Complete First-Order Reasoning for Properties of Functional Programs

ADITHYA MURALI, University of Illinois Urbana-Champaign, USA
LUCAS PEÑA, University of Illinois Urbana-Champaign, USA
RANJIT JHALA, University of California San Diego, USA
P. MADHUSUDAN, University of Illinois Urbana-Champaign, USA

Several practical tools for automatically verifying functional programs (e.g., Liquid Haskell and Leon for Scala programs) rely on a heuristic based on unrolling recursive function definitions followed by quantifier-free reasoning using SMT solvers. We uncover foundational theoretical properties of this heuristic, revealing that it can be generalized and formalized as a technique that is in fact *complete* for reasoning with combined First-Order theories of algebraic datatypes and background theories, where background theories support decidable quantifier-free reasoning. The theory developed in this paper explains the efficacy of these heuristics when they succeed, explain why they fail when they fail, and the precise role that user help plays in making proofs succeed.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Functional languages*; • **Theory of computation → Logic and verification**; **Automated reasoning**.

Additional Key Words and Phrases: First-Order Logic, Completeness, Natural Proofs, Thrifty Instantiation, Liquid Haskell, Refinement Types

ACM Reference Format:
Adithya Murali, Lucas Peña, Ranjit Jhala, and P. Madhusudan. 2023. Complete First-Order Reasoning for Properties of Functional Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 259 (October 2023), 37 pages. https://doi.org/10.1145/3622835

1 INTRODUCTION

The automation of program verification has been revolutionized with the advent of efficient *logic engines* that check validity of logical formulas over various theories that capture domains that programs work with (arithmetic, strings, arrays, algebraic datatypes, pointer-based heaps, etc.). In particular, *quantifier-free logics* over various theories admit decidable validity checking, and further, permit decision procedures for the combination of theories (Nelson-Oppen style combinations) that have been realized by efficient DPLL(T)-based SMT solvers [Bradley and Manna 2007; de Moura and Bjørner 2008; Nelson 1980; Nelson and Oppen 1979].

However, automation's grip becomes tenuous when it comes to the verification of first-order properties of *functional programs* over *algebraic data types* (ADTs) such as lists or trees over basic types like integers. Functional programs over ADTs can be expressed mathematically using a set of *recursively defined functions* over types. Programs hence can be expressed as a set of first-order

Authors' addresses: Adithya Murali, adithya5@illinois.edu, University of Illinois Urbana-Champaign, Urbana, USA; Lucas Peña, lucaspena13@gmail.com, University of Illinois Urbana-Champaign, Urbana, USA; Ranjit Jhala, rjhala@eng.ucsd.edu, University of California San Diego, San Diego, USA; P. Madhusudan, madhu@illinois.edu, University of Illinois Urbana-Champaign, Urbana, USA.

2475-1421/2023/10-ART259
https://doi.org/10.1145/3622835

definitions of functions *Defs* that are *universally quantified* over their inputs. The goal of verification, then, is to determine whether a particular FO (First-Order) theorem T involving these defined (or interpreted) functions is mathematically valid under a set of definitions *Defs*.

Automation is Impossible in Theory. Even though the theorem T that needs to be validated is universally quantified (and hence can be seen as a quantifier-free formula), reasoning about the validity of T under *interpreted definitions Defs* is extremely hard. The validity problem is not decidable (while validity of T under *uninterpreted functions is* typically decidable). Worse, the problem is not even recursively enumerable (there is no complete proof system nor a semi-decision procedure that is guaranteed to terminate on at least all valid theorems). A simple proof of this fact is that we can define addition and multiplication as defined (interpreted) functions using recursion, and use universal quantification to specify the neither decidable nor recursively enumerable problem of determining the *non-existence* of solutions to Diophantine equations [Matiyasevich 1993].

Automation is Effective in Practice. Despite the above hardness, there has been significant progress in systems that provide varying degrees of automation to the process of verifying such theorems. Liquid Haskell (LH) [Vazou et al. 2018] and Leon/Stainless [Blanc et al. 2013; Hamza et al. 2019] both exploit the automation provided by logic engines via decidable *quantifier-free reasoning* to prove FO theorems. Extrinsic-style verification in LH reduces checking quantifier-free (implicitly universal) properties of functions over ADTs to proving pre- and post-condition contracts that assert those properties in the code ("proofs") written by the verification engineer. [1] The Leon verifier [Blanc et al. 2013] (as well as its successor Stainless [Hamza et al. 2019]) uses a similar style of reasoning for Scala programs with quantifier-free contracts, where the contracts themselves are written using recursively defined pure Scala functions. Leon verifies each function's contract by compiling the body of the function to a *verification condition* (VC), modeling functions called in the body using defined functions and assuming they satisfy their contracts [2]. While LH and Leon provide different mechanisms for users to prove properties via induction and auxiliary lemmas, we observe that they share a common fundamental interface to logic engines: verification is reduced to proving VCs of the form $Defs \rightarrow \varphi$ where φ is universally quantified. LH treats functions defined in *Defs* as *uninterpreted* using a heuristic called *logical evaluation* that finitely unfolds the definitions for terms that appear in φ. Leon's strategy is also to unfold the recursive definitions based on function applications that occur in φ. However, it differs from LH in that it does this *recursively*, unfolding definitions iteratively for larger and larger depths and assuming that such unfolded calls to functions satisfy their contract.

To summarize, both tools automate verification via logical engines by (1) generating VCs of the form $Defs \rightarrow \varphi$ (where φ is a universally quantified FO formula), (2) treating all defined functions as largely uninterpreted, (3) instantiating definitions repeatedly only on certain terms, and (4) dispatching them to an SMT solver that does quantifier-free decidable reasoning. This technique is certainly sound but clearly not a decision procedure: LH just makes a fixed set of instantiations, which may be insufficient; Leon can continuously unfold definitions and may proceed forever (timeout). Yet, despite the hardness results, this heuristic works well in practice, giving predictable results though they may require users to find new inductive lemmas and guidance in proofs!

Why does the heuristic of unfolding recursive definitions followed by quantifier-free reasoning work so well in practice? In this paper, we establish *foundational results* that this procedure is in fact a *complete* procedure for the underlying combination of first-order theories. Our results not only

[1] LH implements various algorithms including refinement inference. In this work, when we say LH, we refer specifically to extrinsic-style full functional correctness proofs over user-defined functions using methods proposed in [Vazou et al. 2018].
[2] This is an inductive proof (induction on the size of the implicit call stack) that all functions satisfy their contracts.

explain *when* this heuristic method works well, but also explains when and why they *fail*, and the role of further help asked of the user.

The Standard Model vs Combined Theories

The answer to this question lies in the tension between the theory of the *standard model* and the *combined FO theory* of the various sorts. First-order theorems that express properties of functional programs can be seen as formulas over a *combination of sorts*, in particular sorts that refer to ADTs (e.g. trees) and the base sorts (e.g. integers) that the datatypes are built upon. When a verification engineer wishes to prove a theorem, they want it to be proven for the *fixed* universe (the standard model) consisting of the various sorts. In this universe, the ADT sort is the natural universe of *algebraic terms* of the appropriate type, with constructors and destructors interpreted in the standard manner, and the integer sort and functions over them (e.g. +) are interpreted in the standard manner.

Axiomatized Models. In first-order logic, however, we often reason with models that are *axiomatized*: we capture various properties of models using a finite or recursive set of axioms, and reason over *any* model that satisfies the axioms. In particular, ADTs can be axiomatized and the universe of integers with addition can be axiomatized. In fact, they can be individually given *complete axiomatizations*— i.e., all models satisfying the axioms satisfy the same first-order theorems as the standard model [Barwise 1977; Bjorner 1999; Hodges 1997; Kovács et al. 2017; Mal'tsev 1962; Presburger and Jabcquette 1991]. There may be other models, called *nonstandard models*, that are not isomorphic to the standard model (in fact they always exist, say, by the Löwenheim-Skolem Theorem) but one cannot distinguish them using a first-order formula. Nonstandard models are well-known in the literature [Hodges 1997; Skolem 1934].

Rogue Nonstandard Models that Disagree with the Standard Model. However, when we combine universes and their theories, interfacing them with uninterpreted functions, the combined axioms are no longer powerful enough. More precisely, it is well known that the combined axioms can admit *rogue* nonstandard models that disagree with standard models on first order expressible theorems, and hence the theory entailed by the combined axioms becomes weaker. Rogue nonstandard models are a special case of nonstandard models of the combined theory that disagree with the standard model on some first-order formulas. For example, if we take the complete axiomatization of ADTs and the complete axiomatization of uninterpreted functions (congruence axioms) and combine them, the union of the axioms admits rogue nonstandard models of ADTs that contradict theorems true in the standard model of ADTs with uninterpreted functions.

Nonstandard models exist even for complete theories, but there are no rogue models in such theories since, by the definition of completeness, all nonstandard models agree with the standard model on all FO expressible theorems. Combined theories are incomplete as there are rogue nonstandard models. However, *quantifier-free* formulas over combinations of theories (using the Nelson-Oppen method) do not suffer from such issues, which is why we can think of validity procedures for them as provers for the standard model.

Main Contributions

Our central insight in this paper is that the method of unfolding recursive definitions and performing quantifier-free reasoning can *always* prove and *only* prove the subset of theorems that are valid over the *combined theory of ADTs and the background sorts*. Consequently, it *cannot* prove theorems that are valid in the standard model but invalid in a rogue nonstandard model. We develop this insight to explain the unusual effectiveness of unfolding recursive definitions into uninterpreted function applications, via four concrete contributions.

1. A FLUID Logic (§ 4). Our first contribution is the definition of a logic called *FLUID* (First-order Logic for Universal properties under Inductive Definitions) that captures the essence of definitions[3] and VCs generated by LH[4] and Leon. *FLUID* formulas are of the form $Defs \rightarrow \varphi$ where *Defs* are *provably terminating* recursive definitions, and φ is a universally quantified formula. Verification conditions for correctness of many functional programs can be formulated using *FLUID* formulas; in fact, systems like LH and Leon generate VCs that are in this logical fragment.

2. Completeness of Unfolding followed by Quantifier-Free Reasoning (UQFR) (§ 5). UQFR is a technique for proving validity by modeling recursive functions as uninterpreted functions, unfolding recursive definitions *Defs* systematically on a class of ground terms, and reasoning with the resulting quantifier-free formulae using decision procedures. Our second contribution is a foundational result that shows that UQFR is a *complete* semi-decision procedure for the validity of *FLUID* formulas over the combined first-order theory of ADTs and background sorts. Namely, UQFR guarantees to prove all theorems that are valid in the combined FO theory. Consequently, when a theorem that is valid over the standard model is *not* proven using this technique, we are guaranteed that there is a rogue nonstandard model (satisfying the ADT and background theories) where the theorem does not hold. The proof of completeness is nontrivial for two reasons. First, the unfoldings of recursive definitions that UQFR uses (and tools such as LH and Leon use) are *thrifty*; they instantiate definitions of functions only on terms on which they are called, and do not expand instantiations to terms that arise from the underlying axiomatizations of theories. Second, every time a theorem that is valid on the standard model is *not* proven, it is nontrivial to construct a rogue nonstandard model falsifying the theorem. The model construction in the proof of this theorem crucially exploits the fact that *FLUID* definitions are provably terminating.

3. Completeness in Practice (§ 6). Thus, far from being a whimsical heuristic that happens to work in practice, UQFR is rather a robust procedure whose completeness may explain why this heuristic performs so predictably well. In particular, it does not miss proving theorems that can be proved using pure FO reasoning of the underlying axioms of the theories. Our third contribution shows how this bears out in practice. We explain how LH performs *FLUID* verification using UQFR. Crucially, when theorems are not proved valid, we show it is because rogue nonstandard models exist, and that the lemmas and induction hints provided by the user then serve to eliminate such models, all while reasoning within the *FLUID* fragment. Next, we show how we can use a slightly different *FLUID* formula to mimic Leon's more sophisticated reasoning which additionally assumes pre/post contracts for functions at each unfolding. Hence, our completeness result also applies to explain the effectiveness of Leon (§ 7).

4. Limits of FLUID (§ 8). Our final contribution is a set of results that show why our results on *FLUID* are unlikely to extend to more expressive logics. We show though the validity problem for *FLUID* admits complete procedures, it is *undecidable*, hence distinguishing it from several decidable fragments identified in the literature (e.g., [Suter et al. 2010]). We also show that attempts to generalize *FLUID*, e.g. by allowing functions whose definitions are required to be terminating (but not *provably* terminating using FO proofs) makes UQFR *not* complete. This result also implies that replacing definitions with arbitrary universally quantified formulas makes UQFR an incomplete procedure.

[3]LH and Leon also support higher-order functions, but such definitions are beyond the scope of this paper. We provide further discussion on higher-order functions in Section 10.

[4]In consultation with the developers of LH, we believe that *FLUID* captures all VCs generated by LH for extrinsic style proofs using refinement reflection!

2 OVERVIEW

In this section we provide an overview of our work, which defines *FLUID*, a fragment of First-Order Logic (FOL) that expresses the quantified verification conditions that arise when verifying correctness properties of functional programs (see Section 4), and show how these VCs can be proved valid by a *thrifty* unfolding of definitions followed by quantifier-free reasoning. Additionally, we show in § 6 and § 7 that verification in systems like LH and Leon reduces to checking validity of *FLUID* fragment formulas. We illustrate these ideas via example programs over the datatype of lists over integers:

```
data List = Nil | Cons Int List
```

2.1 Insertion and Sortedness

Consider the following program that inserts an element into a (sorted) list. We define both `insert`-ion and `sorted`-ness via recursive functions

```
sorted :: List → Bool
sorted Nil                    = True
sorted (Cons h Nil)           = True
sorted (Cons h1 (Cons h2 tl)) = h1 ≤ h2 && sorted (Cons h2 tl)

insert :: List → Int → List
insert Nil k                      = Cons k Nil
insert (Cons x xs) k | x >= k     = Cons k (Cons x xs)
                     | otherwise  = Cons x (insert xs k)
```

Definitions. We can encode the above Haskell programs in FOL where each function's definition introduces no new variables, instead using destructors (*head*, *tail*) and recognizers (*isNil*, *isCons*) to simulate pattern matching. To ensure that destructors are applied sensibly, we *guard* the use of terms of the form $head(t)$ and $tail(t)$ with the recognizer $isCons(t)$.

$$\forall x.\mathsf{List}.\ sorted(x) = \mathsf{ite}(isNil(x), True, \mathsf{ite}(isNil(tail(x)), True, head(x) \leq head(tail(x)) \wedge sorted(tail(x))))$$

$$\forall x : \mathsf{List},\ k : \mathsf{Int}.\ insert(x,k) = \mathsf{ite}(isNil(x), Cons(k, Nil), \mathsf{ite}(head(x) \geq k, Cons(k,x), Cons(head(x), insert(tail(x),k))))$$

where we treat *sorted* and *insert* as uninterpreted functions in the signature. We refer to these formulae as the *definitions* of *sorted* and *insert* and denote them by def_{sorted} and def_{insert} respectively.

Verification Conditions. Let us consider the example of verifying that inserting an element k into the empty list yields a sorted list. We state this formally as the following *verification condition* (VC):

$$(def_{sorted} \wedge def_{insert}) \rightarrow sorted(insert(Nil, k))$$

Note that this VC is of the form $DEF \rightarrow \varphi$, where DEF is a set of definitions (when it appears in a formula we are referring to the conjunction of the formulas in the set) and φ is quantifier-free, i.e., all variables are implicitly universally quantified. Informally, the VC says that the property φ should hold *assuming* the set of definitions DEF. The *FLUID* fragment we define (see Section 4) consists of such formulas.

