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

Matthew L. Daggitt
matthewdaggitt@gmail.com
Heriot-Watt University
Edinburgh, UK

Robert Atkey
robert.atkey@strath.ac.uk
University of Strathclyde
Glasgow, UK

Wen Kokke
University of Strathclyde
Glasgow, UK

Ekaterina Komendantskaya
Heriot-Watt University
Edinburgh, UK

Luca Arnaboldi
University of Edinburgh
Edinburgh, UK

Abstract

Modern verification tools frequently rely on compiling high-level specifications to SMT queries. However, the high-level specification language is usually more expressive than the available solvers and therefore some syntactically valid specifications must be rejected by the tool. In such cases, the challenge is to provide a comprehensible error message to the user that relates the original syntactic form of the specification to the semantic reason it has been rejected.

In this paper we demonstrate how this analysis may be performed by combining a standard unification-based type-checker with type classes and automatic generalisation. Concretely, type-checking is used as a constructive procedure for under-approximating whether a given specification lies in the subset of problems supported by the solver. Any resulting proof of rejection can be transformed into a detailed explanation to the user. The approach is compositional and does not require the user to add extra typing annotations to their program. We subsequently describe how the type system may be leveraged to provide a sound and complete compilation procedure from suitably typed expressions to SMT queries, which we have verified in Agda.

CCS Concepts: **• Software and its engineering → Domain specific languages**; **• Hardware → Theorem proving and SAT solving**.

Keywords: SMT solvers, verification, domain specific languages, type-checking, compilers, Agda

ACM Reference Format:
Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. 2023. Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3573105.3575674

1 Introduction

As the performance of SMT solvers and other automatic theorem provers has improved, they have been applied to a wide range of domains, including policy verification [2], program synthesis [24, 25], model-based testing [16] and neural network verification [14]. However, manually writing queries for these solvers is tedious and error-prone, so tools often provide a higher-level domain-specific language (DSL) which is then compiled down to an equisatisfiable set of queries.

Such DSLs aim to provide a layer of abstraction between the user and the solvers, but do not entirely succeed. The main failure point of the abstraction is that each individual solver only supports a limited class of problems (known as "logics" in SMTLib) and that some classes of problems are more difficult than others. Therefore, an apparently innocuous tweak to a high-level specification may change the class of problems the compiled queries belong to and consequently the choice of solver. In turn, this may drastically increase the time taken to check the specification or even render it unable to be solved if no suitable solver is available.

We have encountered the latter issue while designing a DSL for writing specifications for neural networks solvers. Due to their large size and inherent complexity, neural networks require specialised solvers [14, 29] which generally only support problems that belong to some subset of quantifier-free linear real arithmetic (QF_LRA). As we made our language more expressive, it became increasingly easy to write specifications that cannot be compiled to an equivalent set of queries in QF_LRA.

To illustrate how a user may write such a specification, consider a higher-order functional DSL for writing specifications equipped with the quantifiers `forall` and `exists` as first-class constructs in the language. We would like to compile specifications about some function $f : \mathbb{R}^n \to \mathbb{R}$ down to queries for a QF_LRA solver. The exact semantics of f is unimportant; it could be an existing piece of C++ code, a neural network or an uninterpreted function to find a solution for. The user begins by writing a function that asserts some value y is in the range of the function f:

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

This can then be reused to modularly specify that, for example, 0 and 1 are in the range of f:

```
zeroAndOneInRange : Bool
zeroAndOneInRange = inRange 0 and inRange 1
```

This specification is equisatisfiable with the quantifier-free query '$f\ a == 0 \wedge f\ b == 1$' and which can then be solved by a QF_LRA solver that can handle f appropriately. However, the user may then use the innocuous `inRange` definition to specify that f is surjective:

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

Fully expanded, this specification has alternating `forall` and `exists` quantifiers and therefore there is no equisatisfiable quantifier-free set of queries it can be compiled to.

1.1 Existing Approaches

One approach to addressing this issue is to design the syntax of the DSL so that all expressions in the language belong to the supported class of problems. For example in Liquid Haskell [27] and Cryptol [17], `exists` is not a first class construct in the language and therefore it is syntactically impossible for users to write non-quantifier-free statements. In particular, it would not be possible to define `inRange` in the examples above. While restricting quantifiers in this way is feasible, applying the same approach to non-linear expressions is significantly more difficult as multiplication is almost universally a first-class construct in languages. The approach also inhibits modularity, as functions containing quantifiers (e.g. `inRange`) cannot be declared and then reused in multiple places, (e.g. as in `zeroOneInRange`). Finally, it is inherently inapplicable when the DSL is compiled to multiple backends with different capabilities, e.g. a solver and some other non-solver, or multiple solvers supporting different logics.

An alternative approach is to first normalise the specification to remove all functions and then subsequently perform a simple syntactic check. This approach is adopted by the Z3 SMTLib frontend [11]. While this procedure is both sound and complete, normalisation discards information about the specification the user has actually written. Therefore when the syntactic check subsequently encounters a problem, it cannot explain why or where in the source file a problem has occurred. For example, asking Z3 to solve an SMTLib encoding of `surjective` with a QF_LRA solver results in an error saying that the quantifiers are unsupported but cannot provide useful information to the user about the location of the quantifiers or how they interact. Although in this case it is easy to spot the problem, it may not be so easy in a large real-world specification where the definition of `inRange` may be far from `surjective` or used through a chain of intermediate function calls.

Finally, tools such as F* [26] throw generic error messages that do not even mention alternating quantifiers. We have collected example messages in Appendix A.

1.2 Our Contributions

In this paper, we propose a solution to this problem in the form of a type-system that can be used to soundly approximate membership of QF_LRA for an expressive specification language with higher-order functions. The basic idea is for the compiler to internally refine the `Bool` and `Real` types into families of types indexed by quantities tracking precisely how they use quantified variables. For example, one type might be the type of Booleans which universally quantify over real numbers which are used linearly. Polymorphism over these quantities is retained via a standard higher-order unification-based type-checker and what we believe to be a novel combination of Haskell-style type-class propagation [28] and Idris-style implicit generalisation [7]. Crucially our approach does not require the user to add additional typing annotations, nor for them to understand advanced type system features.

Type-checking can then be used as a constructive decision procedure for membership, from which a proof of whether a specification can or cannot be compiled can be extracted. In the latter case, the proof can then be used to construct a detailed error message for the user. For example, passing in the example specification `surjective` results in the error:

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 7, columns 13-19.*

When the type-checking procedure succeeds, we are guaranteed that the specification can be compiled to a format suitable for a QF_LRA solver. In Section 4, we constructively demonstrate this fact by presenting a Normalisation by Evaluation (NbE) procedure [5] that accepts the fully elaborated output of our type checker and generates a semantically equivalent formula. NbE for our specification language is non-trivial due to the fact that our specification language allows mixing of if-then-else and uninterpreted function applications anywhere in terms. Both of these need to be extracted from the context in which they appear and hoisted to the level of Boolean constraints. We explain our method for accomplishing this lifting via a monad in Section 4.3.3. The totality, type- and semantic-preservation properties of our NbE procedure have all been formalised in the Agda proof assistant [21].

Our approach has been incorporated into the DSL of Vehicle [10], a new tool for writing and checking neural network specifications. While the paper focuses on QF_LRA, the technique is modular in the sense that it decides membership of quantifier-freeness separately from linearity. Therefore, while quantifier-freeness and linearity are arguably the two most important subsets of SMT logics, we hypothesise that the technique is more generally applicable to approximating membership of other SMT logics, e.g. integer difference logics [15].

The paper is laid out as follows. Section 2 presents an example of a higher-order DSL for writing specifications about functions over vectors. We then discuss the different classes of expressions tracked by our analysis. Section 3 describes our type based analysis procedure, and discusses its advantages and disadvantages compared to other approaches. Section 4 describes our formally verified normalisation procedure to SMT queries.

2 User Language

Figure 1 shows the grammar of a high-level language for writing specifications about abstract functions. A specification is a sequence of declarations of which there are three types: type synonyms, abstract functions declarations and definitions. Expressions contain many of the usual constructs available in functional languages, as well as sized vector literals with type-safe indexing and first-class universal and existential quantifiers. Due to space limitations it does not include redundant operations such as let-bindings and disjunction. Example 2.1 shows how the specification "function f is monotonically increasing in its second input" can be written in the language.

Example 2.1 (Monotonicity specification).

```
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
```

where `Vec` A n is the type of vectors of length n containing elements of type A and `Index` n is the type of indices into a vector of length n.

A *property* is defined to be any definition whose type is `Bool`, and the goal of the compiler is to compile properties down to a set of queries for a QF_LRA solver. For example, the `monotonic` property might be compiled to the following satisfaction query:

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

where all the variables are implicitly existentially quantified over. The $x0 \ldots x4$ and $y0 \ldots y4$ variables represent the elements of the original quantified vectors x and y respectively. This query is satisfiable if and only if the original `monotonic` property is false.

The language is equipped with a standard semantics that we will describe in Section 4. Here, it suffices to note two key points. Firstly, the language is designed for writing specifications rather than performing general-purpose computation and therefore it does not support recursion. Secondly, the semantics of the declared functions f are external to the specification. In the paper syntax will be written using a type-writer font, e.g. `x == 2`, and semantics using mathematical type-setting, e.g. $x = 2$.

