as syntax.

- The `Real` type is interpreted as real-valued expressions
- The `Bool` type is interpreted as boolean-value expressions

Linearity and Polarity information refines these interpretations:

- `Real C` – constants
- `Real L` – linear expressions in some variables
- `Real N` – arbitrary expressions

Each primitive operation (e.g., +, `and`) is interpreted as a syntactic manipulation.

Closed term of type `Bool L ∃` is guaranteed to yield an existential query with linear constraints.

Technicalities I : Free variables

Not-yet-quantified expressions and constraints may have free variables.

So type interpretations are parameterised by linear variable contexts

$$[\![\texttt{Real C}]\!] : \mathit{LinVarCtxt} \longrightarrow \mathsf{Set}$$

and must support renaming.

Types are presheaves over the category of linear variable contexts.

Most of the rest of the interpretation follows, using the standard interpretation of λ-calculus in a presheaf category. **Higher-order functions melt away.**

Technicalities II : Function lifting and If-then-else

Specifications may mix linear expressions and uninterpreted functions:

$$\texttt{f x + 5 < y}$$

and these need to be translated into separate constraints:

$$\exists\ \texttt{z. f x = z} \land \texttt{z + 5 < y}$$

Technicalities II : Function lifting and If-then-else

Specifications may mix linear expressions and uninterpreted functions:

$$\texttt{f x + 5 < y}$$

and these need to be translated into separate constraints:

$$\exists\ \texttt{z. f x = z} \wedge \texttt{z + 5 < y}$$

May also contain if-then-else:

$$\texttt{(if p then x else y) < z}$$

which must be translated into boolean logic:

$$\texttt{(p} \wedge \texttt{x < z)} \vee \texttt{(}\neg\ \texttt{p} \wedge \texttt{y < z)}$$

Technicalities II : Function lifting and If-then-else

Implementing these on syntax that does not support it is not possible.

Technicalities II : Function lifting and If-then-else

Implementing these on syntax that does not support it is not possible.

So we add the ability to **make definitions** (for function lifting) and to do **if-then-else** as *effects* from a pervasive monad.

Technicalities II : Function lifting and If-then-else

Implementing these on syntax that does not support it is not possible.

So we add the ability to **make definitions** (for function lifting) and to do **if-then-else** as *effects* from a pervasive monad.

Define a monad Lift with the following operations:
(interpreted with respect to a linear variable context)

- $\mathsf{if} : \mathsf{Constraint} \to \mathsf{Lift}\,A \to \mathsf{Lift}\,A \to \mathsf{Lift}\,A$
- $\mathsf{letExp} : \mathsf{LinExp} \to (\mathsf{Var} \to \mathsf{Lift}\,A) \to \mathsf{Lift}\,A$
- $\mathsf{letFun} : \mathsf{Var} \to (\mathsf{Var} \to \mathsf{Lift}\,A) \to \mathsf{Lift}\,A$

Technicalities II : Function lifting and If-then-else

Implementing these on syntax that does not support it is not possible.

So we add the ability to **make definitions** (for function lifting) and to do **if-then-else** as *effects* from a pervasive monad.

Define a monad Lift with the following operations:
(interpreted with respect to a linear variable context)

- if : Constraint $\rightarrow$ Lift A $\rightarrow$ Lift A $\rightarrow$ Lift A
- letExp : LinExp $\rightarrow$ (Var $\rightarrow$ Lift A) $\rightarrow$ Lift A
- letFun : Var $\rightarrow$ (Var $\rightarrow$ Lift A) $\rightarrow$ Lift A

The interpretation uses the Moggi Call-by-Value monadic translation.

With a Lift Constraint value, the operations are translated into real ones.

Normalisation by Evaluation

The procedure has been formalised in Agda.

Syntax of the analysed source is in an intrinsically typed nameless representation $t : \Gamma \vdash \tau$.

Normalisation is a function:

$$\mathcal{N}[\![-]\!] : \epsilon \vdash \texttt{Bool L } \exists \longrightarrow \mathsf{PrenexFormula}$$

We also defined a *standard semantics*:

$$\mathcal{S}[\![-]\!] : (\mathbb{Q} \rightarrow \mathbb{Q}) \rightarrow \epsilon \vdash \texttt{Bool}\ lp \rightarrow \mathsf{Set}$$

parameterised by the interpretation of the uninterpreted function.

Correctness

Via a logical relations argument (actually another interpretation), we get agreement between the standard and normalised semantics:

Theorem

For closed terms $t : \vdash \mathtt{Bool}\ L\exists$, the standard semantics and the interpretation of the normalising semantics are equi-satisfiable, for all concrete interpretations of the (syntactically) uninterpreted function f:

$$\mathcal{S}[\![t]\!]\, f \Leftrightarrow [\![\mathcal{N}[\![t]\!]]\!]$$

Proved in Agda.

Conclusions

Contributions and Future Work

Goal: to translate high level specifications into SMT solvers with good error messages and guaranteed semantic preservation.

- Compositional analysis
- Provenance tracking for good error messages
- Novel NbE procedure with correctness proof

`https://github.com/vehicle-lang/vehicle`
`https://github.com/vehicle-lang/vehicle-formalisation`

Contributions and Future Work

Goal: to translate high level specifications into SMT solvers with good error messages and guaranteed semantic preservation.

- Compositional analysis
- Provenance tracking for good error messages
- Novel NbE procedure with correctness proof

`https://github.com/vehicle-lang/vehicle`
`https://github.com/vehicle-lang/vehicle-formalisation`

Future work:

- Close gaps between the formalisation and the real implementation
- Efficiency of constraint solving
- Other SMT theories
- Generalising and applying the NbE procedure to other DSLs