Unfolding. We prove the above VC valid by *unfolding* the definitions. For a term t, let $def_{sorted}[t]$ denote the quantifier-free formula obtained by instantiating the quantified variable x in def_{sorted} with t. We refer to this as unfolding the definition of *sorted* on t. Similarly we can define the unfolding $insert[\bar{t}]$. To prove the VC valid we simply unfold definitions on arguments that occur in φ, i.e., we attempt to prove

$$(def_{insert}[(Nil, k)] \wedge def_{sorted}[insert(Nil, k)]) \rightarrow sorted(insert(Nil, k))$$

This formula can be dispatched using SMT solvers [Barrett et al. 2011a; de Moura and Bjørner 2008] that use a combination of decision procedures for ADTs and Integers. It is in fact valid because unfolding def_{insert} on (Nil, k) shows that $insert(Nil, k)$ evaluates to $Cons(k, Nil)$, and unfolding def_{sorted} on $Cons(k, Nil)$ shows that $sorted(insert(Nil, k))$ evaluates to $True$.

We generalize this technique of Unfolding definitions followed by Quantifier-Free Reasoning into an algorithm UQFR (Section 5), and argue that tools like Liquid Haskell (Section 6) and Leon (Section 7) perform similar reasoning on such formulas.

2.2 Insertion Preserves Sortedness

Next, let us turn to a more interesting theorem, namely that insertion preserves sortedness. Formally, we wish to prove the following *contract* for insertion: $\forall x, k.sorted(x) \rightarrow sorted(insert(x, k))$. Here the VC is $VC_{simple} \equiv (def_{sorted} \wedge def_{insert}) \rightarrow (sorted(x) \rightarrow sorted(insert(x, k)))$

Unlike the example in Section 2.1, it turns out that there is no set of terms such that unfolding the definitions on these terms can prove the VC valid. Consequently, LH fails to prove the theorem.

Using Contracts. Tools like Leon not only unfold definitions but also use contracts for terms generated during unfolding. For example, note that unfolding def_{insert} on (x, k) yields the term $insert(tail(x), k)$. Then, the VC that Leon attempts to prove is not VC_{simple} but rather

$$VC_{\text{Leon}} \equiv (def_{sorted} \wedge def_{insert}) \rightarrow \big((x \neq Nil \rightarrow (sorted(tail(x)) \rightarrow sorted(insert(tail(x), k)))) \\ \rightarrow (sorted(x) \rightarrow sorted(insert(x, k))) \big)$$

which additionally assumes the contract for $insert(tail(x), k)$ (when $x \neq Nil$). Observe that this VC is also of the form $DEF \rightarrow \varphi$ and is therefore in the *FLUID* fragment. We show in Section 7 that VC_{Leon} can be obtained automatically from the original VC, i.e., VC_{simple}.

We attempt to prove VC_{Leon} using the same technique of unfolding recursive definitions on arguments appearing in the formula (UQFR). This succeeds, and one can verify that unfolding *insert* on $\{(x, k), (tail(x), k)\}$ and unfolding *sorted* on $\{x, tail(x), insert(x, k), insert(tail(x), k)\}$ proves VC_{Leon} valid[5]. Observe that the unfolding strategy used in the examples we have seen is *thrifty* in the sense that definitions are unfolded *exactly* on terms that occur as arguments to the corresponding functions. We discuss the utility of this strategy in Section 10.

It is clear that using contracts is a more powerful approach. In general, there are theorems whose proofs require even more instantiations of contracts on terms obtained during further unfoldings. We show in Section 7 using a reduction that the use of multiple repeated instantiations of definitions as well as contracts can also be viewed as proving *FLUID* fragment formulas using UQFR. Consequently, our results apply not only to LH but also to tools like Leon.

2.3 Membership in a Sorted List

One prominent aspect of program verification in LH or Leon is proof by induction. However, induction is *not* part of UQFR. In this section we discuss the example of checking membership in a sorted list where all the above approaches fail, and explain the role of induction (in the form of explicit user help) in these tools. We first define a function *elems* to capture the set of elements stored in a list and a function *mem* that checks the membership of an element in a sorted list.

$\forall x : \mathsf{List}.\ elems(x) = \mathsf{ite}(isNil(x), \emptyset, \{head(x)\} \cup elems(tail(x)))$

$\forall x : \mathsf{List},\ k : \mathsf{Int}.\ mem(x, k) = \mathsf{ite}(isNil(x), False,\ \mathsf{ite}(k = head(x), True,$
$\mathsf{ite}(k < head(x), False,\ mem(tail(x), k))))$

[5]The reader may note here that we only argue the validity of VC_{Leon} and not the original goal VC_{simple}. We discuss why validity of the former implies validity of the latter in Section 7.

We want to verify that *mem* precisely captures membership for sorted lists. Formally, the contract is: $sorted(x) \rightarrow (mem(x, k) \leftrightarrow k \in elems(x))$

However, the approaches discussed above do not work for this example. They do not succeed even if the definitions are unfolded infinitely and contracts is assumed for all of the infinitely many terms/tuples that occur in the unfoldings.

To see why this is the case, consider what happens when we replace the usual *standard* model of ADTs and Integers we have in our minds with complete axiomatizations for each of the sorts, along with congruence axioms for the function symbols *elems* and *mem*. In this setting, the standard model is only one of the possible models and in general a model of the axiomatized universes may not be identical to the standard model. Validity in the axiomatized setting is an under-approximation to validity in the standard model, and as we show in Section 3.3 it is in fact a strict under-approximation. There are theorems that are true on the standard model that do not hold under axiomatization. This is because of the presence of *rogue nonstandard models* where the property we want to prove is not true. A rogue nonstandard model is a model that obeys the axioms but is not identical to the standard model, and further, falsifies the desired theorem. Nonstandard models always exist in the axiomatized setting, but they may satisfy all the same first-order properties as the standard model using a first-order formula. However, *rogue* nonstandard models, when they exist, can disagree with the standard model on a desired first-order theorem.

Our soundness and completeness results in Section 5 show that the proving power of unfolding definitions and using contracts is *precisely* that of validity over the axiomatized universe. Therefore, if there is a rogue nonstandard model that falsifies a property, then unfolding based reasoning *cannot* prove it. Indeed, both LH and Leon fail on the above example without extra help.

Rogue Nonstandard Model. Let us look at the rogue nonstandard model where our theorem does not hold. The universe U is:

$$\{s \mid s \text{ is a finite sequence of integers}\}$$
$$\cup \{(s, i) \mid s \text{ is an infinite sequence of integers}, i \text{ is an integer}\}$$

The finite sequences correspond to ADT lists of integers as we think of them but the infinite sequences are *nonstandard elements*[6]. *Nil* is interpreted to be the empty sequence and *Cons* behaves as expected on standard elements (prepending an element to a finite sequence). On the nonstandard elements *Cons* is defined by $Cons(j, (s, i)) = (j :: s, i + 1)$ where $j :: s$ denotes prepending j to the sequence s. *head* and *tail* behave as inverses to *Cons* in the usual sense. One can check easily that this model satisfies the usual axioms of ADTs [Bjorner 1999; Hodges 1997].

The meaning of *sorted* on this model is as expected: we define only elements with non-decreasing sequences to be sorted. The definition of $elems(x)$ is as follows

$$elems(x) = \begin{cases} \{v \mid v \text{ is an element of } x\} & \text{for a standard element } x \\ \{v \mid v \text{ is an element of } x\} \cup \{-1\} & \text{for a nonstandard element } (x, i) \end{cases}$$

Lastly $mem(x, k)$ holds if and only if k occurs in the longest non-decreasing prefix of the sequence corresponding to x. Note that if x is sorted, the longest non-decreasing prefix of x is x itself.

The above interpretations are consistent with the definitions. Consider the function for *elems*, for example. On standard elements it is consistent with the definition because it has the expected value. It is also consistent on nonstandard elements. Observe that for a nonstandard element x,

[6]The standard model of ADTs consists exactly of all terms. *Nonstandard elements* are elements in a nonstandard model that do not correspond to any term. In particular, one cannot destruct them a finite number of times to reach *Nil*. Nonstandard models always have such elements with "infinite tails".

$tail(x)$ is also a nonstandard element. Therefore, the inclusion of an extraneous element -1 in the *elems* of both x and $tail(x)$ is consistent with the recursion $elems(x) = \{head(x)\} \cup elems(tail(x))$.

Finally, we see this is a rogue nonstandard model as it does not satisfy the property $sorted(x) \rightarrow (mem(x, k) \leftrightarrow k \in elems(x))$. Consider the nonstandard element $x = ([0, 0, 0 \ldots], 0)$. Note that x is sorted since it is a non-decreasing sequence, and $elems(x) = \{0, -1\}$ by the above construction. Hence $-1 \in elems(x)$. However $mem(x, -1) = False$ since -1 does not occur in x.

Role of User Help. To prove the above example in LH, one must provide additional hints or *inductive lemmas* (whose proof of the induction step is itself performed using unfolding/UQFR)[7]. We show in Section 6 that these lemmas eliminate rogue nonstandard models like the one shown above, and therefore enable the VC to be proven using unfolding techniques. There is also work in recent literature [Murali et al. 2022; Reynolds and Kuncak 2015; Sivaraman et al. 2022; Yang et al. 2019] on automatically synthesizing lemmas.

Rogue Nonstandard Models of Integers. It is tempting to think that the above difficulties can be avoided by stating a constraint that lists are finite, i.e., there must exist a non-negative integer corresponding to the length. However, this does not work. This is because there exist *rogue nonstandard models of the integers* containing elements considered 'non-negative' by the model's interpretation but do not correspond to an integer (i.e., decrements do not reach 0). The lengths of infinite lists would be interpreted to such nonstandard numbers, and we would still need user help.

3 PRELIMINARIES

In this section we define the general setting of multi-sorted first-order logic over algebraic datatypes (ADTs) and other base types with recursively defined functions. We define *FLUID*, our logic of study, as a fragment of this logic in Section 4.

3.1 Syntax and Semantics

The logic we work with is defined over a finite set of disjoint nonempty sorts $\mathcal{S}$. We distinguish certain sorts among these as *foreground* sorts. The foreground sorts support a signature of Algebraic Datatypes (ADTs) which we describe below. The other sorts are referred to as *background* sorts (background sorts could also consist of ADTs).

An ADT signature for a sort σ consists of a finite set of function symbols $ctor_i, 1 \leq i \leq m$ called *constructors*. Each constructor has an arity $r_i \geq 0$ and a signature $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_{r_i} \rightarrow \sigma$, where $\sigma_j, \sigma \in \mathcal{S}$. Corresponding to each constructor with the above signature, we also have r_i many *destructors* $dtor_{ij}$ with signature $\sigma \rightarrow \sigma_j$ for $1 \leq j \leq r_i$, and *recognizers* is_ctor_i with signature $\sigma \rightarrow Bool$.

For example, the algebraic datatype of lists over natural numbers *ListNat* is defined by the nullary constructor $nil : ListNat$ and the binary constructor $Cons : Nat \times ListNat \rightarrow ListNat$ whose corresponding destructors are $head : ListNat \rightarrow Nat$ and $tail : ListNat \rightarrow ListNat$. The recognizer *is_cons* identifies elements that correspond to non-nil lists. Note that standard pattern matching idioms for ADTs used in functional programs can be expressed using this vocabulary.

We can also define hierarchical datatypes (e.g., lists of lists of integers), mutually recursive datatypes (e.g., terms corresponding to a context-free grammar), as well as sum (unions) and product types (tuples). We cannot define co-inductive datatypes such as infinite lists in our logic. However, we do not lose generality with respect to the various tools studied in this paper; in fact, LH's termination checker precludes the creation of values like infinite lists.

[7] Leon is able to verify *mem*, but it does so using a heuristic for *structural induction* rather than its primary algorithm of instantiating definitions and contracts. There are other examples involving list reversal where Leon also requires lemmas to deal with rogue nonstandard models that fail the theorem (see Section 7).

Our logics have signatures of the form $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{D})$, where:

- $\mathcal{S}$ is a finite non-empty set of sorts as defined above with a partitioning of sorts into a set of foreground ADT sorts and a set of background sorts. We require that there is at least one foreground sort.
- $\mathcal{F}$ is a set of constant, function, and relation symbols over the sorts $\mathcal{S}$. These will be used to model symbols over the sorts that models give interpretations to. These include functions like integer addition or set union, and constructors, destructors, and recognizers over ADT sorts.
- $\mathcal{D}$ is a set of function symbols distinct from $\mathcal{F}$ that will be used to model functions that have *definitions*.

The syntax is standard multi-sorted first-order logic over sorts $\mathcal{S}$ and over symbols $\mathcal{F} \cup \mathcal{D}$. We make two modifications. First, we require that every occurrence of a destructor term $dtor_{ij}(t)$ is *guarded* by the corresponding recognizer $is\_ctor_i(t)$ to ensure that destructor terms are well-defined. We do not lose generality as any formula with well-defined destructor terms can be rewritten to an equivalent one with the appropriate guards. In practice, tools check that $is\_ctor_i(t)$ holds by generating a separate Verification Condition. Second, we allow ite ($if - then - else$) expressions over terms and formulas. The semantics of our formulas is the standard one for FOL. We refer the reader to a standard reference text [Enderton 1972] for the notion of first-order logic, first-order models, syntax, and semantics. Semantics is defined in terms of models (aka structures) that give interpretation to all symbols, including those in $\mathcal{D}$. We use the notation $M \models \varphi$ to denote that a sentence φ evaluates to *true* in a model M, and $\varphi \models \psi$ to denote semantic entailment (all models satisfying φ also satisfy ψ).

Inductive Definitions. Intuitively, a *definition* of D (for $D \in \mathcal{D}$) gives a particular interpretation for D. The definition of a function $D \in \mathcal{D}$ of arity r is a quantified formula def_D of the form

$$\forall x_1, x_2, \ldots, x_r.\, D(x_1, x_2, \ldots, x_r) = \rho(x_1, x_2, \ldots, x_r)$$

where ρ is a quantifier-free formula over x_1 through x_r called the *body* of the definition. Of course, the body may use other inductively defined symbols $G \in \mathcal{D}$. We require that every function in $\mathcal{D}$ has exactly one definition.

In order to obtain well-defined definitions, we demand a notion of termination. We define this notion using the *standard model* of our logic, which we introduce in the next section.

3.2 The Standard Model

The intended standard interpretation of an ADT signature is the initial term algebra where the universe consists of terms that respect the sorts and the interpretation of constructors is that of term application, i.e., $[\![ctor_i]\!](e_1, \ldots, e_{r_i}) = ctor_i(e_1, \ldots, e_{r_i})$.

The destructors are interpreted as $[\![dtor_{ij}]\!](ctor_i(e_1, \ldots, e_{r_i})) = e_j$ and is otherwise interpreted to be identity on other elements [8]. Finally, recognizers are only true on terms constructed with the corresponding constructor: $[\![is\_ctor_i]\!](ctor_i(e_1, e_2, \ldots e_{r_i})) = True$, and is *False* for other elements.

More generally, our logic is parameterized by a *standard model* $\mathcal{M}_{\mathcal{S},\mathcal{F}}$ of the foreground and background sorts. This is typically true of sorts employed in program verification: ADTs, integers, sets, etc. Note that this model does not give interpretations to functions in $\mathcal{D}$.