The aim of this paper is to analyse whether a specification is compatible with a quantifier-free real linear arithmetic solver (QF_LRA). This is a property of a specification's semantics rather than syntax, therefore we now classify expressions according to their *semantics*. For the linearity analysis, there are three classes of expressions:

1. **Constant** (C) - the expression's semantics only involves constants, e.g.
$$3^2 + 1 \leq 10$$
2. **Linear** (L) - the expression's semantics uses quantified variables linearly (in the arithmetic sense), e.g.
$$\exists x.\ 0 \leq 2x \land 3x \leq 1$$
3. **Non-linear** (N) - the expression's semantics uses quantified variables non-linearly, e.g.
$$\exists x.\ x^3 + 5x^2 - 1 >= x$$

In the quantifier analysis we identify five classes, which we will call *polarity* classes:

1. **Unquantified** (U) - no quantifiers in the expression's semantics, e.g.
$$1 + 1 \leq 2^2$$
2. **Existentially quantified** ($\exists$) - only existential quantifiers in the expression's semantics, e.g.
$$\exists x.\ f(x) \geq 7$$

$$\begin{aligned}
\langle decl \rangle &::= \texttt{type}\ \langle tId \rangle = \langle type \rangle \\
&\mid\ \texttt{fun}\ \langle id \rangle : \langle type \rangle \\
&\mid\ \langle id \rangle : \langle type \rangle \\
&\quad\ \langle id \rangle\ [\langle binder \rangle] = \langle expr \rangle \\[4pt]
\langle type \rangle &::= \langle type \rangle\ \texttt{->}\ \langle type \rangle \\
&\mid\ \langle tId \rangle \\
&\mid\ \texttt{Vec}\ \langle type \rangle\ n \\
&\mid\ \texttt{Index}\ n \\
&\mid\ \texttt{Bool} \mid \texttt{Real} \\[4pt]
\langle binder \rangle &::= \langle id \rangle \mid (\langle \texttt{id} \rangle : \langle type \rangle)
\end{aligned}$$

$$\begin{aligned}
\langle expr \rangle &::= \langle expr \rangle\ \langle expr \rangle \\
&\mid\ \texttt{lam}\ \langle binder \rangle\texttt{.}\ \langle expr \rangle \\
&\mid\ \texttt{forall}\ \langle binder \rangle\texttt{.}\ \langle expr \rangle \\
&\mid\ \texttt{exists}\ \langle binder \rangle\texttt{.}\ \langle expr \rangle \\
&\mid\ \langle id \rangle \\
&\mid\ \langle expr \rangle\ \langle bop \rangle\ \langle expr \rangle \\
&\mid\ \texttt{if}\ \langle expr \rangle\ \texttt{then}\ \langle expr \rangle\ \texttt{else}\ \langle expr \rangle \\
&\mid\ \texttt{not}\ \langle expr \rangle \\
&\mid\ i \in \mathbb{N} \mid b \in \mathbb{B} \mid r \in \mathbb{R} \\
&\mid\ [\langle expr \rangle, \ldots, \langle expr \rangle] \\[4pt]
\langle bop \rangle &::= \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{and} \mid \texttt{=>} \mid \texttt{+} \mid \texttt{*} \mid \texttt{!}
\end{aligned}$$

Figure 1. The grammar for a high-level user language for writing specifications.

3. **Universally quantified** ($\forall$) - only universal quantifiers in the expression's semantics, e.g.
$$\forall x.\ f(x) \geq 7$$
4. **Parallel quantifiers** (P) - both universal and existential quantifiers present in the expression's semantics, but their scopes never intersect, e.g.
$$(\forall x.\ f(x) >= 0) \wedge (\exists x.\ f(x) \geq 5)$$
5. **Alternating quantifiers** (A) - both universal and existential quantifiers present in the expression's semantics with a non-empty intersection of their scopes, e.g.
$$\forall y.\ \exists x.\ f(x) = y$$

Properties whose expressions are in the U, $\exists$, $\forall$ and P classes may all be checked with a QF_LRA solver. Expressions in $\exists$ are equisatisfiable with a QF_LRA expression. Expressions in $\forall$ may be negated to obtain an expression in $\exists$, and then the desired result is the negation of the output of the solver. Expressions in P may be split into multiple disjoint queries each of which are either in $\exists$ or $\forall$ and may be checked individually.

However, detecting which class a given specification belongs to is not straightforward. For example, negating a Boolean expression will flip the polarity of the quantifiers within the semantics. Therefore although the following only contains `forall` quantifiers syntactically, semantically it has alternating quantifiers and therefore is a member of the A class:

$$\texttt{forall}\ y\texttt{.}\ \texttt{not}\ \texttt{forall}\ x\texttt{.}\ f\ x\ \texttt{!=}\ y$$

Furthermore, the analysis needs to distinguish between quantifiers over variables with finite and infinite domains. For example, despite:

$$\texttt{forall}\ x\texttt{.}\ \texttt{exists}\ (\texttt{i} : \texttt{Index}\ 5)\texttt{.}\ f\ x\ \texttt{!}\ i \texttt{<=} 1$$

containing both a `forall` and an `exists` syntactically, it belongs to the $\forall$ class rather than the A class because the `exists` quantifies over the `Index 5` type and therefore can be expanded out into five disjunctions.

In general, even for linearity, a syntactic check can never be sufficient as shown by the following example function:

$$\texttt{lam}\ (\texttt{x} : \texttt{Real})\texttt{.}\ x * x \texttt{>=} 0$$

whose output is constant if the input is constant and otherwise is non-linear.

3 QF_LRA Analysis

To facilitate the analysis, we now describe an intermediate representation (IR) for the language just defined and an associated type system. The latter is capable of soundly underapproximating membership of QF_LRA, and in the case of a negative result constructing a proof of non-membership which can be turned into readable diagnostic information for the user.

3.1 Intermediate Representation

The grammar for the IR is shown in Figure 2. At the expression level, it has exactly the same expressive power as the high-level language in Section 2.

However, at the type-level it is significantly more complex. As briefly discussed in Section 1.2, our proposed analysis tracks the linearity and polarity of expressions by turning the `Real` and `Bool` into type families indexed by the linearity classes (C, L, N) and polarities classes ($U, \forall, \exists, P, A$) identified in Section 2. The `Real` type is indexed by the linearity annotations and the `Bool` type is indexed by both linearity annotations and polarity annotations. For example `Real` L is the type of real expressions that are linear in its free variables, and `Bool` $\forall$ N is the type of Boolean expression that contain only `forall` quantifiers, at least one of whose quantified variable is used non-linearly.

This naturally leads to a type system that is a variant of System F[1] [13, 22], where types themselves have types, which are referred to as *kinds*. There are four base `Kinds` in our system: `Type`, `Linearity`, `Polarity` and `Size`, and

[1]In practice, a dependently-typed system might provide a simpler metatheory, but we have chosen System F as it is the minimal setting that gives us sufficient expressive power.

```
⟨decl⟩ ::=
  | ⟨tId⟩ : ⟨kind⟩
    ⟨tId⟩ = ⟨type⟩
  | fun ⟨id⟩ : ⟨type⟩
  | ⟨id⟩ : ⟨type⟩
    ⟨id⟩ = ⟨expr⟩

⟨kind⟩ ::= ⟨kind⟩ -> ⟨kind⟩
  | Type
  | Linearity
  | Polarity
  | Size

⟨type⟩ ::= ∀⟨tBinder⟩.⟨type⟩
  | tlam ⟨tBinder⟩.⟨type⟩
  | ⟨type⟩ -> ⟨type⟩
  | {{⟨type⟩}} -> ⟨type⟩
  | ⟨type⟩ ⟨type⟩
  | ⟨tId⟩
  | ⟨builtinType⟩
  | ⟨typeClassConstraint⟩
  | ⟨polarityConstraint⟩
  | ⟨linearityConstraint⟩
  | ⟨linearity⟩
  | ⟨polarity⟩
  | n ∈ ℕ
  | ?m

⟨tBinder⟩ ::= { ⟨tId⟩ : ⟨kind⟩ }

⟨typeClassConstraint⟩ ::=
  | HasEq
  | HasOrd
  | HasForall
  | HasExists
  | HasIndexLiteral n
  | Subtypes

⟨linearity⟩ ::= C | L | N

⟨linearityConstraint⟩ ::=
  | MaxLin
  | MulLin
  | InputLin ⟨id⟩
  | OutputLin ⟨id⟩

⟨polarity⟩ ::= U | ∀ | ∃ | P | A

⟨polarityConstraint⟩ ::=
  | MaxPol
  | NegPol
  | ImpliesPol
  | ForallPol
  | ExistsPol
  | InputPol ⟨id⟩
  | OutputPol ⟨id⟩

⟨builtinType⟩ ::= Vec
  | Index
  | Real
  | Bool

⟨expr⟩ ::=
  | ⟨expr⟩ ⟨arg⟩
  | lam ⟨binder⟩.⟨expr⟩
  | ⟨id⟩
  | ⟨builtin⟩
  | b ∈ 𝔹 | i ∈ ℕ | r ∈ ℝ

⟨binder⟩ ::= ( ⟨id⟩ : ⟨type⟩ )

⟨arg⟩ ::
  | ⟨expr⟩
  | { ⟨type⟩ }

⟨builtin⟩ ::
  | == | != | <= | >= | < | >
  | and | not | => | forall | exists
  | + | * | ! | ite
```

Syntactic sugar
$\forall\{l\}.e \equiv \forall\{l : \texttt{Linearity}\}.e$
$\forall\{p\}.e \equiv \forall\{p : \texttt{Polarity}\}.e$
$\forall\{\tau\}.e \equiv \forall\{\tau : \texttt{Type}\}.e$
$\forall\{xy\}.e \equiv \forall\{x\}.\forall\{y\}.e$

Figure 2. Intermediate representation for the language in Figure 1.

new Kinds can be constructed from these using the function arrow. For example, the Bool type has kind:

```
Linearity -> Polarity -> Type
```

Types can also contain type-level variables. A new type-level variable τ which has kind k can be abstracted over using the syntax '$\forall\{\tau : k\}.\ldots$', commonly known as a *pi-binder*. As discussed in Section 1, one of our aims is to avoid the user having to write additional typing information. Instead missing types will be represented as *meta-variables* and written as $?m$ where m is a unique identifying number. Meta-variables are inserted by the elaborator and type-checker, and may represent type variables of any Kind.

In addition to the standard function type, τ_1 -> τ_2, the IR also has the *instance* function type, $\{\{\tau_1\}\}$ -> τ_2. This type enforces that the constraint τ_1 needs to be resolved in order to obtain something of type τ_2. There are three classes of constraints in our system. The first is a set of type-classes that allow operators in the user language to be overloaded for multiple different types. For example, a HasEq constraint will be generated whenever an equality comparison operator is encountered, to enforce that the arguments are of a comparable type. The second and third groups of constraints exist to encode the relationships between linearity and polarity meta-variables. These will be generated by expressions in the user language that alter the polarity or linearity of an expression (e.g. a MulLin constraint is generated by a multiplication) or an event we would like to appear in the error messages (e.g. a InputLin constraint is generated when something is used as an input to a user-defined function).

3.2 Elaboration

Elaboration from the user language to the intermediate language is straightforward. Expression binders with a missing type get assigned a fresh meta-variable for their type. The builtin infix operations (+, *, if etc.) are turned into prefix operators. The quantifiers are changed from binding variables directly to builtins that take a lambda as their only argument, e.g. forall x. e is elaborated to forall (lam x . e). Types are elaborated directly, and, with the exception of meta-variables, none of the additional type-level constructs introduced in the IR get added to the type signatures during elaboration.

3.3 Type Checking

Type-checking of a specification proceeds on a declaration-by-declaration basis. For each declaration:

1. Fresh meta-variables for missing linearity and polarity type indices are inserted.
2. A standard bidirectional type checking/inference pass is made over the declaration type and body, generating new meta-variables and constraints.
3. An attempt is made to solve the meta-variables and constraints generated in the previous step.
4. Constraints that track the application of user functions are introduced to the set of unsolved constraints.
5. The declaration is generalised over by prepending unsolved meta-variables and constraints to the declaration's type signature.