We require that inductive definitions are terminating on the standard model using a standard *eager* semantics [Winskel 1993]. Informally, we evaluate a definition on concrete elements over the standard model as follows: (i) we evaluate recursively defined function terms by evaluating the definition on the arguments; (ii) for ite expressions, we evaluate the conditions first and then

[8] Since we consider only formulas that are guarded to check elements to be of the right sort before applying destructors, the semantics of the formula on other elements is irrelevant.

only evaluate the appropriate branch; (iii) for all other expressions, we first evaluate all recursively defined function terms (with subterms evaluated before their superterms) and then evaluate the expression. A terminating definition is one for which this procedure terminates on all inputs.

The following proposition states that over $\mathcal{M}_{\mathcal{S},\mathcal{F}}$, there exists a unique valuation for the defined functions $\mathcal{D}$ that is consistent with their definition.

Proposition 3.1. *Given $(\mathcal{S},\mathcal{F},\mathcal{D})$ let $DEF = \{def_D \mid D \in \mathcal{D}\}$ be a set of definitions. There exists a unique model $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}}$ such that the interpretation of symbols in $\mathcal{F}$ coincides with $\mathcal{M}_{\mathcal{S},\mathcal{F}}$ and interpretations of symbols in $\mathcal{D}$ satisfy their definitions, i.e., $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}} \models def_D$ for every $D \in \mathcal{D}$.*

Functional programs can be modeled using the definitions (we only consider terminating programs, of course). Universal FO properties φ of functional programs can be modeled as validity of formulas of the form $DEF \rightarrow \varphi$, where we use DEF to mean the conjunction of formulas in the set of definitions $DEF = \{def_D \mid d \in \mathcal{D}\}$.

However, as discussed in Section 1 it is easy to show that the problem of validity of even quantifier-free formulas on $M_{\mathcal{S},\mathcal{F},\mathcal{D}}$ is *not recursively enumerable*.

Proposition 3.2 (Incompleteness Theorem for the Standard Model). *There exists a sort σ with an ADT signature $\mathcal{F}$ and defined functions $\mathcal{D}$ such that checking $\mathcal{M}_{\{\sigma\},\mathcal{F},\mathcal{D}} \models \varphi$ is not recursively enumerable for quantifier-free φ.*

Note that validity over ADTs without background sorts and definitions is decidable [Mal'tsev 1962] since it has a complete axiomatization [Hodges 1997]. The introduction of definitions (programs) is what leads to incompleteness.

3.3 Combinations of Theories, Nonstandard models, and Rogue Nonstandard Models

A primary observation we make in this paper is that techniques for reasoning based on function unfolding and quantifier-free reasoning (as in Liquid Haskell and Leon) do not reason with the standard model but rather with a certain *combination of first-order theories*. We will show this in Section 5.2. In this section we formalize the notation for combined theories.

A *theory* $\mathcal{T}$ for a signature is an entailment closed set of first-order sentences. A model $\mathcal{M}$ satisfies a theory $\mathcal{T}$, denoted $\mathcal{M} \models \mathcal{T}$, if every sentence in the theory holds in the model. A sentence ψ is valid in $\mathcal{T}$, denoted $\mathcal{T} \models \psi$ if ψ belongs to $\mathcal{T}$.

A *theory tuple* for $(\mathcal{S},\mathcal{F},\mathcal{D})$ is:

- The first-order theory of ADTs $\mathcal{T}_\sigma$ for each foreground sort σ. This is the precisely the theory of the standard ADT model for σ, which may involve functions over other sorts using which the elements of σ are to be constructed. These other sorts are themselves constrained by theories like Presburger Arithmetic, or an ADT theory.
- A theory $\mathcal{T}_{bg}$ for the combined signature of the background sorts that is recursively enumerable. We require the background sorts in the standard model to satisfy this theory. In practice, this theory is the union of several axiomatized theories, say for arrays, integers, bitvectors, etc.
- Theory of uninterpreted functions with equality for symbols in $\mathcal{D}$ (i.e., the empty theory).

The combined theory $\mathcal{T}_{comb}$ of a theory tuple is the entailment closure of the union of the theories in the tuple. A model satisfies a theory tuple (and consequently the combined theory) if the projection of the model to each subset of sorts satisfies the theories constraining those sorts. The combined theory $\mathcal{T}_{comb}$ is the set of all FO sentences that hold in all these models. For example, consider the ADT *ListNat* of lists over natural numbers introduced earlier. A theory tuple for this signature could be one that has (a) the theory of ADT lists for the foreground sort, and (b) the theory of Presburger Arithmetic (natural numbers with addition) for the background sort. The combined theory is the entailment closure of the union of the two theories.

Note that the first order theory of ADTs is *complete*. Therefore, the above setting is agnostic to the choice of any complete axiomatization for the ADT sorts! [Hodges 1997; Kovács et al. 2017]. Consequently, our results are also quite general and agnostic to the choice of axiomatization.

The standard models for each sort satisfy their respective theories. The other models of the individual theories are called *nonstandard models*. The combined standard $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}}$ is a model of $\mathcal{T}_{comb}$, and similarly the other models of $\mathcal{T}_{comb}$ are nonstandard models for the combined theory.

Since the standard model is a model of $\mathcal{T}_{comb}$, it is clearly the case that $\mathcal{T}_{comb}$ is a *subset* of the theory of the standard model $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}}$, which we denote by $\mathcal{T}_{std}$. However, the reverse is not true in general, and in fact the combined theory can be *strictly smaller* than the theory of the standard model. For example, consider the above example of *ListNat* where we extend the logic with the predicate symbol R with the following recursive definition:

$$\begin{aligned} R(x) = \mathit{ite}(\mathit{is\_nil}(x), \mathit{False},\\ \mathit{ite}(\mathit{is\_nil}(\mathit{tail}(x)), \mathit{head}(x) = 1,\\ \mathit{head}(x) = \mathit{head}(\mathit{tail}(x)) \wedge R(\mathit{tail}(x)))) \end{aligned}$$

One would expect that $R(x)$ holds only for nonempty lists x whose elements are all 1. Indeed, the statement $R(x) \rightarrow \mathit{head}(x) = 1$ is valid on the standard model. However, this sentence is *not valid* in the combined theory as there is a *rogue nonstandard model* that falsifies it. In this work, we define a rogue nonstandard model as a nonstandard model that falsifies a theorem of interest which is valid on the standard model.

An example of a rogue nonstandard model falsifying $R(x) \rightarrow \mathit{head}(x) = 1$ is as follows. It has an element u in the ADT universe that does not correspond to any standard (i.e., finite) ADT term such that $R(u)$ is true and $\mathit{head}(u) = 2$. Destructing u consecutively would proceed forever without reaching *nil* and and all these elements will satisfy R and have their head element to be 2, hence satisfying the recursive equation for R. We had discussed other such examples of rogue nonstandard models in Section 2.

Standard and nonstandard models satisfy the same FO properties for ADTs, but the addition of recursively defined functions destroys this. Although the combination of ADTs and recursively defined functions is the primary technical hurdle, we develop completeness results for a theory that also includes background sorts. This is crucial to verify functional programs as they invariably involve background theories.

Formally, in this paper we work with a notion of validity under the combination of theories $\mathcal{T}_{comb}$. We will also henceforth use the extended signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$.

3.4 Validity under Defined Functions

Let *DEF* be a set of definitions def_D for each $D \in \mathcal{D}$. We define the validity of a first-order formula φ under definitions *DEF* by considering pairs of the form (DEF, φ).

Definition 3.3 (Validity of FOL Formulae with Defined Functions). Given $(\mathcal{S}, \mathcal{F}, \mathcal{D})$ with definitions *DEF* of functions in $\mathcal{D}$, we say that a formula φ is $\mathcal{T}$-valid under the definitions iff $\mathit{DEF} \rightarrow \varphi$ is $\mathcal{T}$-valid, i.e., is in $\mathcal{T}$. Note that we are using *DEF* in the formula to mean the conjunction of definitions in that set. We denote this by $\mathcal{T} \models (\mathit{DEF}, \varphi)$. □

We can utilize the above notion in the case of the theory of the standard model or the combined theories, writing $\mathcal{T}_{std} \models (\mathit{DEF}, \varphi)$ or $\mathcal{T}_{comb} \models (\mathit{DEF}, \varphi)$ respectively. As before, if $\mathcal{T}_{comb} \models (\mathit{DEF}, \varphi)$ then $\mathcal{T}_{std} \models (\mathit{DEF}, \varphi)$.

4 A *FLUID* LOGIC

In this section we define our first main contribution: the *FLUID* (First-Order Logic of Universal properties under Inductive Definitions) fragment that captures VCs generated by tools like LH and Leon. The heart of the *FLUID* fragment is a class of inductive definitions called *provably acyclic* definitions.

Recall that we require definitions to terminate on the standard model. We demand in the *FLUID* fragment that definitions also satisfy a *provable acyclicity* condition, which is a notion similar to termination. Intuitively, acyclicity means that when definitions are unrolled, there is no cyclic dependency between the recursive calls. Note terminating functions must be acyclic, but acyclic definitions can be non-terminating. For example, the function *forever* on Lists defined by $forever(x) = forever(Cons(0, x))$ does not terminate, but it is acyclic because the recursive calls do not repeat. We demand in the *FLUID* fragment that the acyclicity property expressed as a first-order formula is *provable* for the recursive definitions[9]. We formulate provable acyclicity below using ranking functions and path conditions, which we first define formally.

An ordered sort $S \in \mathcal{S}$ is one with a binary predicate $<$ such that $<$ forms a strict partial order. Formally, $<$ must satisfy the FO axioms expressing irreflexivity and transitivity under $\mathcal{T}_{comb}$. Note that $<$ need not be well-founded because we only require acyclicity, not termination[10]. Every ADT sort is an ordered sort with respect to the (strict) subterm relationship.

For a recursively defined function $D \in \mathcal{D}$, a *ranking function* for D is a recursively defined function $Rank_D \in \mathcal{D}$ from the domain of D to an ordered sort. We require $\mathcal{D}$ is *stratified*. The stratum of a function $D \in \mathcal{D}$ is a natural number denoted by $strat(D)$. Note that multiple functions can have the same strata. We require that every $D \in \mathcal{D}$ with $strat(D) > 0$ has a ranking function $Rank_D$ whose stratum is *strictly lower* than D. When $strat(D) = 0$, we require that D is unary over an ordered sort, and its ranking function is the identity function. Finally, we require that the definition of a function at stratum i can only call functions of strata lower than or equal to i.

We now define path conditions.

Definition 4.1 (Path Condition). Given a formula ρ[11], we denote by $Path_\rho(\psi, E)$ that the sub-expression (subterm or subformula) E occurs in ρ with path condition ψ. It is the least relation satisfying the following recurrence:

$$
\begin{aligned}
&Path_\rho(True, \rho) && \text{holds}\\
\text{If } &Path_\rho(\psi, \mathsf{ite}(cond, E_1, E_2)) && \text{then } Path_\rho(\psi, cond)\\
\text{If } &Path_\rho(\psi, \mathsf{ite}(cond, E_1, E_2)) && \text{then } Path_\rho(\psi \wedge cond, E_1)\\
\text{If } &Path_\rho(\psi, \mathsf{ite}(cond, E_1, E_2)) && \text{then } Path_\rho(\psi \wedge \neg cond, E_2)\\
\text{If } &Path_\rho(\psi, D(E_1 \ldots, E_n)) \text{ for } D \in \mathcal{D} && \text{then } Path_\rho(\psi, E_j), 1 \leq j \leq n\\
\text{If } &Path_\rho(\psi, \oplus(E_1 \ldots, E_n)) \text{ for } \oplus \neq \mathsf{ite}, \oplus \notin \mathcal{D} && \text{then } Path_\rho(\psi, E_j), 1 \leq j \leq n
\end{aligned}
$$

Informally, the path condition is the conjunction of all the conditions of ite expressions that must be satisfied in order to "reach" the given sub-expression.

We are now ready to define provable acyclicity:

Definition 4.2 (Provably Acyclic Definitions). Given a signature with combined theory $\mathcal{T}_{comb}$ and stratified definitions *DEF*, a definition $def_D \equiv \forall \overline{x}.\, D(\overline{x}) = \rho(\overline{x})$ is provably acyclic if for every $G(\overline{t})$

[9] A subtle point here is that even though terminating functions are acyclic, they need not be *provably* acyclic (see Appendix A.1 for an example). Therefore, termination and provable acyclicity are incomparable.

[10] Well-foundedness is not expressible in FOL anyway.

[11] ρ is usually the body of a recursively defined function

($G \in \mathcal{D}$) occurring in ρ with $strat(G) = strat(D)$, $Rank_G$ and $Rank_D$ have the same range sort, and furthermore, for every ψ such that $Path_\rho(\psi, G(\bar{t}))$:

$$\mathcal{T}_{comb} \models \left(\bigwedge_{strat(H) < strat(D)} def_H \right) \rightarrow \left(\psi \rightarrow Rank_G(\bar{t}) < Rank_D(\bar{x}) \right)$$

where the overloaded symbol $<$ represents an order predicate in the corresponding sort. □

Informally, the above definition says that the arguments to recursive calls must be *provably* (w.r.t $\mathcal{T}_{comb}$) smaller than the input arguments as measured using ranking functions. Although we say 'provable', note that the definition uses semantic entailment ($\models$). However, these two notions are the same since FOL is complete. Provable acyclicity ensures that when a definition is unrolled, there is no cyclic dependency between recursive calls as the arguments will always decrease. We can use the definitions of functions in lower strata and the path condition to the recursive call to establish this property. We give an example below.

Example 4.3 (Sorted List Merge). Consider the usual function $merge(x, y)$ for merging sorted lists. Let its stratum be 1, with its ranking function being the sum of lengths of x and y. The stratum of the length function *length* is 0.

Consider the recursive call $merge(tail(x), y)$. The path condition in this case is $x \neq Nil \wedge y \neq Nil \wedge head(x) < head(y)$. We can show that this call has smaller rank, i.e., $(length(tail(x)) + length(y)) < (length(x) + length(y))$ using the definition of *length* and the path condition ($x \neq Nil$ ensures that the term $tail(x)$ is well-defined). We can also show similarly that the other recursive call has smaller arguments, and therefore *merge* is provably acyclic.

We can also show that *length* is provably acyclic. Since its stratum is 0, its ranking function must be identity, therefore we have to show that the arguments to recursive calls must themselves decrease. This is evidently true since $length(x)$ recurses on $tail(x)$, which is smaller according to the ADT subterm ordering. □

Aside. We note here some subtleties in the definition of provable acyclicity. First, the relation $<$ is a *mathematical* one, and does not need to be part of the signature or logically defined. Consequently, Definition 4.2 can be established by a user/system in any way. For example, if $<$ denotes the subterm ordering on ADT Lists, then a system can trivially deduce that $tail(x) < x$. In particular, a definition that recurses on destructions of the called arguments is immediately provably acyclic. Second, ADTs are an ordered sort regardless of the choice of axiomatization because the subterm relation is an order in any model that satisfies a complete axiomatization of ADTs, including nonstandard models (even rogue ones). Therefore, we do not need ADT signatures/axiomatizations with an explicit subterm predicate [Kovács et al. 2017]. Third, observe that ranks need not be well-founded as we only require acyclicity, not termination. Contrary to the usual ranking functions in literature, ranks need not be lower-bounded. For example, the function *forever* defined above is provably acyclic because we can say $Rank(Cons(0, x)) < Rank(x)$ with the rank being negative of the length, which has no lower bound.

Our fragment is very general and includes most definitions we know that tools use. In practice, provable acyclicity is satisfied when functional programs are proved terminating. This is because, to the best of our knowledge, every tool that proves functional programs terminating uses ranking functions that map arguments to a well-founded order (typically tuples of natural numbers, often associated with the size of ADTs), and shows that (1) the ranking function decreases (according to some order relation $<$) on recursive calls, and (2) the order $<$ on which the ranking function decreases is well-founded. Condition (1) is precisely the property in Definition 4.2!

Intuitively, provable acyclicity generalizes the idea of proving termination. Termination makes sense on a standard model, but in a nonstandard model ADT elements can have "infinite tails" and

therefore a function that terminates on the standard model can be nonterminating on a nonstandard model[12]. In contrast, provable acyclicity makes sense on all models, standard and nonstandard. We show that in any $\mathcal{T}_{comb}$ model, provably acyclic definitions are always satisfiable (though they may not have a unique interpretation). Formally (see Appendix A.2.1 for a proof):

Theorem 4.4. *Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set of stratified definitions DEF that are provably acyclic, and a model $\mathcal{M}$ of $\mathcal{T}_{comb}$, there exists a model $\mathcal{M}'$ of $\mathcal{T}_{comb}$ such that the interpretation of symbols in $\mathcal{F}$ coincides with $\mathcal{M}$ and interpretations of symbols in $\mathcal{D}$ satisfy their definitions.*

FLUID. We now define the *FLUID* fragment.

Definition 4.5 (FLUID Fragment). Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$ and a set of stratified definitions *DEF* for the symbols in $\mathcal{D}$, the pair (DEF, φ) is in the *FLUID* fragment if (1) every definition in *DEF* is provably acyclic, and (2) φ is purely universally quantified.

5 COMPLETENESS OF RECURSIVE FUNCTION UNFOLDING AND QUANTIFIER-FREE REASONING

In this section we describe the algorithm UQFR, based on Unfolding definitions followed by Quantifier-Free Reasoning, for checking validity of universal properties. We show that the algorithm intrinsically only proves theorems in the combined theory $\mathcal{T}_{comb}$. We then prove the main technical contribution of this paper, that the algorithm is *complete* for $\mathcal{T}_{comb}$. Let us fix a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$ through this section. Recall that $\mathcal{T}_{comb}$ represents the combined theory for the foreground and background sorts, with $\mathcal{D}$ being uninterpreted. Fix also the theory of the standard model $\mathcal{T}_{std}$.

We require for our algorithm that $\mathcal{T}_{comb}$-validity is *decidable* for *quantifier-free formulas*, and that the quantifier-free fragments of $\mathcal{T}_{comb}$ and $\mathcal{T}_{std}$ are identical. We are agnostic to the choice or presence of an axiomatization for the theories and have no other constraints on $\mathcal{T}_{comb}$. This assumption is satisfied for several combined theories, including those that admit Nelson-Oppen combination [Nelson and Oppen 1979; Tinelli and Zarba 2004] e.g. ADTs, linear arithmetic, reals, etc. In fact, such theories also admit efficient decision procedures as evidenced by SMT solvers [Barrett et al. 2011a; de Moura and Bjørner 2008]. Checking validity is achieved by negating and checking for unsatisfiability. Note that the *quantified* theories $\mathcal{T}_{std}$ and $\mathcal{T}_{comb}$ are however typically different; see Section 3.3.

5.1 UQFR Algorithm

The high-level picture of the algorithm is as follows: presented with a set of definitions *DEF* and a quantifier-free formula φ, UQFR systematically unfolds the definitions on terms on which functions are applied and dispatches the resulting quantifier-free formulas to a decision procedure for satisfiability. We first provide some definitions that are useful in describing the algorithm.

Definition 5.1 ($\mathcal{D}$-Application). A $\mathcal{D}$-application is a pair $(D, \bar{t})$ for $D \in \mathcal{D}$ and a tuple of terms $\bar{t} = (t_1, t_2 \ldots, t_r)$ such that $D(\bar{t})$ is well-formed, i.e., D has signature $\sigma_1 \times \sigma_2 \ldots \times \sigma_r \rightarrow \sigma$ and t_i is of type σ_i for $1 \leq i \leq r$. A $\mathcal{D}$-application $(D, \bar{t})$ occurs in a formula ψ if $D(\bar{t})$ occurs in ψ. □

Definition 5.2 (Definition Unfolding). Let $\varphi \equiv \forall x_1. \forall x_2. \ldots \forall x_n. \psi$ be a universally quantified formula such that ψ is quantifier free. The instantiation of φ with a tuple of terms $\bar{t} \equiv (u_1, u_2, \ldots u_n)$, written $\varphi[\bar{t}]$, is the quantifier-free formula $\psi[u_1/x_1, \ldots u_n/x_n]$. □

[12]Here we mean nonterminating in the sense that the evaluation procedure described in Section 3.2 does not terminate for all inputs drawn from the nonstandard model.

UQFR$(\mathcal{S}; \mathcal{F}; \mathcal{D}; \mathcal{T}_{std})$
Input: (DEF, φ) such that φ is universally quantified, with $DEF = \{def_D \mid D \in \mathcal{D}\}$
Output(s): *VALID* (when it terminates)
Imports: QFreeSAT for deciding $\mathcal{T}_{std}$-satisfiability of quantifier-free formulas
(1) $formulas := \{\neg\varphi\}$ // Negate the formula and check for satisfiability
(2) While *True*
(3) $res :=$ QFreeSAT$(formulas)$ // Check sat of $\neg\varphi$ with current set of unfoldings
(4) If $(res = UNSAT)$ Then
(5) Return *VALID* // (DEF, φ) is valid
(6) Else
(7) // Compute $\mathcal{D}$-applications occurring in *formulas*
(8) $\mathcal{D}\_applications := \{(D, \bar{t}) \mid D(\bar{t})$ occurs in ψ for $\psi \in formulas\}$
(9) // Unfold the definitions and add them to formulas
(10) $formulas := formulas \cup DEF[\mathcal{D}\_applications]$

Fig. 1. Algorithm for Unfolding Definitions followed by Quantifier-Free Reasoning

Given a set of C of $\mathcal{D}$-applications and a set $DEF = \{def_D \mid D \in \mathcal{D}\}$ of definitions we denote $DEF[C] = \{def_D[\bar{t}] \mid (D, \bar{t}) \in C\}$. Informally, $DEF[C]$ is the set of quantifier-free formulas that contains all the unfoldings of definitions for functions D on arguments $\bar{t}$ given by C.

Algorithm Description. Figure 1 shows the pseudocode for the algorithm UQFR, parameterized by the signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{std})$ with the theory of the standard model. It takes as input a set of definitions $DEF = \{def_D \mid D \in \mathcal{D}\}$ and a formula φ such that φ is universally quantified. UQFR attempts to prove validity by establishing unsatisfiability of the negation $DEF \wedge \neg\varphi$ (see Definition 3.3 to see that these are equivalent). Finally, UQFR also assumes access to an external procedure QFreeSAT that checks the satisfiability of quantifier-free formulas with respect to the theory of the standard model $\mathcal{T}_{std}$. It takes as input a set of formulas and outputs *SAT* if the conjunction of the formulas is $\mathcal{T}_{std}$-satisfiable and *UNSAT* otherwise.

The algorithm maintains a set *formulas* of quantifier-free formulas consisting of $\neg\varphi$ along with finitely many unfoldings of the definitions. If this set is unsatisfiable then the formula $\wedge DEF \wedge \neg\varphi$ is unsatisfiable as well, i.e., φ is valid under *DEF*. Initially the set contains only $\neg\varphi$. Since φ is purely universal, we treat $\neg\varphi$ as a quantifier-free formula by adding the existentially quantified variables as new ground terms (constants) in our signature.

At a general point in the algorithm (line 2), we check the $\mathcal{T}_{std}$-satisfiability of *formulas* using the external procedure QFreeSAT (line 3). Note that although there is a unique valuation for every $D \in \mathcal{D}$ on the standard model consistent with *DEF*, the set *formulas* only enforces this consistency for finitely many unfoldings of *DEF* and otherwise treats the symbols in $\mathcal{D}$ as uninterpreted, which is an over-approximation. If *formulas* is unsatisfiable we exit and return *VALID*. Otherwise, we refine our approximation by unfolding the definitions on more terms. We compute the set of $\mathcal{D}$-application terms occurring in *formulas* (line 8), add the corresponding unfoldings of definitions to *formulas* (line 10), and go back to the beginning of the loop. Observe that if the algorithm is not able to prove the unsatisfiability of $\neg\varphi$ using any amount of unfoldings then it does not terminate.

5.2 Soundness and Completeness of UQFR under Combined Theories

In this section we prove the primary contribution of this work, namely that UQFR is complete for $\mathcal{T}_{comb}$-validity of *FLUID* formulas. We first show the soundness of UQFR.

Theorem 5.3 (Soundness of UQFR w.r.t $\mathcal{T}_{comb}$). *If UQFR$(\mathcal{S};\mathcal{F};\mathcal{D};\mathcal{T}_{std})$ terminates on (DEF,φ) then $\mathcal{T}_{comb} \models (DEF,\varphi)$.*

Proof. In each round of the algorithm the set *formulas* is of the form $DEF[C] \cup \{\neg\varphi\}$, where $DEF[C]$ contains unfoldings (i.e., instantiations) of DEF on a set C of $\mathcal{D}$-applications. Therefore, if UQFR terminates then $DEF[C] \wedge \neg\varphi$ is unsatisfiable with respect to $\mathcal{T}_{std}$ (line 3).

Now, QFreeSAT can also be seen as a satisfiability procedure for the combined theory $\mathcal{T}_{comb}$ since the input formulas are *quantifier-free*. We hence have that $DEF[C] \wedge \neg\varphi$ is unsatisfiable with respect to $\mathcal{T}_{comb}$, which yields $DEF \wedge \neg\varphi$ is unsatisfiable with respect to $\mathcal{T}_{comb}$, i.e., $\mathcal{T}_{comb} \models (DEF,\varphi)$. □

We showed in Section 3.3 that typically $\mathcal{T}_{std}$ is strictly larger than $\mathcal{T}_{comb}$. The above result shows that the proving power of UQFR is in fact bounded by $\mathcal{T}_{comb}$. Therefore, not only are there valid theorems in $\mathcal{T}_{std}$ that are not valid in $\mathcal{T}_{comb}$, but it is also the case that UQFR (and hence systems such as LH and Leon) will never be able to prove those theorems.

We now show that UQFR is complete.

Theorem 5.4 (Completeness of UQFR w.r.t $\mathcal{T}_{comb}$ for *FLUID*). *If (DEF,φ) belongs to the FLUID fragment (Definition 4.5) and $\mathcal{T}_{comb} \models (DEF,\varphi)$, then UQFR$(\mathcal{S};\mathcal{F};\mathcal{D};\mathcal{T}_{std})$ terminates on (DEF,φ) and reports it valid.*

We dedicate the rest of this section to the proof of the completeness theorem.

Prologue: Theorem Simplification and Reduction to Model Construction

We make some simplifications for ease of presentation. First, we assume that *DEF* has only one stratum. We provide a generalization of the argument made here to multiple strata in Appendix A.2. Second, we assume without loss of generality that the signature $(\mathcal{S},\mathcal{F},\mathcal{D},\mathcal{T}_{comb})$ is such that if a formula Γ is satisfiable in a $\mathcal{T}_{comb}$ model, then it is satisfiable in a Herbrand model consisting of the terms occurring in Γ and closed under the applications of functions in $\mathcal{F} \cup \mathcal{D}$. This can always be done by Skolemizing $\mathcal{T}_{comb}$ and expanding $\mathcal{F}$ with new function symbols.

We first rewrite the statement of the theorem to an equivalent one. Consider the value of the sets *formulas* and $\mathcal{D}\_applications$ through the algorithm:

$$formulas_0 = \{\neg\varphi\} \qquad \text{(initial value)}$$

$$\mathcal{D}\_applications_i = \{(D,\bar{t}) \mid D(\bar{t}) \text{ occurs in } \psi \in formulas_{i-1}\} \qquad (i > 0)$$

$$formulas_i = formulas_{i-1} \cup DEF[\mathcal{D}\_applications_i] \qquad (i > 0)$$

where the subscript i denotes their values in the i^{th} *round* of the outermost loop on line 2. Observe that $formulas_i \subseteq formulas_j$ and $\mathcal{D}\_applications_i \subseteq \mathcal{D}\_applications_j$ for every $j > i$. The completeness result can then be stated as follows:

Theorem 5.5 (Completeness of UQFR w.r.t $\mathcal{T}_{comb}$). *If $\mathcal{T}_{comb} \models (DEF,\varphi)$ then $formulas_i$ is $\mathcal{T}_{comb}$-unsatisfiable for some $i \geq 0$.*

Note that the above theorem implies that UQFR is complete for $\mathcal{T}_{comb}$-validity because if for some i we have that $formulas_i$ is $\mathcal{T}_{comb}$-unsatisfiable, then it is also $\mathcal{T}_{std}$-unsatisfiable, therefore the algorithm will terminate in round i. By the soundness theorem (Theorem 5.2), (DEF,φ) is $\mathcal{T}_{comb}$-valid.

We show the contrapositive of the above statement. Let us assume that $formulas_i$ is $\mathcal{T}_{comb}$-satisfiable for every $i \in \mathbb{N}$. We show that $DEF \wedge \neg\varphi$ is $\mathcal{T}_{comb}$-satisfiable. Specifically, we construct a $\mathcal{T}_{comb}$ model $\mathcal{N}$ such that $\mathcal{N} \models DEF \wedge \neg\varphi$.

Proof Plan. We construct $\mathcal{N}$ in two stages:

Stage 1: We first use the assumption that $formulas_i$ is $\mathcal{T}_{comb}$-satisfiable for every $i \in \mathbb{N}$ to construct a model $\mathcal{M}$ (using compactness) that satisfies $\bigcup_{i\geq 1} DEF[\mathcal{D}\_applications_i]$ and $\neg\varphi$. Note that this model need not satisfy *DEF everywhere* (as we have only instantiated definitions for a subset of terms).

Stage 2: In this stage we take the model $\mathcal{M}$ and consider a finite set K of pairs of the form $(D, \bar{t})$ such that the interpretation of D in $\mathcal{M}$ does not satisfy the definition of D on $\bar{t}$. We show that we can 'repair' the model so that definition of D now holds on $\bar{t}$ for every $(D, \bar{t}) \in K$. We then show that definitions can be repaired everywhere using a compactness argument. This results in the model $\mathcal{N}$ we seek.

Stage 1: Model of Infinite Instantiations

We recall the compactness theorem for FOL under combinations of theories.

Proposition 5.6 (FOL Compactness with Theories). *Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set of formulas Γ (finite or infinite) is $\mathcal{T}_{comb}$-satisfiable if and only if every finite subset of Γ is $\mathcal{T}_{comb}$-satisfiable.* □

From our assumption we know that $formulas_i$ is $\mathcal{T}_{comb}$-satisfiable for every i. Using compactness and the fact that $formulas_i$ form an increasing sequence w.r.t $\subseteq$, it follows that the infinite set $Inf = \bigcup_{i\in\mathbb{N}} formulas_i$ is $\mathcal{T}_{comb}$-satisfiable. We rewrite this as $Inf = \{\neg\varphi\} \cup \bigcup_{i\geq 1} DEF[\mathcal{D}\_applications_i]$. Let $\mathcal{M}$ be a $\mathcal{T}_{comb}$-model that satisfies Inf. From our simplifying assumptions, we can assume that $\mathcal{M}$ is a Herbrand model. It satisfies $\neg\varphi$ and satisfies the definitions only on certain tuples, namely for $(D, \bar{t}) \in \bigcup_{i\geq 1} \mathcal{D}\_applications_i$.