We will now describe these steps in more detail.

3.3.1 Inserting Linearity and Polarity Meta-variables. By design, the user is unaware of the linearity and polarity annotations. Consequently after elaboration, the user's program will be ill-typed. Therefore the first phase of type-checking is to traverse all the types in the current declaration, inserting meta-variables into the indexed types. The two main indexed types are `Bool` and `Real`, however user-defined type synonyms that make use of `Bool` and `Real` will also be indexed by polarity and linearity annotations.

For example, consider the first two lines of the *monotonicity* specification in Example 2.1:

```
type Input = Vec Real 5

fun f : Input -> Real
```

Assume that the type-synonym `Input` has already been type-checked and generalised over to obtain:

`Input` : $\forall\{i:$ `Linearity`$\}$.`Type`
`Input` = `tlam` $\{l:$ `Linearity`$\}$.`Vec` (`Real` l) 5

and that we are type-checking the declaration `fun f`. Then inserting fresh meta-variables ?0 and ?1 in f for the missing linearity types results in:

`fun` f : `Input ?0 -> Real ?1`

3.3.2 Bidirectional Type Checking. A second pass is then made over the declaration, using a bidirectional type-checking algorithm [9]. We follow the lead of Agda [21] and Idris [7] by adapting it to insert a fresh meta-variable wherever a pi-binder is encountered, and wherever we find an instance argument we add the constraint contained within it to the set of constraints. The pi-binders can therefore be thought of as the mechanism for introducing new unknown types and the constraints encode the relationship between them, and hence provide information for how they should be solved. Note that one of the strengths of our analysis is that only the inference typing rules for the builtins and literals are non-standard and the core bidirectional algorithm remains unaltered. Correspondingly, we don't reproduce the full algorithm here.

Figure 3 shows the inference typing rules for the type and expression builtins. `Real` and `Bool` are now indexed families of types. The `Vec` and `Index` types are typed as expected.

Every builtin operation, with the exception of the lookup operation, has a constraint associated with it, and can be divided into two categories. The first are builtins that don't need to be overloaded and therefore the polarity and linearity constraints appear directly in their type. For example, `+` takes two `Real`s of linearity l_1 and l_2 and returns a third `Real` with linearity l_3, assuming that the constraint `MaxLin` l_1 l_2 l_3 is satisfied (i.e. l_3 is the maximum of l_1 and l_2). The second are builtins that are overloaded, in which case their type-class constraints will first resolve the type, and only then add linearity and polarity constraints if required (see Section 3.3.3). For example `forall` generates the `HasForall` type-class constraint.

The typing of literals are shown in Figure 4. Real number literals are given the constant linearity annotation (C), and Boolean literals are given constant linearity (C) and unquantified polarity (U) annotations. Index literals require a constraint to enforce that its value is smaller than the size of the resulting type. The most interesting rule case is vector literals. Although the types of the elements of a vector need to be homogeneous in the high-level language, in the IR they may be heterogeneous. For example, the first component may be a `Real` L, and the second a `Real` C. This problem is solved by encoding an n-ary subtyping relation via the `Subtypes` constraint.

3.3.3 Constraint Solving. The next step is to try to solve the set of constraints generated in the bidirectional phase. Each constraint is tried in turn, and is either a) returned to the set of constraints if it is currently blocked by an unsolved meta variable, b) removed from the set of constraints if it can be solved, the process of which may generate solutions for meta-variables and new (smaller) constraints, or c) marked as failed if it is provably unsolvable (in which case type-checking itself will fail). There are three different types of constraints that can be generated.

Unification constraints. Unification constraints are of the form $\tau_1 \sim \tau_2$ and assert that types τ_1 and τ_2 are equal. They can be generated both by the bidirectional pass and when solving type-class and polarity/linearity constraints. They are solved using a standard pattern unification algorithm, which may generate further unification constraints involving strictly syntactically smaller terms than the original two terms. For more details of the algorithm see the presentation by Nipkow [20].

Type-class constraints. Type-class constraints are used to resolve overloaded operators. Each type-class constraint

$$\texttt{Bool} : \texttt{Linearity -> Polarity -> Type} \qquad \texttt{Real} : \texttt{Linearity -> Type}$$

$$\texttt{Vec} : \texttt{Type -> Size -> Type} \qquad \texttt{Index} : \texttt{Size -> Type} \qquad \texttt{!} : \forall\{\tau n\}.\,\texttt{Vec}\,\tau\,n\ \texttt{->}\ \texttt{Index}\,n\ \texttt{->}\ \tau$$

$$\texttt{+} : \forall\{l_1 l_2 l_3\}.\,\{\{\texttt{MaxLin}\,l_1\,l_2\,l_3\}\}\ \texttt{->}\ \texttt{Real}\,l_1\ \texttt{->}\ \texttt{Real}\,l_2\ \texttt{->}\ \texttt{Real}\,l_3$$

$$\texttt{*} : \forall\{l_1 l_2 l_3\}.\,\{\{\texttt{MulLin}\,l_1\,l_2\,l_3\}\}\ \texttt{->}\ \texttt{Real}\,l_1\ \texttt{->}\ \texttt{Real}\,l_2\ \texttt{->}\ \texttt{Real}\,l_3$$

$$\texttt{and} : \forall\{l_1 l_2 l_3 p_1 p_2 p_3\}.\,\{\{\texttt{MaxLin}\,l_1\,l_2\,l_3\}\}\ \texttt{->}\ \{\{\texttt{MaxPol}\,p_1\,p_2\,p_3\}\}\ \texttt{->}\ \texttt{Bool}\,l_1\,p_1\ \texttt{->}\ \texttt{Bool}\,l_2\,p_2\ \texttt{->}\ \texttt{Bool}\,l_3\,p_3$$

$$\texttt{not} : \forall\{l p_1 p_2\}.\,\{\{\texttt{NegPol}\,p_1\,p_2\}\}\ \texttt{->}\ \texttt{Bool}\,l\,p_1\ \texttt{->}\ \texttt{Bool}\,l\,p_2$$

$$\texttt{=>} : \forall\{l_1 l_2 l_3 p_1 p_2 p_3\}.\,\{\{\texttt{MaxLin}\,l_1\,l_2\,l_3\}\}\ \texttt{->}\ \{\{\texttt{ImpliesPol}\,p_1\,p_2\,p_3\}\}\ \texttt{->}\ \texttt{Bool}\,l_1\,p_1\ \texttt{->}\ \texttt{Bool}\,l_2\,p_2\ \texttt{->}\ \texttt{Bool}\,l_3\,p_3$$

$$\texttt{==}, \texttt{!=} : \forall\{\tau_1\tau_2\tau_3\}.\,\{\{\texttt{HasEq}\,\tau_1\,\tau_2\,\tau_3\}\}\ \texttt{->}\ \tau_1\ \texttt{->}\ \tau_2\ \texttt{->}\ \tau_3$$

$$\texttt{<=}, \texttt{>=}, \texttt{<}, \texttt{>} : \forall\{\tau_1\tau_2\tau_3\}.\,\{\{\texttt{HasOrd}\,\tau_1\,\tau_2\,\tau_3\}\}\ \texttt{->}\ \tau_1\ \texttt{->}\ \tau_2\ \texttt{->}\ \tau_3$$

$$\texttt{forall} : \forall\{\tau_1\tau_2\}.\,\{\{\texttt{HasForall}\,\tau_1\,\tau_2\}\}\ \texttt{->}\ \tau_1\ \texttt{->}\ \tau_2 \qquad \texttt{exists} : \forall\{\tau_1\tau_2\}.\,\{\{\texttt{HasExists}\,\tau_1\,\tau_2\}\}\ \texttt{->}\ \tau_1\ \texttt{->}\ \tau_2$$

$$\texttt{ite} : \forall\{l p \tau_1\tau_2\tau_3\}.\,\{\{\texttt{Subtypes}\,\tau_3\,[\tau_1,\tau_2]\}\}\ \texttt{->}\ \texttt{Bool}\,L\,U\ \texttt{->}\ \tau_1\ \texttt{->}\ \tau_2\ \texttt{->}\ \tau_3$$

Figure 3. Type system for the builtin constants in the intermediate representation.

$$\text{(index)}\frac{i \in \mathbb{N}}{\Gamma \vdash i : \forall\{\tau\}.\,\{\{\texttt{HasIndexLiteral}\,i\,\tau\}\}\ \texttt{->}\ \tau} \qquad \text{(real)}\frac{r \in \mathbb{R}}{\Gamma \vdash r : \texttt{Real}\,C}$$

$$\text{(vecLit)}\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash [e_1 \ldots e_n] : \forall\{\tau\}.\,\{\{\texttt{Subtypes}\,\tau\,[\tau_1,\ldots,\tau_n]\}\}\ \texttt{->}\ \texttt{Vec}\,\tau\,n} \qquad \text{(bool)}\frac{b \in \mathbb{B}}{\Gamma \vdash b : \texttt{Bool}\,C\,U}$$

Figure 4. Type system for the literal expressions in the intermediate representation.

$$\text{(hasForallReal)}\frac{l_1 \sim L \quad l_2, p_1, p_2 = ?^+ \quad \tau_1 \sim \texttt{Bool}\,l_2\,p_1 \quad \tau_2 \sim \texttt{Bool}\,l_2\,p_2 \quad \texttt{ForallPol}\,p_1\,p_2}{\texttt{HasForall}\ (\texttt{Real}\,l_1\ \texttt{->}\ \tau_1)\ \tau_2}$$

$$\text{(hasForallVec)}\frac{\texttt{HasForall}\ (\tau_1\ \texttt{->}\ \tau_2)\ \tau_3}{\texttt{HasForall}\ (\texttt{Vec}\,n\,\tau_1\ \texttt{->}\ \tau_2)\ \tau_3} \qquad \text{(hasForallIndex)}\frac{l, p = ?^+ \quad \tau_1 \sim \texttt{Bool}\,l\,p \quad \tau_2 \sim \texttt{Bool}\,l\,p}{\texttt{HasForall}\ (\texttt{Index}\,n\ \texttt{->}\ \tau_1)\ \tau_2}$$

Figure 5. Rules for solving the `HasForall` type-class constraint

has a set of rules, which are tried in order. For example, Figure 5 shows the rules for the `HasForall` type-class. These rules allows users to quantify over `Real`, `Index` and arbitrarily nested `Vec`s of `Real` and `Index`.

Informally, the first rule *hasForallReal* can be read as follows: when quantifying over a `Real` with linearity l_1 then i) unify l_1 with L as any quantified variable is linear with respect to itself, ii) generate one new linearity meta-variable l_2 and two new polarity meta-variables p_1, p_2, iii) unify both the output type of the function and the output type of the quantifier with `Bool` types of linearity l_2 and polarities p_1 and p_2, thereby ensuring that linearity is preserved iv) constrain p_1 and p_2 to be related by the `ForallPol` constraint.

`MaxLin` $l_1\ l_2\ l_3$
Derived as the least upper bound ($\sqcup$) operator from the lattice to the right.

$$\begin{tikzcd}
N \text{ (NonLinear)} \\
L \text{ (Linear)} \arrow[u] \\
C \text{ (Constant)} \arrow[u]
\end{tikzcd}$$

`ExistsPol` $p_1\ p_2$		
p_1	$\rightarrow$	p_2
U	$\rightarrow$	$\exists$
$\exists$	$\rightarrow$	$\exists$
$\forall$	$\rightarrow$	A
P	$\rightarrow$	A
A	$\rightarrow$	A

`MaxPol` $p_1 p_2 p_3$
Derived as the least upper bound ($\sqcup$) operator from the lattice to the right.

$$\begin{tikzcd}
& A \text{ (Alternating)} & \\
& P \text{ (Parallel)} \arrow[u] & \\
\forall \text{ (Forall)} \arrow[ur] & & \exists \text{ (Exists)} \arrow[ul] \\
& U \text{ (Unquantified)} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

`MulLin` $l_1\ l_2\ l_3$			
l_1	l_2	$\rightarrow$	l_3
C	l	$\rightarrow$	l
l	C	$\rightarrow$	l
L	L	$\rightarrow$	N
N	l	$\rightarrow$	N
l	N	$\rightarrow$	N

`ForallPol` $p_1\ p_2$		
p_1	$\rightarrow$	p_2
U	$\rightarrow$	$\forall$
$\exists$	$\rightarrow$	A
$\forall$	$\rightarrow$	$\forall$
P	$\rightarrow$	A
A	$\rightarrow$	A

`NegPol` $p_1\ p_2$		
p_1	$\rightarrow$	p_2
U	$\rightarrow$	U
$\exists$	$\rightarrow$	$\forall$
$\forall$	$\rightarrow$	$\exists$
P	$\rightarrow$	P
A	$\rightarrow$	A

`ImpliesPol` $p_1\ p_2\ p_3$
`NegPol` $p_1\ p_1'$ $\wedge$ `MaxPol` $p_1'\ p_2\ p_3$

Figure 6. Solutions for linearity and polarity constraints which alter their arguments.

Note that the use of type-class constraints gives us the flexibility to distinguish between finite and infinite quantifiers, as discussed at the end of Section 2. In particular, when solving the same constraint for the `Index` type the polarity is preserved without change. The rule for `Vec` ensures that this behaviour is inherited correctly depending on the vector's element type. The full set of rules for solving type-class constraints can be found in Appendix B.

Linearity/polarity constraints. The third type of constraints are those that operate directly on the linearity and polarity types. Polarity and linearity types flow upwards through the AST, from the constants and variables at its leaves, through the nodes of interest. Unlike type-class constraints, linearity and polarity constraints encode total functions, i.e. the last type argument in each constraint is deterministically derivable from the previous type arguments. Therefore they can never fail, although they may still temporarily block on unsolved meta-variables.

The solutions for the linearity and polarity constraints which alter their arguments are shown in Figure 6. For example if expression e_1 has polarity $\exists$ and e_2 has polarity $\forall$ then 'e_1 `and` e_2' would generate the `MaxPol` constraint, and would therefore be assigned polarity P. Implication is an interesting case, as `ImpliesPol` can be defined in terms of `NegPol` and `MaxPol`. However, we keep `ImpliesPol` as a separate constraint so that we can distinguish between a quantifier that was negated by a '`not`' and one that was negated by being on the LHS of an '`=>`' in the error messages.

3.3.4 Function Application Constraint Insertion.

An observant reader may have noted that so far the constraints `InputPol`, `OutputPol`, `InputLin`, `OutputLin` in the grammar in Figure 2 have neither shown up in the types in Figures 3 & 4 nor in the constraint solutions in Figure 6. Their purpose is purely to track calls to user definitions so that the calls will show up in the error messages (e.g. line 2. in the example error message in Section 1.2). They therefore act as the identity function on linearity and polarity types as they have been presented so far.

How do they work then? Our approach, as presented so far, indexes every `Bool` with a linearity and polarity type. However, in Section 1.2 we promised constructive proofs. Therefore, in the same way that a compiler annotates nodes in the abstract syntax tree of an expression with their provenance in the source file, we annotate the linearity and polarity types with a proof of how that value was constructed. The grammar for these proofs terms is shown in Figure 7. For example, the non-linear type N has two pieces of provenance information attached, one each for the left and right sides of the non-linear multiplication respectively.

The constraints added during the bidirectional pass inherit the provenance of the operation that generated them. Therefore when solving a constraint, the name and provenance of the symbol that generated the constraint is available to be added to the constructed polarity or linearity type. When an `InputPol`/`OutputPol`/`InputLin` /`OutputLin` constraint is solved it leaves the actual linearity or polarity type untouched but appends a `FunctionInput`/`FunctionOutput` node to the provenance information associated with it.

The remaining issue is how to ensure these constraints are generated in the right place. In particular they need to be generated when the user function is applied, not when it is defined. The solution is to have them inserted into the

$\langle prov \rangle$ ::= location in source file

$\langle linearity \rangle$ ::= C
  | $L\langle linearityProv \rangle$
  | $N\langle linearityProv \rangle\ \langle linearityProv \rangle$

$\langle linearityProv \rangle$ ::=
  | QuantifiedVariable $\langle prov \rangle\ \langle id \rangle$
  | FunctionInput $\langle prov \rangle\ \langle linearityProv \rangle$
  | FunctionOutput $\langle prov \rangle\ \langle linearityProv \rangle$

$\langle polarity \rangle$ ::= U
  | $\forall\ \langle polarityProv \rangle$
  | $\exists\ \langle polarityProv \rangle$
  | $P\ \langle polarityProv \rangle\ \langle polarityProv \rangle$
  | $A\ \langle prov \rangle\ \langle polarityProv \rangle$

$\langle polarityProv \rangle$ ::=
  | Quantifier $\langle prov \rangle\ \langle id \rangle$
  | Negation $\langle prov \rangle\ \langle polarityProv \rangle$
  | FunctionInput $\langle prov \rangle\ \langle polarityProv \rangle$
  | FunctionOutput $\langle prov \rangle\ \langle polarityProv \rangle$

Figure 7. Grammar for the internal proof structure of the linearity and polarity constraints. This is a refinement of the $\langle linearity \rangle$ and $\langle polarity \rangle$ classes in Figure 2.

type signature of user definitions *after* constraint solving. Concretely, we again iterate through all the unique polarity and linearity values in the input and output types of the type signature. For every value, we replace it with a fresh meta-variable and then link the old value and the new meta-variable by the appropriate constraint. For example, suppose we have the following type after constraint solving:

```
f : Real L -> Bool L ∀
```

then after this operation we will have:

```
f : Real ?0 -> Bool ?1 ?2
```

and the following additional three constraints:

```
InputLin f L ?0   OutputLin f L ?1   OutputPol f ∀ ?2
```

3.3.5 Generalisation. The final generalisation phase takes all the unsolved meta-variables and constraints and prepends them to the type signature. To explore why some constraints and meta-variables may be unsolved (excluding those introduced for input and output constraints in the previous step) consider the following (not very useful) definition:

```
g : Bool -> Bool
g = not
```

At the end of the constraint solving phase, the type-checker will be in the following state:

```
g : Bool ?0 ?1 -> Bool ?0 ?2        Constraints
g x = not x                          NegPol ?1 ?2
```

None of the meta-variables nor the NegPol constraint itself can be solved as the type-checker doesn't know what the input linearity and polarity annotations are. In order to obtain the most general expression, we follow the constraint propagation approach inspired by the implicit generalisation feature of Idris [7]. First of all any unsolved constraints are appended to the front of the type:

```
g : {{NegPol ?1 ?2}} -> Bool ?0 ?1 -> Bool ?0 ?2
g x = not x
```

Next we replace each unsolved meta with a new universally quantified type-variable:

```
g : ∀{l p1 p2}.
    {{NegPol p1 p2}} -> Bool l p1 -> Bool l p2
g x = not x
```

to result in the final function. It should be no surprise that in this case we have re-obtained the original type of not from Section 3.3.2.

Of course, in the above example we have been ignoring the function input and output constraints added in the previous phase, so the true final type is:

```
g : ∀{l1 l2 l3 p1 p2 p3 p4}.
   {{InputLin g l1 l2}} -> {{InputPol g p1 p2}} ->
   {{OutputLin g l2 l3}} -> {{OutputPol g p3 p4}} ->
   {{NegPol p2 p3}} -> Bool l2 p2 -> Bool l2 p3
g x = not x
```

3.4 Worked Example

To illustrate all five stages of type-checking, we take the equalExceptAt declaration from Example 2.1:

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

Note that for clarity, we use the high-level syntax rather than the elaborated intermediate representation. Under the assumption that the type-synonym Input has already been generalised as described in Section 3.3.1, then the first step is to insert meta-variables for the missing linearity and polarity annotations in the user's types as follows:

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

In this case, the function body is not modified at this stage, but if it had contained user supplied type annotations, e.g., in binders, then these would also have had fresh meta-variables inserted.