Note here that if the model $\mathcal{M}$ happened to be the *standard model* the repair we wish to do would be trivial as def_D is uniquely defined (see Proposition 3.1) for each $D \in \mathcal{D}$ and we can simply 'complete' the model with the correct valuations. However, $\mathcal{M}$ can be a nonstandard model, and this results in the nontrivial aspects of our construction below.

Stage 2: Computational Closure and Model Repair

The reason we can repair $\mathcal{M}$ is because the set $\bigcup_{i\geq 1} \mathcal{D}\_applications_i$ has a special property: it is *computationally closed*. We define this property below.

Definition 5.7 (Computationally Closed Set). Let Γ be a set of quantifier-free formulas. A set C of $\mathcal{D}$-applications is said to be computationally closed with respect to Γ if: (1) if $D(\bar{t})$ occurs in some formula in Γ then $(D, \bar{t}) \in C$, and (2) if $(D, \bar{t}_1) \in C$ and a $\mathcal{D}$-application $(G, \bar{t}_2)$ occurs in $def_D[\bar{t}_1]$ then $(G, \bar{t}_2) \in C$.

Intuitively, for a recursively defined function D, the computational closure of a term $D(t)$ contains all the recursive calls (at any level) made by a call to D on t, where we represent a recursive call to a function G on a term r by the $\mathcal{D}$-application (G, r). The set is called a computational closure because it is the set of calls that occur when 'computing' the value of D on t symbolically. The computational closure of a formula is then the union of the computational closures of all terms of the form $D(t)$ occurring in the formula. For example, consider the length function *length* on Lists. The computational closure of $length(Cons(1, Nil))$ is the set $\{(length, Cons(1, Nil)),\ (length, Nil)\}$. Similarly, the computational closure of $length(x)$ is $\{(length, x),\ (length, tail(x)),\ (length, tail(tail(x))),\ \ldots\}$.

Using the above definition we can see that $\bigcup_{i\geq 1} \mathcal{D}\_applications_i$ is computationally closed for $\neg\varphi$. We now show that we can repair definitions everywhere on a Herbrand model if the definitions are

already satisfied on a computationally closed sub-universe. Using this result, we can repair $\mathcal{M}$ so that definitions are satisfied everywhere, which is what we want.

Lemma 5.8 (Finite Repair outside Computational Closure). *Let $\mathcal{M}$ be a Herbrand model, Γ a set of quantifier-free formulae, C a computationally closed set for Γ, and K a finite set of $\mathcal{D}$-applications not in C. Let $\mathcal{M}$ satisfy $DEF[C] \cup \Gamma$. Then there exists a model $\mathcal{M}'$ that satisfies $DEF[K] \cup DEF[C] \cup \Gamma$.*

Proof. Observe that if K is singleton, say $\{(D, \bar{t})\}$, we can construct $\mathcal{M}'$ by simply 'updating' the interpretation of D on $\bar{t}$ according to the definition. Formally, the model $\mathcal{M}[D(\bar{t}) := [\![\rho(\bar{t})]\!]_{\mathcal{M}}]$ satisfies $DEF[\{(D, \bar{t})\}] \wedge DEF[C] \wedge \Gamma$. Here $\mathcal{M}[(D, \bar{t}) := v]$ denotes an updated model whose interpretation of $D(\bar{t})$ is v but is otherwise identical to $\mathcal{M}$. We also use $[\![\cdot]\!]_{\mathcal{M}}$ to denote the interpretation of $\mathcal{M}$. The correctness of this construction follows from the fact that the definitions over C are satisfied despite the update since C is computationally closed. Consequently the satisfaction of Γ is also unaffected because if $G(\bar{r})$ occurs in Γ then $(G, \bar{r})$ belongs to C.

To show that $DEF[K]$ is satisfiable for an arbitrary finite subset K, we take $\mathcal{M}$ and apply updates as above on each pair in K. However, we have to do this carefully so that each repair does not break any previous repairs. Fix a set K' and a model $\mathcal{M}'$ such that $K' \subseteq K$ and $\mathcal{M}'$ is $\mathcal{M}$ with updated with the fixes for the elements in K'. Initially $K' = K$ and $\mathcal{M}' = \mathcal{M}$. We describe below a mechanism $Minimal(K, K', \mathcal{M}')$ to choose a 'minimal' element in K that has not been fixed yet, and repair it as described above.
$Minimal(K, K', \mathcal{M}')$ is as follows:

(1) Pick an arbitrary element $(D, \bar{t}) \in (K \setminus K')$. Let the body of def_D be ρ.
(2) We evaluate $\rho(\bar{t})$ on $\mathcal{M}'$ in the following way: subterms must be evaluated before superterms, and for conditionals we evaluate the condition first and then only evaluate the appropriate branch.
(3) If the evaluation as described above does not encounter any element in $K \setminus K'$, then return $(D, \bar{t})$.
(4) If the evaluation of $\rho(\bar{t})$ encounters a term $G(\bar{r})$ such that $(G, \bar{r}) \in (K \setminus K')$, we recurse, going back to Step (2) and evaluating $\tau(\bar{r})$ where τ is the body of def_G.

Informally, this mechanism has the flavor of an *eager evaluation*, in that we evaluate $\rho(\bar{t})$ eagerly, following the evaluation procedure down (recursively) to a minimal unfixed $\mathcal{D}$-application in K.

Finally, when the procedure returns an element $(H, \bar{u})$, we add it to K' and update $\mathcal{M}'$ with the repair for $(H, \bar{u})$. We then repeat this process of picking a minimal element and repairing the model on it until all elements in K are fixed. This completes our construction, and the model $\mathcal{M}'$ obtained at the end of all the fixes is the model we desire.

A subtle point in the construction is the termination of $Repair(K, K', \mathcal{M}')$ as the choice of minimal element is not well-defined otherwise. If the mechanism does not terminate, it must be because the evaluation of some $D(\bar{t})$ encounters itself. However, this is impossible as definitions are provably acyclic (Definition 4.2). Formally, we have the following proposition:

Proposition 5.9. *Let $(D, \bar{t})$ be a $\mathcal{D}$-application in K, ρ be the body of the definition of D, and $\mathcal{M}$ be a model of $\mathcal{T}_{comb}$. Let $(G, \bar{r})$ be a $\mathcal{D}$-application such that $G(\bar{r})$ is a sub-expression of $\rho(\bar{t})$ and the evaluation of $\rho(\bar{t})$ in $\mathcal{M}$ as performed in the Repair mechanism above encounters $G(\bar{r})$. Further, let ψ be the path condition of the sub-expression $G(\bar{r})$ that is reached (see Definition 4.1). Then, we have that $\mathcal{M} \models \psi$.*

We skip the proof of this proposition as it follows trivially from the description of the evaluation mechanism and Definition 4.1. Now, from the definition of provable acyclicity (Definition 4.2), we know:

$$\mathcal{T}_{comb} \models \left(\bigwedge_{strat(H) < strat(D)} def_H \right) \rightarrow \psi \rightarrow Rank_G(\bar{r}) < Rank_D(\bar{t})$$

Per our assumption we have only one stratum, so the set $\{H\}_{H \in \mathcal{D}, strat(H) < strat(D)}$ is empty. Since $\mathcal{M}$ is a $\mathcal{T}_{comb}$ model that satisfies ψ, we obtain $\mathcal{M} \models Rank_G(\bar{r}) < Rank_D(\bar{t})$. Therefore, if updating $D(\bar{t})$ requires updating $G(\bar{r})$, then the rank of $G(\bar{r})$ is smaller. Therefore, each recursive call of *Repair* is made on a smaller element of K, and therefore the evaluation of $D(\bar{t})$ cannot depend on itself. The mechanism for picking a minimal element is indeed well-defined and we can produce at the end of the procedure a model $\mathcal{M}'$ that satisfies $DEF[K] \cup DEF[C] \cup \Gamma$.

End of proof of Lemma 5.8 □

Repair for All Tuples. We now show that we can repair definitions everywhere, i.e., on arbitrarily large sets of $\mathcal{D}$-applications. We capture this result separately in Lemma A.2 in Appendix A.2 and simply provide the arguments relevant to the completeness proof here. Recall that we have a $\mathcal{T}_{comb}$ model $\mathcal{M}$ from Stage 1 such that $\mathcal{M} \models DEF[C] \cup \Gamma$ for a computationally closed set C. Consider the set $DApp$ of all possible $\mathcal{D}$-applications. Note that $C \subseteq DApp$.

Formally, we show that $DEF[DApp] \cup \Gamma$ is $\mathcal{T}_{comb}$-satisfiable. First, rewrite the formulas as $DEF[C] \cup DEF[\overline{C}] \cup \Gamma$, where $\overline{C} = DApp \setminus C$ is the complement of C. Since $DEF[C] \cup \Gamma$ is already satisfiable, it is sufficient to show that $DEF[C] \cup \Gamma \cup B$ is satisfiable for an arbitrary finite subset B of $DEF[\overline{C}]$ and apply the compactness theorem. Observe that a finite subset of $DEF[\overline{C}]$ is of the form $DEF[K]$ for a finite set $K \in DApp$. We are now done, since we know that $DEF[C] \cup \Gamma \cup DEF[K]$ is satisfiable from Lemma 5.8.

Consider a model $\mathcal{N}$ such that $\mathcal{N} \models DEF[DApp] \cup \Gamma$. Since the formulae are all universally quantified, we can assume that $\mathcal{N}$ is a Herbrand model, therefore the universe of $\mathcal{N}$ (across the sorts) is in fact the set of all possible terms. Therefore, we can simply replace $DEF[DApp]$ by DEF and conclude that $\mathcal{N} \models DEF \cup \Gamma$, as desired.

This concludes the proof of the completeness theorem (Theorem 5.4). □

6 *FLUID* REASONING IN LIQUID HASKELL

Next, let us see how the Liquid Haskell verifier (LH) employs a particular instance of *FLUID* reasoning referred to by the tool as *reflection* and *proof by logical evaluation* (PLE). We show how a user might use LH to develop a small library of theorems about Peano numbers to illustrate why it can be viewed as *FLUID* reasoning, why its *FLUID*-style instantiation heuristics are effective in practice, and, perhaps more importantly, why extra information is *really* required from the user when instantiation fails.

Peano Addition. Consider the definition of *Peano* numbers

```
data Peano = Z | S Peano
```

As described in Section 3.1, Liquid Haskell uses the above definition to generate an ADT `Peano` with (1) two *constructors* Z and S, (2) two *recognizers* isZ and isS, and (3) a single *destructor* $pred$. Next, suppose the user writes the following function that recursively defines `Peano` addition as

```
plus :: Peano → Peano → Peano
plus Z     m = m
plus (S n) m = S (plus n m)
```

LH generates a *definition* for `plus` which is an "axiom" constraining the interpretation of *plus* [Vazou et al. 2018]

$$def_{plus} \equiv \forall n, m.\ plus(n, m) = ite(isZ(n),\ m,\ S(plus(pred(n), m))) \quad (1)$$

6.1 Proof by Instantiation

Propositions as Types. Suppose we wish to verify that the addition of `Z` is an identity function, i.e. the proposition $\forall n : Peano.\ plus(Z, n) = n$. In LH, a user uses the recipe of "Propositions as Types" to *specify* the property as a type, and *verify* it via a function `zeroL` that inhabits the type:

```
zeroL :: n:Peano → { plus Z n == n }
zeroL n = ()
```

In the above type signature, the *input parameter* has the effect of quantifying over all n, and the *output post-condition* stipulates the particular property that must hold for each n [Wadler 2015].

Programs as Proofs. To check this proof, LH generates a VC $def_{plus} \rightarrow \forall n.plus(Z, n) = n$. Next, it uses *logical evaluation* (PLE) [Vazou et al. 2018] to instantiate the definition of *plus* (1) at (Z, n) to get the instantiated VC $\forall n.\ def_{plus}[Z, n] \rightarrow plus(Z, n) = n$ using the instantiation

$def_{plus}[Z, n] \equiv (plus(Z, n) = ite(isZ(Z),\ n,\ S(plus(pred(Z), n))$

The SMT solver proves the above instantiated VC is valid even when *plus* is uninterpreted, thereby verifying that `plus Z` is an identity function.

6.2 Proof by Induction

LH makes no attempt to automate inductive proofs. Instead, the programmer must *explicate induction via recursion*, by writing programs where the induction hypothesis is made explicit in the VC via the asserted post-conditions of recursive calls to smaller inputs. As an example, suppose that we wish to verify that the definition of `plus` is commutative. As before, the programmer would start by specifying the above proposition as the type shown at the top of Figure 2, and might attempt a direct proof `comm n m = ()`[13] that would yield the *FLUID* VC

$$def_{plus} \rightarrow (\forall n, m.\ plus(n, m) = plus(m, n)) \tag{2}$$

Sadly, PLE does not find any suitable instantiations, and so the SMT solver *cannot* prove the above is valid when *plus* is uninterpreted and hence *rejects* the code on the left.

Rogue Nonstandard Model. Did LH simply give up too early — maybe some carefully chosen instantiations would produce a valid instantiated VC? Surprisingly, this is not the case. In fact, verification fails because (2) is refuted by a rogue nonstandard model (Figure 4 in Appendix A.3.1) where the interpretation for the constructors and destructors respects the ADT axioms for `Peano` and *plus* satisfies its definition, but there exists an element i' such that $plus(i', Z) \neq plus(Z, i')$ in the model. Let us banish such rogue models by proving that adding `Z` on the *right* is also an identity,

$$\forall n : Peano.\ plus(n, Z) = n \tag{3}$$

A direct proof of 3 is doomed: it yields the VC below which is refuted by the model in Figure 4:

$$def_{plus} \rightarrow \forall n.\ plus(n, Z) = n$$

An Inductive Proof. The programmer must spell out an inductive proof as a (recursive) piece of code that yields a VC which *excludes* rogue nonstandard models by explicitly stating the induction hypothesis as an antecedent in the VC. This is achieved via the proof `zeroR` shown on the left in Figure 2. First, we split cases (via a pattern match) on the first argument, treating separately the cases where the argument is `Z` or `S n`. Second, the recursive call to `zeroR n` puts the post-condition of `zeroR` for the *smaller input* `n` as a hypothesis for the new VC

$$def_{plus} \rightarrow (\forall n.\ plus(Z, Z) = Z \wedge plus(n, Z) = n \rightarrow plus(S(n), Z) = S(n))$$

PLE instantiates the definition def_{plus} (1) at (Z, Z) and $(S(n), Z)$ to get the instantiated VC

$def_{plus}[Z, Z] \rightarrow def_{plus}[S(n), Z] \rightarrow (plus(Z, Z) = Z \wedge plus(n, Z) = n \rightarrow plus(S(n), Z) = S(n))$

[13] `()` is a "unit proof" with no extra hints from the user. LH attempts to prove the VC directly given a unit proof.

```
-- Succeeds
zeroR :: n:_ →
  {plus n Z == n}

zeroR Z     = ()
zeroR (S n) = zeroR n
```

```
-- Fails
comm :: n:_ → m:_ →
  {plus n m == plus m n}

comm Z     m = zeroR m
comm (S n) m = comm n m
```

```
-- Succeeds
comm :: n:_ → m:_  →
  {plus n m == plus m n}

comm Z     m = zeroR m
comm (S n) m = comm n m
                 && succR m n
```

Fig. 2. Proof of the commutativity of *Peano* addition: The explicit case-splitting, recursion and "lemma application" are needed to eliminate rogue nonstandard models.