The second step is the bidirectional type-checking pass. The types that are assigned to the subterms of interest during this phase are as follows:

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

During the bidirectional pass, there are four operations which generate constraints. These constraints are then solved in the third stage of type-checking as follows:

$$\begin{tikzcd}
\textcircled{2} & & \texttt{HasEq (Index 5) (Index 5) (Bool ?6 ?8)} \arrow[dl] \arrow[dr] & \\
& ?6 \sim C & & ?8 \sim U \arrow[dddd, bend left] \\
\textcircled{4} & & \texttt{HasEq (Real ?0) (Real ?1) (Bool ?7 ?9)} \arrow[dl] \arrow[dr] & \\
& \texttt{MaxLin ?0 ?1 ?7} & & ?9 \sim U \arrow[dd, bend left] \\
\textcircled{3} & \texttt{MaxLin ?6 ?7 ?4} & & \texttt{ImpliesPol ?8 ?9 ?5} \arrow[d] \\
& & & ?5 \sim U \\
\textcircled{1} & & \texttt{HasForall (Index 5 -> Bool ?4 ?5) (Bool ?2 ?3)} \arrow[dl] \arrow[dr] & \\
& ?4 \sim ?2 & & ?5 \sim ?3
\end{tikzcd}$$

At the end of this stage, the following meta-variables and constraints remain unsolved:

$$\begin{array}{ll}
\text{Meta-variables :} & \{?0, ?1, ?4, ?7\} \\
\text{Constraints :} & \{\texttt{MaxLin ?0 ?1 ?7},\ \texttt{MaxLin}\ C\ \texttt{?7 ?2}\} \\
\text{Type :} & \texttt{Index 5 -> Input ?0 ->} \\
& \texttt{Input ?1 -> Bool ?2}\ U
\end{array}$$

The polarity of the output can be resolved to U, as there are no quantifiers with infinite domains. However, the linearity of the output, ?2, can't be directly solved for because the linearity of the inputs ?0 and ?1 are unknown. The unsolved constraints encode how these three unknowns are related. Note that if we added an extra rule that encodes the fact that C is the bottom element of the total order, we could also get $\texttt{MaxLin}\ C\ ?7\ ?2$ to reduce to the constraint $?7 \sim ?2$. However, this optimisation is unnecessary for the correctness of the algorithm.

In the fourth stage of type-checking, we generate fresh meta-variables for each outstanding polarity and linearity meta-variable in the type, and link them to the prior unsolved meta-variables via additional function input and output constraints. Concretely fresh meta-variables ?10, ?11, ?12 are introduced for ?0, ?1, ?2 respectively:

$$\begin{array}{ll}
\text{Meta-variables :} & \{?0, ?1, ?2, ?7, ?10, ?11, ?12\} \\
\text{Constraints :} & \{\texttt{MaxLin ?0 ?1 ?7},\ \texttt{MaxLin}\ C\ \texttt{?7 ?2} \\
& \texttt{InputLin equalExceptAt ?10 ?0} \\
& \texttt{InputLin equalExceptAt ?11 ?1} \\
& \texttt{OutputLin equalExceptAt ?2 ?12}\} \\
\text{Type :} & \texttt{Index 5 -> Input ?10 ->} \\
& \texttt{Input ?11 -> Bool ?12}\ U
\end{array}$$

Finally, in the fifth stage, the final type of the declaration is obtained by generalising over all the unsolved constraints and meta-variables:

$$\begin{array}{l}
\texttt{equalExceptAt} : \forall \{l_1\ l_2\ l_3\ l_3\ l_4\ l_5\ l_6\ l_7\}\,. \\
\quad \texttt{\{\{InputLin equalExceptAt}\ l_5\ l_1\texttt{\}\} ->} \\
\quad \texttt{\{\{InputLin equalExceptAt}\ l_6\ l_2\texttt{\}\} ->} \\
\quad \texttt{\{\{OutputLin equalExceptAt}\ l_3\ l_7\texttt{\}\} ->} \\
\quad \texttt{\{\{MaxLin}\ l_1\ l_2\ l_4\texttt{\}\} -> \{\{MaxLin}\ C\ l_4\ l_3\texttt{\}\} ->} \\
\quad \texttt{Index 5 -> Input}\ l_5\ \texttt{->} \\
\quad \texttt{Input}\ l_6\ \texttt{-> Bool}\ l_7\ U \\
\texttt{equalExceptAt i x y = forall}\ j\,.\ i\ \texttt{!=}\ j\ \texttt{=>}\ x\ \texttt{!}\ j\ \texttt{==}\ y\ \texttt{!}\ j
\end{array}$$

Although this type looks scary, once the function is applied and values are assigned to l_1 and l_2 then all the constraints unravel to compute an output linearity for l_7. For example if $l_1 = C$ and $l_2 = L$ then $l_7 = L$ as expected.

3.5 Using the Analysis

Once the entire program has been type-checked then the types can be used to decide whether or not a specification is compilable to QF_LRA. In particular, any property with the type $\texttt{Bool}\ l\ p$ where $l \neq N$ and $p \neq A$ may be compiled. In Section 4 we will show that this is sound. On the other-hand if $l = N$ or $p = A$ then the constructed provenance attached to the N or A types, as described in Section 3.3.4, may be used to construct a suitable error message for the user.

However, as with any type-system that seeks to analyse semantic properties of a program, the analysis is inherently incomplete. For example, the following will be labelled as non-linear even though the actual result is constant:

$$\texttt{forall}\ x\,.\ [x * x, 0]\ \texttt{!}\ 1 \texttt{<=} 1$$

Appendix C contains a more complete list of the sources of incompleteness.

One consequence of the incompleteness is that if the analysis is used before normalisation then there exist specifications that belong to QF_LRA that our analysis will incorrectly identify as not belonging to it. There are two different philosophical approaches to this. The first, hard-line, approach is that users should not be writing such specifications. They are inherently fragile, and as our language is strictly more expressive than the language of the underlying solver, such a specification can always be rewritten to a form such that the analysis succeeds.

The second, more forgiving, approach is to use the analysis as a diagnostic procedure and only perform it on the original specification *after* normalisation has encountered a problem. This means that no valid specifications will be rejected, but at the cost that the error messages will sometimes be inaccurate.

3.6 A Possible Alternative Approach

In Section 1.1, we mentioned how Z3 and other tools detect such linearity and polarity problems by first normalising the specification. The main advantage of the normalisation approach is that, unlike ours, it is complete. However, as discussed in Section 1.1, it discards much of the structure present in the user specification, without which it is impossible to recreate useful error messages.

One obvious modification that would allow it to generate useful error messages, while retaining completeness, would be to augment each node in the normalised AST with the history of how the node was generated. This is roughly equivalent to dynamically computing the proof term in Figure 7 that our approach computes statically. When a problem was encountered, this data could then be used to reconstruct a useful error message.

However, there we argue there are still two main drawbacks to this approach. Firstly, storing the computation history for each node would have a memory requirement that would be at least linear in the length of the execution path. We hypothesise that this would be prohibitive for very large specifications with significant amounts of intermediate computation. In contrast, our type-based approaches' memory requirements are proportional to the size of the syntax of the user's program rather than the length of the execution paths.

Secondly, the normalisation approach is not compositional, i.e. it requires the whole program to be available to be run and normalised. Therefore, it cannot be used to identify linearity and polarity problems in incomplete programs or external libraries. In contrast, type-based algorithms are inherently compositional. Therefore, our analysis can be run on incomplete programs and external libraries can be checked for problems independently of the user's own code.

4 SMT Translation by Evaluation

We have described how to use types to identify specifications that can be translated to SMT solvers that handle linear constraints and uninterpreted functions. We complete the journey by using the inferred typing information as input for a Normalisation by Evaluation (NbE) procedure that turns a higher-order specification from Section 3 into input suitable for an SMT solver. The results of this section have been formalised in Agda[2]. To simplify our development, we assume that all higher kinded types have been fully applied (this is guaranteed by the elaboration process) and that we only deal with queries in existential form (universal and parallel queries would be minor extensions).

Due to the relatively high complexity of the IR — indexed types, polymorphism, arrays, and higher-order functions — we use Categorical Logic techniques to build three models that constitute our normalisation procedure and correctness proof. The first model in Section 4.2 defines the "standard semantics" of IR, i.e., where the `Real` type is interpreted as rational numbers, unquantified Booleans are interpreted as Booleans, and quantified Booleans are interpreted as Agda Sets. The second model in Section 4.3 defines a semi-syntactic "normalising" model, where linear `Reals` are interpreted as linear expressions, unquantified Booleans as constraint expressions, and quantified Booleans as logical formulas with quantifiers. Interpreting a specification in the normalising model, and evaluating it inside Agda, yields a logical formula suitable for an SMT solver, exploiting typing information to ensure that normalisation is always possible.

The key challenge in constructing the normalising model is to handle the two tricky features of the language that make normalisation non local: the polymorphic if-then-else, and the separation of those constraints involving function terms and those containing linear arithmetic. Translation of these features requires them to be "lifted" out of their context and translated into additional constraints in the final query. We accomplish this via a specially designed monad.

The correctness criterion for our normalisation procedure is that the standard semantics of a specification t agrees with the semantics of the normalised form of t, in the sense that they are equi-inhabited Agda Sets. This means that if the SMT solver a model for the existential quantifiers in a query, then it is possible to translate these back to the quantifiers in the original specification.

To prove correctness, we define a third model of the language that relates the standard and normalising semantics. This is effectively a Kripke logical relation between the two models, defined as a model itself. Our use of categorical logic to construct this model means that most of the effort is concentrated on the specifics of dealing with linear arithmetic and logical constraints with quantification.

4.1 Representing the Syntax

We represent the syntax of the IR in Agda using an intrinsically typed representation [1, 4, 6]. This means that we define a datatype $\Delta \mid \Gamma \vdash A$ of well-typed terms, where Δ is a kinding context assigning kinds to type level variables, Γ is a typing context assigning types to term level variables, and A is the result type. The constructors of this type include the standard typed λ-calculus rules for type and term level abstraction and incorporate the typing rules given in Figures 3 and 4. Variables, at both the type and term level are represented using de Bruijn indices. We do not require

[2]Agda code is available at https://github.com/vehicle-lang/vehicle-formalisation. We assume functional extensionality as an axiom.

renaming and substitution at the term level, but we do at the type level in order to interpret universal quantification.

4.2 Standard Interpretation

The "standard interpretation" of the IR interprets the calculus as if it were making statements directly in mathematics. We interpret types as sets and terms as functions from the interpretation of contexts to the interpretation of the result type. We interpret `Real` l for all l as the set of rational numbers and the operations of addition and multiplication as the actual operations. Due to our use of Agda as our metatheory, we are led to be a little more refined in our interpretation of `Bool`. For `Bool` $U\ l$, we use the set of Boolean values. Using Booleans allows us to interpret the `if then else` construct as an actual if-then-else. For all other polarities p, `Bool` $p\ l$, the natural interpretation would be to interpret the type as an Agda Set and interpret universal and existential quantification using Agda's Π- and Σ-types. However, this leads to a universe level mismatch with the interpretation of numbers and unquantified Booleans (rationals and Booleans are at level 0, while Set is at level 1) which requires noisy explicit lifting the interpretation. Therefore, we interpret Booleans with quantifier polarity using codes for the quantifiers and translate into Set for complete programs.