The instantiated VC is valid even when *plus* is uninterpreted, thus proving (3). In essence, the (well-founded) *recursive call* to `zeroR` establishes the *induction hypothesis* for the smaller `n`, thereby eliminating the rogue nonstandard models, letting us verify the proposition for *any* `Peano n`.

6.3 Proof by Lemmas

Next, let us see how to use auxiliary lemmas like `zeroR` to eliminate rogue nonstandard models that thwarted the direct proof of the commutativity of `plus`. First, the programmer might attempt an inductive proof (like the `zeroR`) as shown in the middle in Figure 2: split cases on whether the first parameter is `Z` or `S n`. In the base case, they would *call* `zeroR m` to eliminate the rogue nonstandard model where $plus(i', 0) \neq plus(0, i')$ (Figure 4). In the inductive case, they would recursively invoke the induction hypothesis via recursively calling `comm n m`. This time, LH generates the VC

$$def_{plus} \rightarrow (\forall n, m.\ (plus(m, Z) = m \rightarrow plus(Z, m) = plus(m, Z)) \\ \wedge\ (plus(n, m) = plus(m, n) \rightarrow plus(S(n), m) = plus(m, S(n)))) \tag{4}$$

Thanks to the equality asserted by the use of the "lemma" `zeroR m`, the first conjunct can proved valid via the instantiation $plus[(m, Z)]$. However, the second conjunct is invalid *despite* the recursive (inductive) call to `comm n m` because of a *different* rogue nonstandard model for *plus* that falsifies the second conjunct! Again, we write a lemma `succR` to eliminate the new rogue nonstandard model. Together, these lemmas yield a VC with the strengthened antecedents that preclude the above rogue nonstandard models, allowing PLE's *FLUID* instantiation to prove the VC as shown by the "Succeeds" proof on the right in Figure 2. See Appendix A.3.2 for descriptions of the rogue nonstandard model(s), the `succR` lemma, and the VCs.

6.4 Rogue Nonstandard Models in Proofs about Data Structures

We use the simple `Peano` datatype to illustrate how LH implements *FLUID* reasoning, and how direct proofs can fail due to rogue nonstandard models which can be eliminated via explicit induction (recursion) and lemmas (function calls). Similar phenomena occur when verifying more complicated properties. Consider the datatype of finite maps from keys (`k`) to values (`v`)

```
data Map k v = Leaf | Node k v (Map k v) (Map k v)
```

Figure 3 shows the code for two functions that respectively `get` the value of a `key` from a tree, and `set` the value of a `key` to some new `val` leaving the values of all other keys unchanged. The following proposition is one of McCarthy's two laws that characterize finite maps

$$\forall m, k, v.\ get(set(m, k, v), k) = Just(v)$$

```
get :: Map k v → k → Maybe v
get (Node k v l r) key
 | key == k  = Just v
 | key <  k  = get l key
 | otherwise = get r key
get Leaf _   = Nothing
```

```
set :: Map k v → k → v → Map k v
set (Node k v l r) key val
 | key == k  = Node key val l r
 | key <  k  = Node k v (set l key val) r
 | otherwise = Node k v l (set r key val)
set Leaf k v = Node k v Leaf Leaf
```

Fig. 3. Implementations of get and set functions for Binary Search Tree.

Similar to the example in Section 6.2, the above property needs an induction proof, which we describe in Appendix A.3.3. We describe here the intuitive rogue nonstandard model that falsifies the theorem.

Rogue Nonstandard Model for Search Trees. Let the universe $\mathcal{U}$ be the set of *finite trees* and *infinite non-regular trees*[14] over (k, v) pairs. The interpretation for $get(m, k)$ on a tree m follows the usual path in a binary search tree to find k, even on infinite trees. If the k is found, *get* returns the corresponding value, and if the path ends or continues forever, then *get* returns `Nothing`. The interpretation for *set* is similar, except that on an infinite computation it returns the input tree.

To see why this model refutes the VC, consider the *infinite binary tree m* that is infinite on all paths, and every node of which has key 0 and value 0. Let us call `set`, to set key 1 to the value 1 and try to *get* the value of key 1 after that, i.e. consider $get(set(m, 1, 1), 1)$ By the above interpretation $set(m, 1, 1) = m$, as all paths in m are infinite, and further $get(m, 1) =$ `Nothing` thereby refuting the proposition despite being a model of the ADT theory and the definitions of `get` and `set`. Intuitively, `set` loses the update entirely, and hence `get` returns `Nothing`.

7 *FLUID* REASONING AND REASONING IN LEON

The Leon system and its successor Stainless [Hamza et al. 2019] reason with functional programs [Blanc et al. 2013; Suter et al. 2010, 2011] using techniques broadly similar to LH and UQFR. Leon reasons about Scala programs with quantifier-free pre/post conditions, and the recursively defined functions occurring in annotations are written in Scala as terminating functions. It also automates certain induction proofs, including induction by "stack-height" (akin to Hoare-style reasoning), as well as structural induction on ADTs. Leon caters to other aspects of development as well, including techniques similar to bounded model-checking for finding errors. We do not discuss these aspects here as they are not relevant to our work.

The first observation is that verification conditions for Leon programs can also be modeled as (DEF, φ) in the *FLUID* fragment. Specifically, the property φ is quantifier-free (implicitly universally quantified) and definitions are proven terminating.

While reasoning in Leon also involves unfolding definitions followed by SMT solving, there are important differences in comparison to LH or UQFR. First, whereas LH typically unfolds definitions only once, Leon continually unfolds definitions over multiple rounds similar to UQFR. Second, Leon asserts the contract of a function along with its definition on the given input arguments during the unfolding. In theory, it does so for *every* unfolding, *ad infinitum*. Observe that when expressing a problem as (DEF, φ), contracts cannot be assumed for all subsequent function calls, as that would require universally quantified assumptions. Assuming contracts for subsequent function calls is also strictly more powerful than simply unfolding definitions, as we illustrated in Section 2.2.

[14] A non-regular tree is one that is not isomorphic to any of its proper subtrees. This is a technical condition we require to ensure that the ADTs are *acyclic*, i.e., it is not possible to reach a term by destructing itself.

Reduction to UQFR and Completeness for Leon. We show that the reasoning mechanism in Leon can in fact be captured in the *FLUID* framework and proven using UQFR. More formally, given a pair (DEF, φ) in the *FLUID* fragment with contracts $\{(pre_D, post_D)\}_{D \in \mathcal{D}}$ for the functions, we construct an effectively computable instance (DEF', φ') in the *FLUID* fragment such that running UQFR on (DEF', φ') mimics assuming the contracts in addition to unfolding definitions. We detail this construction in Appendix A.4. in the full version of the paper. The key idea is to construct, for every $D \in \mathcal{D}$, an additional recursively defined predicate $Contract_D$ with the same input signature that returns a Boolean value indicating whether the pre/post condition holds for the input parameters as well as all recursive calls D makes in the computation on these parameters. We then check validity of verification conditions that further assume that the immediate calls to other functions D have $Contract_D$ evaluate to *true*. UQFR applied on this formula mimics the procedure that Leon does and Theorem 5.4 argues the completeness with respect to the underlying combined theory.

As far as we know the above result is new. Prior literature on Leon [Blanc et al. 2013; Suter et al. 2010, 2011] shows soundness of the procedure. Restricted fragments [Suter et al. 2010] involving certain kinds of "measures" (functions from ADTs to background sorts) have been shown to admit complete unfolding based reasoning with respect to the *standard model*, with a *decidable* validity problem. In contrast, we show completeness (i.e., recursively enumerable procedures) for validity with respect to the *combined theory* for a more general class of functions. Further, our logic is *undecidable* (see Section 8), which shows that it is fundamentally different from decidable subclasses reported in prior art [Suter et al. 2010] (see also Section 9).

Our results also show that when theorems are not provable in Leon, there ought to be rogue nonstandard models. We considered a few such examples and were indeed able to construct rogue nonstandard models. For example, Leon fails to prove $rev(rev(x)) = x$ automatically, where *rev* reverses a list. It has a rogue nonstandard model that is eliminated by an inductive lemma provided by the user.

8 FURTHER RESULTS

We show some technical results pertaining to the *FLUID* fragment.

Undecidability of the FLUID Fragment. We show undecidability of the *FLUID* fragment even when the combined theory admits decision procedures for quantifier-free reasoning. In other words, the validity problem for the *FLUID* fragment, for which we proved UQFR is a semi-decision procedure in Section 5.2, does not admit any decision procedures.

Theorem 8.1. *The validity problem for FLUID formulas is undecidable.*

The proof is subtle, reducing the complement of the validity problem, i.e., satisfiability, to the non-halting problem. The subtlety is that validity is with respect to the combined theory, not the standard model. Therefore, simple reductions that encode executions of a Turing machine using ADTs (say, as lists of configurations) and define halting executions as a recursive predicate are made impossible because of nonstandard models. We provide a detailed proof in Appendix A.5.1.

Incompleteness with Terminating Definitions. It is natural to ask whether UQFR is complete for all terminating functions, not just provably acyclic ones. We show that this is not the case.

Theorem 8.2. *There exists a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set DEF of well-defined definitions for $\mathcal{D}$ that are not provably acyclic, and a universally quantified formula φ such that $\mathcal{T}_{comb} \models (DEF, \varphi)$ but UQFR does not terminate.*

We provide this construction in Appendix A.5.2. Note that the above result implies that generalizing definitions to arbitrary universally quantified formulas also leads to incompleteness.

9 RELATED WORK

We discussed the relationship of our work to Liquid Haskell [Rondon et al. 2008; Vazou et al. 2018] and Leon [Blanc et al. 2013; Suter et al. 2011] extensively in Sections 6 and 7. The mechanism for verifying specifications in the functional programming sub-language of Dafny [dafny-lang community 2022; Leino 2010] is also similar. There is much prior work on techniques based on unfolding recursive definitions [Amin et al. 2014; Chamarthi et al. 2011; De Angelis et al. 2018; Leino and Polikarpova 2013], going back to ACL2 [Kaufmann et al. 2000] and the NQTHM prover [Boyer and Moore 1988]. The work on Set-Of-Support resolution [Haifani et al. 2021; Wos et al. 1965] is also similar, but it does not consider background theories.

The work in [Suter et al. 2010] shows that Leon-like reasoning (and UQFR in this paper) is actually a decision procedure for certain restrictive logics. More precisely, it exhibits a logic over restricted classes of user-defined abstractions of ADTs to collections/measures in a decidable sort using catamorphisms, and shows that unfolding function definitions just *once* followed by quantifier-free reasoning is a decision procedure. The classes of such abstractions (*infinitely surjective and sufficiently surjective abstractions*) however are extremely semantically restrictive compared to *FLUID*. In particular, as we show in Section 8, validity of *FLUID* is undecidable, which argues this difference. The work in [Vazou et al. 2018] shows that the PLE heuristic implemented in LH is complete if an 'equational proof' exists, but this result is much weaker than ours, and in fact PLE fails to prove simple theorems that UQFR can prove. See Appendix A.3.4 for an example.

The Why3 system [Bobot et al. 2011, 2015; Filliâtre and Paskevich 2013] also verifies functional programs against contracts, but reduces verification conditions to first-order logic, integrating with several first order logic reasoning engines like Vampire [Kovács and Voronkov 2013]. FO provers such as Vampire [Hajdú et al. 2021] and Zipperposition [Cruanes 2017] support reasoning about ADTs, even providing some automation for inductive reasoning. While it is also possible in our setting to use FO theorem provers to prove formulas of the form $Defs \rightarrow \varphi$, the practical effectiveness of such a reduction has not been evaluated and seems to need overcoming some challenges, especially background theory reasoning; see [Reger et al. 2017].

SMT solvers [Barrett et al. 2011a; Bjorner 1999; de Moura and Bjørner 2008; Reynolds and Blanchette 2017] provide powerful automation for logic reasoning, especially for quantifier-free fragments of decidable combinations of theories [Bradley and Manna 2007; Nelson and Oppen 1979; Tinelli and Harandi 1996]. There is work that develops decidable fragments by building over SMT solvers [Pham and Whalen 2013; Suter et al. 2010] as well as specialized decision procedures [Hojjat and Rümmer 2017; Kapur et al. 2006; Manna et al. 2007; Zhang et al. 2006]. Extensions of the ADT theory have also been studied [Kovács et al. 2017; Rybina and Voronkov 2001].

Prior work on combining theories include Nelson-Oppen decidable combinations of theories [Nelson 1980; Nelson and Oppen 1979; Tinelli and Harandi 1996] and following work [Baader and Ghilardi 2005; Fontaine 2007; Ghilardi 2004; Krstic et al. 2007; Tinelli and Zarba 2005; Wies et al. 2009] extending this result. Local theory extensions [Ihlemann et al. 2008; Sofronie-Stokkermans 2009] have also been employed for constructing decidable logics. Our work can be seen as reasoning with a particular fragment of quantified first-order logic (*FLUID*) over combined theories using particular procedures (especially SMT solvers) that work well in practice in certain domains.

Techniques based on quantifier instantiation have been popular in automatic reasoning of quantified logics, including works on quantifier instantiation for SMT solvers [Barrett et al. 2011b; de Moura and Bjørner 2008; Reynolds 2017]. There are many methods to guide instantiation, such as triggers/E-matching [Amin et al. 2014; Detlefs et al. 2005; Moskal 2009; Rümmer 2012], MBQI [Ge and de Moura 2009], etc. In general, systematic quantifier instantiation in the style of UQFR is not applicable to SMT solvers as the set of terms blows up exponentially.

The work reported in [Löding et al. 2018] that shows completeness of a heuristic in practice called *natural proofs* [Pek et al. 2014; Qiu et al. 2013] is closest to our work, They show that natural proofs can be viewed as reasoning in FOL by instantiating quantifiers using terms over a foreground uninterpreted universe followed by quantifier-free reasoning. They prove that this technique is complete for a *safe fragment* of first-order logic. There are, however, fundamental differences in our work. First, the foreground sorts in our setting are ADT sorts and not uninterpreted. Second, the safe fragment identified in [Löding et al. 2018] is very restrictive as it disallows uninterpreted functions to involve background sorts, which in our setting would mean programs cannot have input parameters of the background sort, like integers. Finally, the quantifier instantiation strategy studied in [Löding et al. 2018] is much more liberal than in our work (and what tools like Liquid Haskell and Leon do). For example, if $\bar{t}$ is a set of terms that occur in a theorem, the instantiation in [Löding et al. 2018] will always instantiate the definition of f on $\bar{t}$ it, while we will do so only when $f(\bar{t})$ occurs in the theorem. Consequently, the proof of our main theorem is quite complex and fundamentally different from the proof of completeness in [Löding et al. 2018].

Triggers are heuristic ways to control quantifier instantiation in SMT solvers, and SMT solvers as well as tools such as Boogie [Leino 2008] and Dafny [Leino 2010] provide mechanisms for specifying triggers, both automatically [Leino and Pit-Claudel 2016] and manually [dafny-lang community 2022]. However, triggers are not simple quantifier instantiations [Leino and Pit-Claudel 2016; Moskal 2009]. Further, trigger-based quantifier instantiation is typically unpredictable and flaky, and to the best of our knowledge, these techniques are also incomplete.

Automating induction has been explored in prior work [Claessen et al. 2013; Cruanes 2017; Hajdú et al. 2020; Ireland and Bundy 1996; Johansson et al. 2010; Passmore et al. 2020] for various specialized fragments [Unno et al. 2017]. In many cases, user help in the form of lemmas is still needed, though there is work on synthesizing inductive lemmas automatically [Murali et al. 2022; Reynolds and Kuncak 2015; Sivaraman et al. 2022; Yang et al. 2019].

10 DISCUSSION

We discuss some salient aspects of our results, their ramifications, and future directions.

On the Completeness Theorem. When proving a theorem of the form $DEF \rightarrow \varphi$ using FOL, it is easy to see that completeness can be achieved if we instantiate *DEF* on *all* possible terms (all possible ADTS formed from all possible terms over the various sorts, such as all possible integers for the integer sort, etc.). However, instantiation performed by tools such as Liquid Haskell and Leon is thrifty, only instantiating function definitions on terms where their applications occur in the current formula. Our completeness theorem proves that even this thrifty instantiation is, in fact, *complete*. This result has practical consequences: thrifty instantiation is computationally cheap, tool designers can comfortably employ it without worrying about missing proofs.

Although we presented our *FLUID* fragment and the completeness of UQFR for it upfront, we spent more than a year on identifying this fragment! The fact that *FLUID* only allows *definitions* is technically crucial. Although *DEF* are universally quantified formulas, definitions do not constrain the space of models of $DEF \rightarrow \varphi$ in significant ways (on the standard model, they are well-defined, and on nonstandard models, they are always satisfiable; see Theorem 4.4). Relaxing this fragment to arbitrary universal quantification destroys completeness (follows from Theorem 8.2 in Section 8). It is remarkable that our completeness result holds even when function definitions are universally quantified over *background sorts* (such as integers), as analogous results of completeness for uninterpreted foreground sorts do not allow such quantification [Löding et al. 2018].

Natural Proofs. The completeness result presented in this paper and the completeness result for *natural proofs* for verification of imperative programs manipulating the heap [Löding et al.

2018] are two results that show completeness of first-order reasoning of programs based on thrifty instantiations employed in practice. In light of this, we in fact think of UQFR for FLUID as *natural proofs for reasoning with first-order logic for recursive programs*.

The two completeness results however have many differences. In natural proofs for imperative programs [Löding et al. 2018], we have an *uninterpreted foreground sort* (useful for modeling arbitrary pointer-based heaps), have universally quantified formulas that quantify only over the foreground sort, and certain restrictions on the logic. In particular, completeness of formula-based quantifier instantiation is guaranteed when functions that involve a background sort in their domain do *not* map to the foreground sort. The work presented in this paper assumes the foreground sort is an ADT sort (FO-axiomatized), allows quantification only for defining functions (which must be provably acyclic), but allows universal quantification in these definitions to span over both the foreground ADT sort as well as the background sorts (i.e., parameters to functions can be of the background sort). One open problem is whether there exists a more general result that extends both these results. For instance, it would be nice to have a result that allows for (user-written) universally quantified lemmas to be incorporated in a completeness result that defines a thrify instantiation scheme for such lemmas (current tools like Liquid Haskell ask for users to provide the lemmas as well as instantiations of them, which of course fits into the fragment defined in this paper). The work on natural proofs for imperative programs [Löding et al. 2018], however, allows already for incorporation of lemmas, as long as quantification is only over the foreground sort, in its completeness result. On the other hand, the results in this paper allow for definitions to allow quantification over background sorts which is disallowed in the completeness results for natural proofs of imperative programs [Löding et al. 2018]. Allowing for definitions involving quantification over background sorts in a completeness result for imperative programs would be interesting.

Future Work. We believe our completeness result not only gives a theoretical foundation for heuristics used by practical verification tools, but also suggests a fundamentally new design paradigm for verification languages. The design of programming languages with specification languages guaranteed to be verifiable using complete techniques can lead to practical automation. Enriching our logic to datatypes beyond ADTs (e.g., abstract data types such as sets, maps, and queues) while supporting complete verification is also an interesting future direction.

Apart from the future directions mentioned above, a particularly interesting extension concerns Higher-Order Functions (HOFs). LH and Stainless do support defining HOFs, but such definitions are beyond the scope of the theory developed in this work. Tools like LH reason with HOFs by defunctionalization [Reynolds 1972], i.e., converting them to FOL definitions by modeling function symbols as constants in a new sort and introducing an uninterpreted function $apply(f, args)$ to model the application of a (higher-order) function f on arguments $args$. However, simply defunctionalizing higher-order definitions and applying UQFR does not yield a completeness result for the appropriate higher-order logic. We conjecture that an analogous completeness theorem does in fact exist for ADTs and background theories with higher-order functions.

That fact that rogue nonstandard models exist when inductive lemmas are needed (since our procedure is FO complete) provides an interesting direction to guide both users and tools towards new lemmas. In particular, one may be able to synthesize finite descriptions of rogue nonstandard models (similar to Section 6.4) using program synthesis, template-based synthesis using DSLs, or even finite model finders [Blanchette and Claessen 2010]. Such models can be presented to users as evidence of proof failure and lemmas suggested by users can be checked against them to evaluate whether they are useful.

Exploiting these models to search automatically for inductive lemmas is also an interesting direction, especially in light of recent work that uses FO models to guide inductive lemma synthesis [Murali et al. 2022] for verifying heap manipulating programs. The Type-1 and Type-3 counterexamples in the work witness the falsehood of the goal and the non-inductiveness of candidate lemmas respectively. Rogue nonstandard models correspond precisely to Type-1 models (since they falsify the goal), and correspond to a variant of Type-3 models (helpful lemmas must not be inductive on rogue nonstandard models). We therefore believe that similar counterexample-guided synthesis techniques using rogue nonstandard models can lead to effective lemma discovery for verifying functional programs.

ACKNOWLEDGMENTS

This work is supported in part by a research grant from Amazon and a Discovery Partners Institute (DPI) science team seed grant.

REFERENCES

Nada Amin, K. Rustan M. Leino, and Tiark Rompf. 2014. Computing with an SMT Solver. In *Tests and Proofs*, Martina Seidl and Nikolai Tillmann (Eds.). Springer International Publishing, Cham, 20–35.

Franz Baader and Silvio Ghilardi. 2005. Connecting Many-Sorted Theories. In *Automated Deduction – CADE-20*, Robert Nieuwenhuis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–294.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011a. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–177.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011b. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177.

Jon Barwise. 1977. *Handbook of Mathematical Logic*. North-Holland Publishing Company, Amsterdam.

Nikolaj Skallerud Bjorner. 1999. *Integrating Decision Procedures for Temporal Verification*. Ph. D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Manna, Zohar. AAI9924398.

Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. 2013. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2489837.2489838

Jasmin Christian Blanchette and Koen Claessen. 2010. Generating Counterexamples for Structural Inductions by Exploiting Nonstandard Models. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Christian G. Fermüller and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–141.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. HAL-Inria, Wroclaw, Poland, 53–64. https://hal.inria.fr/hal-00790310

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2015. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer (STTT)* 17, 6 (2015), 709–727. https://doi.org/10.1007/s10009-014-0314-5 See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

Robert S. Boyer and J. Strother Moore. 1988. *A Computational Logic Handbook*. Academic Press Professional, Inc., USA.

Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg.

Harsh Raju Chamarthi, Peter Dillinger, Panagiotis Manolios, and Daron Vroon. 2011. The ACL2 Sedan Theorem Proving System. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(TACAS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 291–295.

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating Inductive Proofs Using Theory Exploration. In *Automated Deduction – CADE-24*, Maria Paola Bonacina (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 392–406.

Simon Cruanes. 2017. Superposition with Structural Induction. In *Frontiers of Combining Systems*, Clare Dixon and Marcelo Finger (Eds.). Springer International Publishing, Cham, 172–188.

The dafny-lang community. 2022. https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2018. Solving Horn Clauses on Inductive Data Types Without Induction. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 452–469. https://doi.org/10.1017/S1471068418000157

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

Herbert B. Enderton. 1972. *A mathematical introduction to logic.* Academic Press, New York.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128.

Pascal Fontaine. 2007. Combinations of Theories and the Bernays-Schönfinkel-Ramsey Class. In *4th International Verification Workshop - VERIFY'07 (CEUR Workshop Proceedings, Vol. 259)*, Bernhard Beckert (Ed.). HAL-Inria, Bremen, Germany, 37–54. https://hal.inria.fr/inria-00186639 URL : http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-259/paper06.pdf.

Yeting Ge and Leonardo de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–320.

Silvio Ghilardi. 2004. Model-Theoretic Methods in Combined Constraint Satisfiability. *Journal of Automated Reasoning* 33, 3 (01 Oct 2004), 221–249. https://doi.org/10.1007/s10817-004-6241-5

Fajar Haifani, Sophie Tourret, and Christoph Weidenbach. 2021. Generalized Completeness for SOS Resolution and its Application to a New Notion of Relevance. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 327–343.

Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. 2020. Induction with Generalization in Superposition Reasoning. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings* (Bertinoro, Italy). Springer-Verlag, Berlin, Heidelberg, 123–137. https://doi.org/10.1007/978-3-030-53518-6_8

Márton Hajdú, Petra Hozzová, Laura Kovács, and Andrei Voronkov. 2021. Induction with Recursive Definitions in Superposition. In *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design – FMCAD 2021*. TU Wien Academic Press, Wien, 246–255. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_34

Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: Formalized Foundations for the Stainless Verifier. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 166 (oct 2019), 30 pages. https://doi.org/10.1145/3360592

Wilfrid Hodges. 1997. *A Shorter Model Theory.* Cambridge University Press, USA.

Hossein Hojjat and Philipp Rümmer. 2017. Deciding and Interpolating Algebraic Data Types by Reduction. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, Tudor Jebelean, Viorel Negru, Dana Petcu, Daniela Zaharie, Tetsuo Ida, and Stephen M. Watt (Eds.). IEEE Computer Society, Los Alamitos, CA, USA, 145–152. https://doi.org/10.1109/SYNASC.2017.00033

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. 2008. On Local Reasoning in Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–281.

Andrew Ireland and Alan Bundy. 1996. Productive Use of Failure in Inductive Proof. In *Automated Mathematical Induction*, Hantao Zhang (Ed.). Springer Netherlands, Dordrecht, 79–111. https://doi.org/10.1007/978-94-009-1675-3_3

Moa Johansson, Lucas Dixon, and Alan Bundy. 2010. Case-Analysis for Rippling and Inductive Proof. In *Proceedings of the First International Conference on Interactive Theorem Proving* (Edinburgh, UK) *(ITP'10)*. Springer-Verlag, Berlin, Heidelberg, 291–306. https://doi.org/10.1007/978-3-642-14052-5_21

Deepak Kapur, Rupak Majumdar, and Calogero G. Zarba. 2006. Interpolation for Data Structures. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) *(SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/1181775.1181789

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, USA.

Laura Kovács, Simon Robillard, and Andrei Voronkov. 2017. Coming to Terms with Quantified Reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. ACM, New York, NY, USA, 260–270. https://doi.org/10.1145/3009837.3009887

Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–35.

Sava Krstic, Amit Goel, Jim Grundy, and Cesare Tinelli. 2007. Combined Satisfiability modulo Parametric Theories. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal) *(TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 602–617.

K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. https://doi.org/10.5555/1939141.1939161

K. R. M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 361–381.

Rustan Leino and Nadia Polikarpova. 2013. Verified Calculations. https://www.microsoft.com/en-us/research/publication/verified-calculations/

Christof Löding, P. Madhusudan, and Lucas Peña. 2018. Foundations for natural proofs and quantifier instantiation. *PACMPL* 2, POPL (2018), 10:1–10:30. https://doi.org/10.1145/3158098

A. I. Mal'tsev. 1962. Axiomatizable classes of locally free algebras of certain types. *Sibirsk. Mat. Zh.* 3 (1962), 729–743. Issue 5.

Zohar Manna, Henny B. Sipma, and Ting Zhang. 2007. Verifying Balanced Trees. In *Logical Foundations of Computer Science*, Sergei N. Artemov and Anil Nerode (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 363–378.

Yuri V. Matiyasevich. 1993. *Hilbert's Tenth Problem.* MIT Press, Cambridge, MA, USA.

Michał Moskal. 2009. Programming with Triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. Association for Computing Machinery, New York, NY, USA, 20–29. https://doi.org/10.1145/1670412.1670416

Adithya Murali, Lucas Peña, Eion Blanchard, Christof Löding, and P. Madhusudan. 2022. Model-Guided Synthesis of Inductive Lemmas for FOL with Least Fixpoints. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 191 (oct 2022), 30 pages. https://doi.org/10.1145/3563354

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA. AAI8011683.

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (oct 1979), 245–257. https://doi.org/10.1145/357073.357079

Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. 2020. The Imandra Automated Reasoning System (System Description). In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 464–471.

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

Tuan-Hung Pham and Michael W. Whalen. 2013. RADA: A Tool for Reasoning about Algebraic Data Types with Abstractions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 611–614. https://doi.org/10.1145/2491411.2494597

Mojżesz Presburger and Dale Jabcquette. 1991. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic* 12, 2 (1991), 225–233. https://doi.org/10.1080/014453409108837187 arXiv:https://doi.org/10.1080/014453409108837187

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and P. Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/2491956.2462169

Giles Reger, Martin Suda, and Andrei Voronkov. 2017. Instantiation and pretending to be an SMT solver with VAMPIRE. *CEUR Workshop Proceedings* 1889 (1 Jan. 2017), 63–75. 15th International Workshop on Satisfiability Modulo Theories, SMT 2017 ; Conference date: 22-07-2017 Through 23-07-2017.

Andrew Reynolds. 2017. Conflicts, Models and Heuristics for Quantifier Instantiation in SMT. In *Vampire 2016. Proceedings of the 3rd Vampire Workshop (EPiC Series in Computing, Vol. 44)*, Laura Kovacs and Andrei Voronkov (Eds.). EasyChair, Portugal, 1–15. https://doi.org/10.29007/jmd3

Andrew Reynolds and Jasmin Christian Blanchette. 2017. A Decision Procedure for (Co)Datatypes in SMT Solvers. *J. Autom. Reason.* 58, 3 (mar 2017), 341–362. https://doi.org/10.1007/s10817-016-9372-6

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98.

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New

York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. *SIGPLAN Not.* 43, 6 (jun 2008), 159–169. https://doi.org/10.1145/1379022.1375602

Philipp Rümmer. 2012. E-Matching with Free Variables. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Nikolaj Bjørner and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–374.

T. Rybina and A. Voronkov. 2001. A Decision Procedure for Term Algebras with Queues. *ACM Trans. Comput. Logic* 2, 2 (apr 2001), 155–181. https://doi.org/10.1145/371316.371494

Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. 2022. Data-Driven Lemma Synthesis for Interactive Proofs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 143 (oct 2022), 27 pages. https://doi.org/10.1145/3563306

Thoralf Albert Skolem. 1934. Über die Nicht-charakterisierbarkeit der Zahlenreihe mittels endlich oder abzählbar unendlich vieler Aussagen mit ausschliesslich Zahlenvariablen. *Fundamenta Mathematicae* 23 (1934), 150–161.

Viorica Sofronie-Stokkermans. 2009. Locality Results for Certain Extensions of Theories with Bridging Functions. In *Automated Deduction – CADE-22*, Renate A. Schmidt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 67–83.

Philippe Suter, Mirco Dotta, and Viktor Kunćak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1706299.1706325

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 298–315.

Cesare Tinelli and Mehdi Harandi. 1996. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In *Frontiers of Combining Systems: First International Workshop, Munich, March 1996*, Frans Baader and Klaus U. Schulz (Eds.). Springer Netherlands, Dordrecht, 103–119. https://doi.org/10.1007/978-94-009-0349-4_5

Cesare Tinelli and Calogero G. Zarba. 2004. Combining Decision Procedures for Sorted Theories. In *Logics in Artificial Intelligence*, Jóse Júlio Alferes and João Leite (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 641–653.