The standard semantics defines a function from closed terms of Boolean type to sets, parameterised by the interpretation of any uninterpreted functions in the term. The informational content of these sets is the values in the quantifiers used in the specification term.

$$\mathcal{S}[\![-]\!] : (\mathbb{Q} \to \mathbb{Q}) \to\ \vdash \texttt{Bool}\, l\, p \to \text{Set}$$

4.3 Normalising Semantics

The key idea behind NbE, in general, is to provide a semantics in the syntax of normal forms, rather the "intended" meaning of the calculus. In this case, we interpret the `Real` type as linear expressions and the `Bool` type as constraint expressions, then, and interpret all the operations of the calculus as ones that manipulate expressions in normal form simulating the intended meaning. Consequently, evaluating a closed term $\vdash t : \texttt{Bool}\, L\, \exists$ will yield a normal form constraint expression suitable for a solver backend which is guaranteed to have the same meaning as t. This technique as similar to the use of "smart constructors" for maintaining normal forms in typed functional programming.

4.3.1 Context Indexed Sets.

Normal form expressions' free variables may not be the same as those of source language expressions. For example, a source language variable of type `Real` C will be a rational value, not a variable, and normal form expressions will only ever have rational-valued variables. To track free variables in normalised terms, we use linear variable contexts defined by the following grammar:

$$\text{LinVarCtxt} \ni \Delta ::= \varepsilon \mid \Delta, \cdot$$

References to variables in a linear variable context are represented as de Bruijn indices counting into the context:

$$\begin{array}{l} \textbf{data}\ \text{Var} : \text{LinVarCtxt} \to \text{Set}\ \textbf{where} \\ \quad \text{zero} : \text{Var}\,(\Delta, \cdot) \\ \quad \text{succ} : \text{Var}\,\Delta \to \text{Var}\,(\Delta, \cdot) \end{array}$$

Normalised syntax will seldom be used in the context where it is created. To move syntax between contexts, we rename the variables in them. A renaming from Δ_1 to Δ_2 is a mapping of variables in Δ_2 to variables in Δ_1: $\Delta_1 \Rightarrow_r \Delta_2 = (\text{Var}\,\Delta_2 \to \text{Var}\,\Delta_1)$. Each of the different kinds of syntax used in the normal forms must be *renamable*, meaning that if A : LinVarCtxt → Set is a set indexed by linear variable contexts, and $\Delta_1 \Rightarrow_r \Delta_2$ is a renaming, then there is a function from $A\,\Delta_2$ to $A\,\Delta_1$ (note that renaming is contravariant, because it acts on contexts).

4.3.2 Normal Forms.

We use the following Agda data type to represent linear arithmetic expressions in normal form. This type is indexed by a linear variable context Δ that controls what variables may appear. The form of the data type ensures that all LinExp Δ terms consist of trees of terms added together, with constants or variables multiplied by constants at the leaves.

$$\begin{array}{l} \textbf{data}\ \text{LinExp}\,(\Delta : \text{LinVarCtxt}) : \text{Set}\ \textbf{where} \\ \quad \text{`const} : \mathbb{Q} \to \text{LinExp}\,\Delta \\ \quad \text{`var} \quad : \mathbb{Q} \to \text{Var}\,\Delta \to \text{LinExp}\,\Delta \\ \quad \_\text{`}+\text{`}\_ : \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \end{array}$$

We adopt a convention of using backticks ` to mark constructors used for normal forms. Constraint (unquantified Boolean) expressions are represented by another data type:

$$\begin{array}{l} \textbf{data}\ \text{Constraint}\,(\Delta : \text{LinVarCtxt}) : \text{Set}\ \textbf{where} \\ \quad \_\text{`}\leq\text{`}\_ \ : \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`}>\text{`}\_ \ : \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`}=\text{`}\_ \ : \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`}\neq\text{`}\_ \ : \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`}=\text{`f}\_ : \text{Var}\,\Delta \to \text{Var}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`}\neq\text{`f}\_ : \text{Var}\,\Delta \to \text{Var}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`and`}\_ : \text{Constraint}\,\Delta \to \text{Constraint}\,\Delta \to \text{Constraint}\,\Delta \\ \quad \_\text{`or`}\_ \ : \text{Constraint}\,\Delta \to \text{Constraint}\,\Delta \to \text{Constraint}\,\Delta \end{array}$$

The first four constructors represent equality, disequality, and inequality constraints between linear expressions. The fifth and sixth constructors represent equality and inequality constraints between variables and the uninterpreted function applied to variables. We separate these two kinds of basic constraint for two reasons. In general SMT solving, the underlying DPLL(T) procedure only deals with individual constraints from pure theories, with communication between them handled by a Nelson-Oppen theory combination. So at the lowest level, the two different kinds of constraint must be separated. The second reason is that in the Marabou solver for neural networks [14], the uninterpreted function is instantiated with an actual neural network at solving time, and Marabou's input format requires references to the input and output variables of the network to be separate from the actual application of the network, which is left implicit. The final two constructors allow combination of Constraints via conjunction and disjunction.

Both the LinExp and Constraint data types enforce a normal form. LinExps push all multiplication by constants to variables, and Constraints represent constraints in negation normal form. Counterparts to the operations of multiplication and negation directly manipulate the syntax trees to maintain these normal forms:

$$\begin{aligned} \_ \circledast \_ &: \mathbb{Q} \to \text{LinExp}\,\Delta \to \text{LinExp}\,\Delta \\ \text{negate} &: \text{Constraint}\,\Delta \to \text{Constraint}\,\Delta \end{aligned}$$

We construct normalised formulas with existentials in them in two stages. First we generate "tree" formulas with constraints at the leaves and quantifiers anywhere, as described by the following data type:

```
data ExFormula : LinVarCtxt → Set where
 ‘constraint : Constraint Δ → ExFormula Δ
 ‘ex         : ExFormula (Δ, ℚ) → ExFormula Δ
 _‘and‘_     : ExFormula Δ → ExFormula Δ → ExFormula Δ
 _‘or‘_      : ExFormula Δ → ExFormula Δ → ExFormula Δ
```

We flatten the ExFormula representation to a formula in prenex form, where all existential quantifiers are at the top level. This representation can be submitted to an SMT solver.

4.3.3 Lifting Monad. With these representations of normal forms, we can nearly define an NbE procedure for our language. However, there are two features of our IR that disrupt a straightforward NbE algorithm.

1. When normalising if s then t else u conditional terms we will no longer get a Boolean result from evaluating the condition s, instead getting a Constraint. If t and u are themselves Constraints, then we can apply the meaning-preserving transformation if s then t else $u \rightsquigarrow ((s \wedge t) \vee (\neg s \wedge u))$ (using the constructors and function defined in the previous section). However, if t and u are both linear expressions, or both of function type, then it is not clear how to proceed. Intuitively, we will always be able to translate away conditionals because a complete property is of type `Bool` $p\ l$, but this relies on knowledge of the context in which linear expressions and functions are interpreted.
2. A similar problem occurs when attempting to interpret applications of uninterpreted functions into the linear expression syntax, which does not contain function applications. An uninterpreted function application needs to be interpreted in the wider context of a constraint expression in terms of existentials and constraints. For example, a use of $f\ e$ in a constraint will be translated to $\exists x.\exists y.x = e \wedge y = f\ x \wedge p$, where y is used as the replacement for $f\ e$ in the translation of the context into p. So we effectively need the ability to make new definitions for variables as we interpret.

To address these features, we enrich our interpretation to pretend that we do have them by treating them as *computational effects* that can be *handled* in a context where they can be interpreted. We define a monad Lift on the category of LinVarCtxt indexed sets that supports operations for interpreting conditionals and definitions. If we define the interpretation of `Real` as not just LinExp but Lift LinExp, then when interpreting terms of type `Real` we are able to pretend we have conditionals and let expressions. By applying Moggi's Call-by-Value [19] translation to systematically extend our interpretation to use Lift everywhere, we will be able to interpret conditionals and definitions at any type. The normalising interpretation of a closed term of existential Boolean type will live in Lift ExFormula, whereupon we will be able to translate conditions and let expressions into Boolean combinations and existentials, respectively.

The carrier of the Lift monad is defined as an indexed inductive type. Each constructor implicitly quantifies over all linear variable contexts Δ.

```
data Lift (A : Set^LinCtxt) : Set^LinCtxt where
 return    : A Δ → Lift A Δ
 if        : Constraint Δ → Lift A Δ → Lift A Δ → Lift A Δ
 letLinexp : LinExp Δ → Lift A (Δ, ℚ) → Lift A Δ
 letFunexp : Var Δ → Lift A (Δ, ℚ) → Lift A Δ
```

The `letFunexp` and `letLinexp` constructors extend the context with the result of a linear expression or function application respectively. They can be thought of as the moral equivalents of let $x = e$ in e'. The `if` constructor represents a conditional predicated on a Constraint, with true and false branches as children. The `return` constructor terminates a piece of virtual syntax with some linear variable context indexed value.

4.3.4 The Normalising Interpretation. Equipped with the normal form types LinExp and ConstraintExp and the lifting monad Lift, we can now define the normalising interpretation of the language. Here we use the linearity and constraint information of well typed specifications to guarantee that an expression of type `Bool` $L\,\exists$ always yields a formula containing only existentials over linear constraints and functional (dis)equalities.

All types are interpreted as renamable sets as defined in Section 4.3.1. The interpretation of the λ-calculus part of the system is interpreted as a standard Kripke ("possible worlds") semantics [18], albeit with the Lift monad inserted in a CBV style. The base types are interpreted like so, using the linearity and polarity information:

$$\begin{aligned} [\![\texttt{Real}\ C]\!]\ \Delta &= \mathbb{Q} \\ [\![\texttt{Real}\ L]\!]\ \Delta &= \text{LinExp}\,\Delta \\ [\![\texttt{Real}\ N]\!]\ \Delta &= \top \end{aligned}$$

$$\begin{aligned} [\![\texttt{Bool}\ U\ l]\!]\ \Delta &= \text{Constraint}\,\Delta \\ [\![\texttt{Bool}\ \exists\ l]\!]\ \Delta &= \text{ExFormula}\,\Delta \end{aligned}$$

Note that non-linear numbers are represented by the unit type $\top$. Any attempt to multiply linear expressions will result in the unique value of this type. Since, by the typing relation, it is not possible for non-linear terms to appear in constraints, this discarding of information has no consequence.

The interpretation of a whole program in the normalising semantics, after expansion of Lift ExFormula into an ExFormula and flattening to prenex form, is a function:

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

Thus, just by our intrinsic typing of source and target programs, we know that we have a total function for normalising programs that is always type correct.

4.4 Correctness

We are guaranteed well-formedness of normal forms by Agda's type checking. To prove full correctness, our desired result is that for closed programs $t : \vdash \texttt{Bool}\, L\, \exists$, the standard semantics, *for all function interpretations* f, $\mathcal{S}[\![t]\!]\, f$ (a set) and the normalising semantics $\mathcal{N}[\![t]\!]$ (a formula) agree, in the sense that the set and the semantics of the normal form are equi-inhabited. This specification only covers the `Bool` type, however, so we define a logical relation to lift this property to all the types in the language. This logical relation relates the standard semantics Section 4.2 and the normalisation semantics Section 4.3 at all types.

The structure of the logical relations argument follows the Kripke semantics of the normalising semantics closely. To show that evaluating the syntactic formula from the normalisation yields the same result as the standard semantics, we move from sets indexed with linear variable contexts to sets indexed by a pair of a linear variable context and an environment mapping variables to concrete rational values. Once this setup is established, the construction of the logical relations proof is structured as an alternative interpretation of the syntax, reusing the general notions from Kripke semantics.

Theorem 4.1. *For closed terms $t : \vdash \texttt{Bool}\, L\, \exists$, the standard semantics and the interpretation of the normalising semantics are equi-satisfiable, for all concrete interpretations of the (syntactically) uninterpreted function f:*

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

5 Conclusions

We have presented a type-system for higher-order specification languages that tracks whether a specification can be compiled to a QF_LRA solver. A key feature of this type system is that the user can program as if it doesn't exist; they are not required to think in terms of the constraints of the underlying solver when writing their specification. Our system is constructive in two senses: if it detects that the specification may not be compilable, it constructs a detailed error message to guide the user to the source of the problem; if the system can prove that a specification is compilable, the type information generated is used directly in a normalisation procedure to generate formulas in a form suitable for a QF_LRA solver. Moreover, we have proved that the normalisation procedure is total, type preserving, and semantics preserving in Agda.

Our system is necessarily incomplete in terms of which specifications can be compiled, as we discussed in Section 3.5. To remove all sources of incompleteness in general appears to require some form of dependent types in order to track the semantic meaning of terms. The alternative, discussed in Section 3.5, is to delay checking until after an error has occurred in normalisation. The result of this would be that all valid specs would compile, at the cost of the error messages sometimes being inaccurate.

Our normalisation procedure is an example of logical encoding of a high level language into SMT form. Solvers such as Z3 [11] and CVC5 [3] perform some encoding internally, e.g., to separate constraints between theories as we did for uninterpreted functions and linear constraints, but make no formal claims of correctness for these. Tools such as SMTCoq [12] embed solving into a higher order language, but assume that the problem is in the right form for a solver before it is processed. Our procedure is remarkably generic in that most of the technical development is concerned with general concepts common to all Kripke semantics, and we expect that it will be applicable to backends beyond SMT solvers. We intend to parametrise over the choices of base kinds and types to develop a general framework for normalising DSLs. An interesting DSL is the Nested Relational Calculus (NRC) which, like our language, is a higher order language that normalises to flat SQL queries. Normalisation proofs for NRC in the past have been significantly complicated by the presence of if-then-else (e.g., Cooper [8], which had a bug fixed by Ricciotti and Cheney [23]). Our Lift monad (Section 4.3.3), provides a generic method.

The system we have presented is currently in use in the Vehicle tool for neural network verification [10]. As the use of DSLs backed by automatic theorem provers is only going to increase, we hope that this work may be of use to others in improving the user experience for the next generation of solvers.

Acknowledgments

This work was funded by the AISEC grant under Engineering and Physical Sciences Research Council. Grant numbers: EP/T026952/1, EP/T026960/1, and EP/T027037/1. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

A Error Messages

A.1 Other Tools

Z3 SMTLib: The following non-linear SMTLib program:

(set-logic QF_LRA)
(declare-const a Real)
(define-fun f ((x Real) (y Real)) Real (x y))*
(assert (>= (f a a) 0)
(check-sat)

when run with Z3 version 4.8.12 outputs the error:

(error "line 5 column 0: logic does not support nonlinear arithmetic")

In this case Z3 correctly identifies that the problem is the non-linearity, but doesn't provide any information about where the non-linearity is. In big specifications with many hundreds of lines, the non-linearity can be very hard to find.

F*: The following F* program with alternating quantifiers:

```
module Test
let f (x : int) = x + 1
let _ =
  assert (forall y . exists x . f x == y)
```

when run with F* version 2021.06.06 outputs the error:

Test.fst(5,8-5,14): (Error 19) assertion failed SMT solver says:
unknown because (incomplete quantifiers) (fuel=8; ifuel=2);
unknown because (incomplete quantifiers) (fuel=4; ifuel=2);
unknown because (incomplete quantifiers) (fuel=2; ifuel=2);
unknown because (incomplete quantifiers) (fuel=2; ifuel=1);

Note: 'canceled' or 'resource limits reached' means the SMT query timed out, so you might want to increase the rlimit; 'incomplete quantifiers' means Z3 could not prove the query, so try to spell your proof out in greater detail, increase fuel or ifuel 'unknown' means Z3 provided no further reason for the proof failing (see also Test.fst(5,27-5,46))

This error message is problematic for two reasons. Firstly, the suggested fixes of increasing the resource limit or the fuel wouldn't work. Secondly, while it does mention the general problem 'incomplete quantifiers', this phrase may not be meaningful to a non-expert user who does not understand the Z3 internals. Furthermore, it doesn't identify the alternating quantifiers as the source of the incompleteness.

We note that in general F* *does* support alternating quantifiers, via SMT patterns. Nonetheless, in this case no such patterns exist and therefore we believe the error message should refer to the alternating quantifiers as the problem.

A.2 Our Error Messages

We now present example error messages generated by our analysis.

Example 1: Alternating negated quantifiers

The specification:

```
fun f : Vec Real 1 -> Vec Real 1

p : Bool
p = forall y. not not exists x. f[x] == [y]
```

gives the error message:

Cannot verify specifications with alternating quantifiers. In particular:
1. *the inner quantifier is the 'exists' located at line 4, columns 22-28*
2. *which is turned into a 'forall' by the 'not' at line 4, columns 18-21*
3. *which is turned into a 'exists' by the 'not' at line 4, columns 14-17*
4. *which alternates with the outer 'forall' quantifier at line 4, columns 5-11.*

Example 2: Non-linear quantified variables.

The specification:

```
fun f : Vec Real 1 -> Vec Real 1

square : Real -> Real
square y = y * y

p : Bool
p = forall x. f [0] ! 0 > square x
```

gives the error message:

Cannot verify specifications with non-linear constraints. In particular the multiplication at line 4, columns 14-15 involves:
1. *the quantified variable 'x' introduced at line 7, columns 5-11*
2. *which is used as an input to the function 'square' at line 7, columns 27-35*
3. *which is used on the left hand side of the multiplication*

and
1. *the quantified variable 'x' introduced at line 7, columns 5-11*
2. *which is used as an input to the function 'square' at line 7, columns 27-35*
3. *which is used on the right hand side of the multiplication*

Example 3: Higher-order function application

The specification:

```
fun f : Vec Real 1 -> Vec Real 1

g : Real -> Bool
g y = forall x. f x ! 0 >= y

notApp : (Real -> Bool) -> Real -> Bool
notApp h x = not (h x)

p : Bool
p = forall y. notApp g y
```

gives the error message:

Cannot verify specifications with alternating quantifiers. In particular:

1. *the inner quantifier is the 'forall' located at Line 4, Columns 14-20*
2. *which is returned as an output of the function 'g' at Line 10, Columns 23-31*
3. *which is used as an input to the function 'notApp' at Line 10, Columns 16-33*
4. *which is turned into an 'exists' by the 'not' at Line 7, Columns 12-15*
5. *which is returned as an output of the function 'notApp' at Line 10, Columns 16-33*
6. *which alternates with the outer 'forall' at Line 10, Columns 5-11*

B Constraint Solving

The subtyping relation (Figure 8) encodes the notion that the linearity and polarity of a type is the maximum of a set of other similar types's linearities and polarities. For example, [Real L, Real C] are subtypes of Real L as their linearities are bounded above by L. In contrast, [Real L, Real N] are not a valid set of subtypes of Real L as N is above L in the lattice defined in Figure 6. The notation Bool _ _ $\in X$ indicates that at least one of the types in X is of type Bool l p for some l and p. Rules are tried in order.

Specific subtyping rules only need to be defined for the types with polarity and linearity annotations (i.e. Real and Bool) and for the Vec which may recursively contain such types. The remaining types are related by equality in the final rule.

Why is the Subtypes relation n-ary? A binary relation would be infeasible as it would not allow for the calculation of a tight upper bound. A ternary relation is minimal, and indeed the n-ary relation generates ternary MaxLin and MaxPol constraints. The transition point from n-ary to ternary is in some sense arbitrary, but our choice to do it here greatly simplifies the type of vector literals. We have not implemented rules for function subtyping, but we hypothesise it could be done following the extensive literature in this area.

The solutions for HasExists (Figure 9) are analogous to those for HasForall presented in Figure 5 in Section 3.3.3 in the main text.

Equality (Figure 10) is overloaded to apply to Real, Index and Vec*of* the above. Note that extending equality to the Bool would necessitate splitting the type class HasEq into two separate classes, one for equality comparisons and one for inequality comparisons. This is because the latter implicitly involves a negation and therefore would affect polarity.

Ordering (Figure 11) is restricted to only apply to the Real and Index types.

Finally, the HasIndexLiteral constraint simply ensures that the literal value fits inside the required index type (Figure 12).

C Incompleteness

As discussed in Section 3.5, the type-system is incomplete and computes an under-approximation of whether a given specification lies in the QF_LRA fragment. Here we list of some of the sources of incompleteness, and a brief discussion of whether they can be mitigated.

Numeric cancellation. As no normalisation is performed, then expressions that cancel will be given a conservative classification e.g.

```
forall x. x * x − x * x >= 0
```

will be marked as non-linear. This is not easily fixable under the current scheme.

Boolean cancellation. Similar to numeric cancellation, the following:

```
False and exists x. x >= 0
```

will be marked as existentially quantified. This is not easily fixable under the current scheme.

Vector indexing. Thanks to the subtyping relation in Appendix B, a vector of rationals takes the maximum linearity of its elements. Therefore if x is a quantified variable then $[x, 0]$ will be typed as a vector of linear rationals. Consequently:

```
forall x. [x * x, 0] ! 1 >= 0
```

will be typed as a non-linear expression despite actually being constant. To solve this, the type-system could be refined to have dependently-typed indexing of heterogeneously-typed products.

If-statements. A similar problem occurs in the different branches of if-statements, e.g.

```
(if x > 0 then y else 1) * (if x < 0 then 1 else y)
```

$$(\text{subtypeBool})\;\frac{\texttt{Bool \_ \_} \in \{\tau, \tau_1, \ldots, \tau_n\} \quad \begin{array}{c} l, l_1 \;\ldots l_n \\ l'_1 \;\ldots l'_{n-1} \\ p, p_1, \ldots, p_n \\ p'_1 \ldots p'_{n-1} \end{array} = ?^{+} \quad \begin{array}{l} \tau \sim \texttt{Bool}\; l \;\; p \\ \tau_1 \sim \texttt{Bool}\; l_1\; p_1 \\ \ldots \\ \tau_n \sim \texttt{Bool}\; l_n\; p_n \end{array} \quad \begin{array}{l} \texttt{MaxLin}\; C \quad l_1\; l'_1 \\ \texttt{MaxLin}\; l'_1 \quad l_2\; l'_2 \\ \ldots \\ \texttt{MaxLin}\; l'_{n-1}\; l_n\; l \end{array} \quad \begin{array}{l} \texttt{MaxPol}\; U \quad p_1\; p'_1 \\ \texttt{MaxPol}\; p'_1 \quad p_2\; p'_2 \\ \ldots \\ \texttt{MaxPol}\; p'_{n-1}\; p_n\; p \end{array}}{\texttt{Subtypes}\; \tau\; [\tau_1 \ldots \tau_n]}$$

$$(\text{subtypeReal})\;\frac{\texttt{Real \_} \in \{\tau, \tau_1, \ldots, \tau_n\} \quad \begin{array}{c} l, l_1 \ldots l_n \\ l'_1 \ldots l'_{n-1} \end{array} = ?^{+} \quad \begin{array}{l} \tau \sim \texttt{Real}\; l \\ \tau_1 \sim \texttt{Real}\; l_1 \\ \ldots \\ \tau_n \sim \texttt{Real}\; l_n \end{array} \quad \begin{array}{l} \texttt{MaxLin}\; C \quad l_1\; l'_1 \\ \texttt{MaxLin}\; l'_1 \quad l_2\; l'_2 \\ \ldots \\ \texttt{MaxLin}\; l'_{n-1}\; l_n\; l \end{array}}{\texttt{Subtypes}\; \tau\; [\tau_1 \ldots \tau_n]}$$

$$(\text{subtypeVec})\;\frac{\texttt{Vec \_ \_} \in \{\tau, \tau_1, \ldots, \tau_n\} \quad n, \tau', \tau 1'_1 \ldots \tau'_n = ?^{+} \quad \begin{array}{l} \tau \sim \texttt{Vec}\; \tau'\; n \\ \tau_1 \sim \texttt{Vec}\; \tau'_1\; n \\ \ldots \\ \tau_n \sim \texttt{Vec}\; \tau'_n\; n \end{array} \quad \texttt{Subtypes}\; \tau'\; [\tau'_1, \ldots, \tau'_n]}{\texttt{Subtypes}\; \tau\; [\tau_1 \ldots \tau_n]}$$

$$(\text{subtypeOther})\;\frac{\tau_1 \sim \tau_2 \quad \ldots \quad \tau_{n-1} \sim \tau_n \quad \tau_n \sim \tau}{\texttt{Subtypes}\; \tau\; [\tau_1 \ldots \tau_n]}$$

Figure 8. Subtyping rules

$$(\text{hasExistsReal})\;\frac{l_1 \sim L \quad l_2, p_1, p_2 = ?^{+} \quad \tau_1 \sim \texttt{Bool}\; l_2\; p_1 \quad \tau_2 \sim \texttt{Bool}\; l_2\; p_2 \quad \texttt{ExistsPol}\; p_1\; p_2}{\texttt{HasExists}\; (\texttt{Real}\; l_1 \;\texttt{->}\; \tau_1)\; \tau_2}$$

$$(\text{hasExistsVec})\;\frac{\texttt{HasExists}\; (\tau_1 \;\texttt{->}\; \tau_2)\; \tau_3}{\texttt{HasExists}\; (\texttt{Vec}\; n\; \tau_1 \;\texttt{->}\; \tau_2)\; \tau_3} \qquad (\text{hasExistsIndex})\;\frac{l, p = ?^{+} \quad \tau_1 \sim \texttt{Bool}\; l\; p \quad \tau_2 \sim \texttt{Bool}\; l\; p}{\texttt{HasExists}\; (\texttt{Index}\; n \;\texttt{->}\; \tau_1)\; \tau_2}$$

Figure 9. `HasExists` rules

will be typed as a non-linear expression despite the fact that the two conditions are mutually exclusive.

A slightly more subtle problem is the linearity and polarity of the conditional statement. For example:

$$[0, 1] \;!\; (\texttt{if}\; x * x > 1\; \texttt{then}\; 0\; \texttt{else}\; 1)$$

is a non-linear specification but the return type of the `if` is `Index 2` and therefore cannot store this information. Out system works around this by requiring the condition to be linear and unquantified. In general, to handle this in general appears to require an effects system to track that the output of the `if` implicitly depends on a non-linear constraint.

References

[1] Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1683)*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.). Springer, 453–468. https://doi.org/10.1007/3-540-48168-0_32

[2] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–9.

$$\text{(hasEqReal)}\ \frac{\texttt{Real}\ \_ \in \{\tau_1, \tau_2\} \quad l_1, l_2, l_3 = ?^{+} \quad \tau_1 \sim \texttt{Real}\ l_1 \quad \tau_2 \sim \texttt{Real}\ l_2 \quad \tau_3 \sim \texttt{Bool}\ l_3\ U \quad \texttt{MaxLin}\ l_1\ l_2\ l_3}{\texttt{HasEq}\ \tau_1\ \tau_2\ \tau_3}$$

$$\text{(hasEqIndex)}\ \frac{\texttt{Index}\ \_ \in \{\tau_1, \tau_2\} \quad n = ?^{+} \quad \tau_1 \sim \texttt{Index}\ n \quad \tau_2 \sim \texttt{Index}\ n \quad \tau_3 \sim \texttt{Bool}\ C\ U}{\texttt{HasEq}\ \tau_1\ \tau_2\ \tau_3}$$

$$\text{(hasEqVec)}\ \frac{\texttt{Vec}\ \_\ \_ \in \{\tau_1, \tau_2\} \quad \tau_1', \tau_2', n = ?^{+} \quad \tau_1 \sim \texttt{Vec}\ \tau_1'\ n \quad \tau_2 \sim \texttt{Vec}\ \tau_2'\ n \quad \texttt{HasEq}\ \tau_1'\ \tau_2'\ \tau_3}{\texttt{HasEq}\ \tau_1\ \tau_2\ \tau_3}$$

Figure 10. `HasEq` rules

$$\text{(hasOrdReal)}\ \frac{\texttt{Real}\ \_ \in \{\tau_1, \tau_2\} \quad l_1, l_2, l_3 = ?^{+} \quad \tau_1 \sim \texttt{Real}\ l_1 \quad \tau_2 \sim \texttt{Real}\ l_2 \quad \tau_3 \sim \texttt{Bool}\ l_3\ U \quad \texttt{MaxLin}\ l_1\ l_2\ l_3}{\texttt{HasOrd}\ \tau_1\ \tau_2\ \tau_3}$$

$$\text{(hasOrdIndex)}\ \frac{\texttt{Index}\ \_ \in \{\tau_1, \tau_2\} \quad n = ?^{+} \quad \tau_1 \sim \texttt{Index}\ n \quad \tau_2 \sim \texttt{Index}\ n \quad \tau_3 \sim \texttt{Bool}\ C\ U}{\texttt{HasOrd}\ \tau_1\ \tau_2\ \tau_3}$$

Figure 11. `HasOrd` rules

$$\text{(hasIndexLiteral)}\ \frac{i < n}{\texttt{HasIndexLiteral}\ i\ (\texttt{Index}\ n)}$$

Figure 12. `HasIndexLiteral` rules

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[4] Françoise Bellegarde and James Hook. 1994. Substitution: A Formal Methods Case Study Using Monads and Transformations. *Sci. Comput. Program.* 23, 2-3 (1994), 287–311. https://doi.org/10.1016/0167-6423(94)00022-0

[5] Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991.* IEEE Computer Society, 203–211. https://doi.org/10.1109/LICS.1991.151645

[6] Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91. https://doi.org/10.1017/s0956796899003366

[7] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

[8] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *Database Programming Languages - DBPL 2009, 12th International Symposium, Lyon, France, August 24, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5708)*, Philippa Gardner and Floris Geerts (Eds.). Springer, 36–51. https://doi.org/10.1007/978-3-642-03793-1_3

[9] Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1-3 (1996), 167–177.

[10] Matthew L. Daggitt, Wen Kokke, Atkey Bob, Natalia Ślusarz, and Marco Casadio. 2022. Vehicle. https://github.com/vehicle-lang/vehicle Accessed on 22.09.2022.

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

[12] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *International Conference on Computer Aided Verification.* Springer, 126–133.

[13] Jean-Yves Girard. 1971. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In *Studies in Logic and the Foundations of Mathematics.* Vol. 63. Elsevier, 63–92.

[14] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The Marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification.* Springer, 443–452.

[15] Hyondeuk Kim and Fabio Somenzi. 2006. Finite instantiations for integer difference logic. In *2006 Formal Methods in Computer Aided Design.* IEEE, 31–38.

[16] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 389–391.

[17] Jeffrey R Lewis and Brad Martin. 2003. Cryptol: High assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, Vol. 2. IEEE, 820–825.

[18] John C. Mitchell and Eugenio Moggi. 1991. Kripke-Style Models for Typed lambda Calculus. *Ann. Pure Appl. Log.* 51, 1-2 (1991), 99–124. https://doi.org/10.1016/0168-0072(91)90067-V

[19] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[20] Tobias Nipkow. 1993. Functional unification of higher-order patterns. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 64–74.

[21] Ulf Norell. 2008. Dependently typed programming in Agda. In *International school on advanced functional programming*. Springer, 230–266.

[22] Benjamin C Pierce. 2002. *Types and programming languages.* MIT press.

[23] Wilmer Ricciotti and James Cheney. 2022. Strongly-Normalizing Higher-Order Relational Queries. *Log. Methods Comput. Sci.* 18, 3 (2022). https://doi.org/10.46298/lmcs-18(3:23)2022

[24] Sanjit A Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating modeling, verification, synthesis and learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 1–10.

[25] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

[26] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 256–270.

[27] Niki Vazou. 2016. *Liquid Haskell: Haskell as a theorem prover.* University of California, San Diego.

[28] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 60–76.

[29] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems* 31 (2018).

Received 2022-09-21; accepted 2022-11-21