Cesare Tinelli and Calogero G. Zarba. 2005. Combining Nonstably Infinite Theories. *Journal of Automated Reasoning* 34, 3 (01 Apr 2005), 209–238. https://doi.org/10.1007/s10817-005-5204-9

Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating Induction for Solving Horn Clauses. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 571–591.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.* 2, POPL (2018), 53:1–53:31. https://doi.org/10.1145/3158141

Philip Wadler. 2015. Propositions as Types. *Commun. ACM* 58, 12 (nov 2015), 75–84. https://doi.org/10.1145/2699407

Thomas Wies, Ruzica Piskac, and Viktor Kuncak. 2009. Combining Theories with Shared Set Operations. In *Frontiers of Combining Systems*, Silvio Ghilardi and Roberto Sebastiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 366–382.

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA.

Lawrence Wos, George A. Robinson, and Daniel F. Carson. 1965. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. *J. ACM* 12, 4 (oct 1965), 536–541. https://doi.org/10.1145/321296.321302

Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma Synthesis for Automating Induction over Algebraic Data Types. In *Principles and Practice of Constraint Programming*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 600–617.

Ting Zhang, Henny B. Sipma, and Zohar Manna. 2006. Decision procedures for term algebras with integer constraints. *Information and Computation* 204, 10 (2006), 1526–1574. https://doi.org/10.1016/j.ic.2006.03.004 Combining Logical Systems.

A APPENDIX

A.1 Continued from Section 4

Provable Acyclicity vs. Termination on the Standard Model.

Termination refers only to termination on the standard model. In contrast, provable acyclicity requires that arguments do not repeat when unfolding a definition (on any model), which is established using an order predicate and ranking functions. However, neither one implies the other.

For example, the function $f(x) = f(cons(0, x))$ is a provably acyclic function since the arguments to the function will never repeat across successive recursive calls. However, this function is not terminating on the standard model as it would just keep calling itself on larger and larger lists.

In contrast, the following predicate g is terminating on the standard model but is not provably acyclic:

$$\forall x.\, std(x) = \text{ite}(x = Nil, True, std(tail(x)))$$
$$\forall x.\, g(x) = \text{ite}(std(x), True, g(x))$$

std always terminates on the standard model returning $True$: it continually destructs the element and recursively calls itself until it reaches *Nil*, at which point it returns *True*. Therefore, $g(x)$ also terminates for elements in the standard model as one would simply evaluate the outermost condition (which terminates), and then take the branch corresponding to the success of the condition (which is just *True*).

However, g is not provably acyclic: it recursively calls $g(x)$ which does not decrease the argument. We can't use the fact we used in the termination argument that the *else* branch will never be taken because for provability we have to consider all models, not just the standard model. There are models where *std* does not always evaluate to *True*.

In fact, we use the fact that *std* cannot be proved to always be *True* to show incompleteness of UQFR when provably acyclic functions are replaced by terminating functions (see Theorem 8.2).

A.2 Expanded Arguments from Proof of Theorem 5.4

Formal Statement of Repair for Arbitrarily Large Sets

We first show some easy results. For a sort σ, consider the set U_σ consisting of all terms of type σ. Then, the set *DApp* of all possible $\mathcal{D}$-applications is:

$$DApp = \{(D, (t_1, t_2, \ldots t_r)) \mid D \in \mathcal{D},\ \text{D has signature } \sigma_1 \times \sigma_2 \ldots \sigma_r \to \sigma,\ u_i \in U_{\sigma_i} \text{ for } 1 \leq i \leq r\}$$

Note that any $\mathcal{D}$-application $(D, \bar{t})$ must belong to *DApp*, and in particular any computationally closed set C, which is a set of $\mathcal{D}$-applications, must be a subset of *DApp*.

Finally, if $\mathcal{N}$ is a Herbrand model of $\mathcal{T}_{comb}$, then its universe for a sort σ is precisely U_σ. Therefore, satisfying definitions *everywhere* on $\mathcal{N}$ simply amounts to satisfying definitions on *DApp*. The following proposition captures this idea:

Proposition A.1 (Definitions on a Herbrand Model). *Let $\mathcal{N}$ be a Herbrand model of $\mathcal{T}_{comb}$. Then, $\mathcal{N} \models DEF$ if and only if $\mathcal{N} \models DEF[DApp]$*

We now show the correctness of repairing arbitrarily large sets of $\mathcal{D}$-applications outside a computational closure.

Lemma A.2 (Definition Completion Lemma). *Let DEF be a set of definitions with only one stratum. Let C be a set that is computationally closed with respect to a set of quantifier-free formulas Γ. If $DEF[C] \wedge \Gamma$ is $\mathcal{T}_{comb}$-satisfiable, then $DEF \wedge \Gamma$ is $\mathcal{T}_{comb}$-satisfiable.*

The proof is the same as the one given in the main text, but we repeat it here.

Proof. We claim that $DEF[DApp] \cup \Gamma$ is $\mathcal{T}_{comb}$-satisfiable. Since $C \subseteq DApp$, let us rewrite this as $DEF[C] \cup DEF[\overline{C}] \cup \Gamma$, where $\overline{C} = DApp \setminus C$ is the complement of C.

We have from the statement of the theorem that $DEF[C] \cup \Gamma$ is satisfiable. Therefore, to show satisfiability of our desired set by compactness, it is sufficient to show that $DEF[C] \cup \Gamma \cup B$ is satisfiable for an arbitrary finite subset B of $DEF[\overline{C}]$.

Observe that a finite subset of $DEF[\overline{C}]$ is of the form $DEF[K]$ for a finite set $K \in DApp$. We are now done, since we know that $DEF[C] \cup \Gamma \cup DEF[K]$ is satisfiable from Lemma 5.8.

Finally, consider a model $\mathcal{N}$ such that $\mathcal{N} \models DEF[DApp] \cup \Gamma$. Without loss of generality, we can assume that $\mathcal{N}$ is a Herbrand model. Applying Proposition A.1 gives us that $\mathcal{N} \models DEF \cup \Gamma$, which concludes the proof. □

Generalizing Model Repair to Stratified Definitions

In proof of completeness in the main text we assumed that *DEF* had only one stratum. For the case of multiple strata, we first begin with the model $\mathcal{M}$ of *Inf* given to us by Stage 1. We then induct on the stratum number i, with the inductive hypothesis being that definitions for functions from strata $< i$ are satisfied everywhere. This hypothesis is true for the base case of the lowest stratum $i = 0$ by Lemma 5.8.

Inductively, we assume the hypothesis and then repair the model on definitions in the current strata by applying Lemma A.2. The arguments for the correctness of repair in this case are identical, i.e., we show the correctness of finite repair and then apply compactness.

However, there is a subtlety involved in showing the correctness of finite repair. The arguments are the same as in the proof of Lemma 5.8, with one exception. When we consider the argument for decreasing ranks, we needed the conjunct $\left(\bigwedge_{strat(H) < strat(D)} def_H \right)$ to be valid. When there is only one stratum, this is trivial since the conjunct is empty. For a general stratum i in our induction proof, the formula demands that the definitions corresponding to defined functions from lower strata are satisfied everywhere. But this is precisely the induction hypothesis! This concludes the proof of correctness of repair for a stratified set of definitions.

A.2.1 Proof of Theorem 4.4.

The above tools also provide us with a proof of Theorem 4.4. Recall that we are given a set *DEF* of stratified definitions, and model $\mathcal{M}$. We must create another model $\mathcal{M}'$ whose interpretation but for $\mathcal{D}$ is identical to $\mathcal{M}$, and whose valuation of functions in $\mathcal{D}$ satisfies their definitions.

We simply apply the arguments of the definition completion lemma (Lemma A.2) to $\mathcal{M}$ for each stratum of the definitions starting with the lowest, setting the set C where definitions are already satisfied as well as the set of formulas Γ to be empty. We can show by induction that when we apply the lemma at stratum i, the resulting model will satisfy definitions at strata $\leq i$ everywhere. This concludes the proof of Theorem 4.4.

A.3 Continued from Section 6

A.3.1 Rogue Nonstandard Model for `plus` *(Section 6.2).*

See Figure 4. Observe that $\mathcal{I}(plus)(i', Z) = (i + 1)' \neq i'$, where $\mathcal{I}()$ is the interpretation of the model.

A.3.2 Additional details for proof of `comm` *(Section 6.3).*

Rogue Nonstandard Model for `comm` ***Attempt 1.*** The following is a rogue nonstandard model for the `Peano` numbers over the same ADT universe as the model in Figure 4 and a different

Constructor			Destructor			Plus		
$\mathcal{I}(S)(i)$	=	$i+1$	$\mathcal{I}(pred)(i)$	=	$n-1$ if $0 < n$	$\mathcal{I}(plus)(i,j)$	=	$i+j$
$\mathcal{I}(S)(i')$	=	$(i+1)'$	$\mathcal{I}(pred)(i')$	=	$(i-1)'$	$\mathcal{I}(plus)(i',j')$	=	$(i+j)'$
$\mathcal{I}(Z)$	=	0				$\mathcal{I}(plus)(i',j)$	=	$(i+j+1)'$
						$\mathcal{I}(plus)(i,j')$	=	$(i+j)'$

Fig. 4. Rogue Nonstandard Model for *Peano* over the universe $\mathcal{U} \equiv \{0,1,2,\ldots\} \cup \{\ldots,-2',-1',0',1',2',\ldots\}$. comprising the naturals and a *primed* version of each integer. The model provides an interpretation for various constructors, destructors and *plus* that respects the ADT axioms, but where $\mathcal{I}(plus)(i',Z) = (i+1)' \neq i'$, refuting (3) , and $\mathcal{I}(plus)(i',j) \neq \mathcal{I}(plus)(j,i')$, refuting (2).

interpretation of `plus` that refutes the second conjunct of (4).

$$\begin{array}{lcllcll} \mathcal{I}(plus)(i,j) & = & i+j & \mathcal{I}(plus)(i',j') & = & (i+j+1)' & \text{if } 0 \leq j \\ \mathcal{I}(plus)(i,j') & = & (i+j)' & \mathcal{I}(plus)(i',j') & = & (i+j-1)' & \text{otherwise} \\ \mathcal{I}(plus)(i',j) & = & (i+j)' & & & & \end{array}$$

The reader should take a moment to check that the above definition respects def_{plus}. However, even though the induction hypothesis trivially holds at $\mathcal{I}(plus)(-1',-1')$ (as the arguments are the same), $\mathcal{I}(plus)(0',-1') = -2'$ which differs from $\mathcal{I}(plus)(-1',0') = 0'$!

Proof of `comm` ***Attempt 2.*** The peculiar property of the rogue nonstandard model from attempt 1 is that it introduces a *discontinuity* at $0'$ that violates $\forall n, m : Peano.\ plus(n, S(m)) = S(plus(n, m))$. We separately specify and inductively prove this property via a lemma `succR`:

```
succR :: n:Peano → m:Peano → { plus n (S m) = S (plus n m) }
```

The proof of `succR` is similar to `zeroR`:

```
succR :: n:Peano → m:Peano → { plus n (S m) = S (plus n m) }
succR Z     _ = ()
succR (S n) m = succR n m
```

Now, we can use both helper lemmas `zeroR` and `succR` to eliminate the rogue nonstandard models that thwarted our previous attempts, by the proof shown on the right in Figure 2. First, we replace the body of `comm Z m` with `comm z m = zeroR m` which add the post-condition of `zeroR` as a lemma. Second, we strengthen the body of `comm (S n) m` with a call `succR n m`. Together, these lemmas yield a VC with the strengthened antecedents that preclude the above rogue nonstandard models

$$\begin{array}{l} def_{plus} \rightarrow (\forall n, m.\ (plus(m,Z) = m \rightarrow plus(Z,m) = plus(m,Z)) \\ \qquad \wedge\ (plus(n,m) = plus(m,n) \rightarrow plus(m,S(n)) = S(plus(m,n)) \rightarrow \\ \qquad\quad plus(S(n),m) = plus(m,S(n)))) \end{array}$$

This time, PLE instantiates def_{plus} at (Z,m) and $(S(n),m)$ to yield an instantiated VC that the SMT solver validates even when *plus* is uninterpreted. Note that as with induction, LH makes no attempt to automate the creation and use of such lemmas: the programmer must explicitly spell them out by defining and proving them, and then "calling" the lemmas inside the theorem body to appropriately "instantiate" them at the relevant values, thereby yielding a VC that can be automatically discharged by *FLUID* reasoning.

A.3.3 Proofs of Search Tree Properties (Section 6.4).
Let us recall the property we want to prove:

$$\forall m, k, v.\ get(set(m, k, v), k) = \mathit{Just}(v)$$

Attempt 1: Direct Proof. In LH, we could try to specify and verify the above law as

```
getEq :: m:_ → k:_ → v:_ → { get (set m k v) k = Just v }
getEq m k v = ()
```

which would yield the VC $(def_{get} \wedge def_{set}) \rightarrow \forall m, k, v.\ get(set(m, k, v), k) = \mathit{Just}(v)$ Unfortunately, no (finite) instantiation can prove the above VC, as it is refuted by the (non-trivial) rogue nonstandard model detailed in Section 6.4.

Attempt 2: Inductive Proof. Instead, we need an inductive proof as in Figure 2.

```
getEq (Node key _ l r) k v
  | k == key   = ()
  | k <  key   = getEq l k v
  | otherwise  = getEq r k v
getEq Leaf _ _ = ()
```

The successful proof splits cases on constructor for `m` and then on the ordering of the two keys, and recursively invokes `getEq` (i.e. applies the induction hypothesis) on the left and right subtrees appropriately. The induction hypotheses in the antecedents of the VC for the above, as in `zeroR` eliminates the rogue nonstandard models, and allows PLE to find suitable instantiations such that the resulting instantiated VC can be validated by the SMT solver.

A.3.4 Example of Theorem Not Provable by PLE [Vazou et al. 2018]. Consider the following code:

```
g :: Int → Int
g x = 1

h :: Int → Int
h x = 2

f :: Int → Int
f x
  | x > 0    = g x
  | otherwise  = h x
```

Let us consider the property $\forall x.\ f(x) > 0$. The PLE heuristic implemented in LH cannot prove this property without the user explicitly providing a hint by splitting cases, but it is easy to see that UQFR will succeed as unfolding definitions on x yields $(f(x) = \text{ite}(x > 0, g(x), h(x)) \wedge g(x) = 1 \wedge h(x) = 2) \rightarrow (f(x) > 0)$, which is FO-valid.

A.4 Reducing Leon-style Reasoning to UQFR (Section 7)

We now show that the procedure Leon uses can in fact be captured in our framework using a formulation that encodes the required assumption of contracts. More precisely, we show that we can automatically derive formula of the form $DEF' \rightarrow \varphi'$ from the original formula $DEF \rightarrow \varphi$ that captures the required contract assumptions. Furthermore, when we use our algorithm (Algorithm UQFR in Figure 1) for this sentence, the procedure will mimic the one by Leon, assuming the contracts on unfolded function definitions.

We assume for simplicity an ADT universe with two destructors d_1 and d_2 that map to the ADT sort and other destructors that map to background sorts, and a single nullary constructor *Nil*. We also assume that there is only one recursively defined function f that takes arguments $(x, \overline{y})$ where x is of an ADT sort. Let the ADT sort have a single nullary constructor *Nil*. Without loss of generality, let us also suppose that the definition of f on arguments $(x, \overline{y})$ has two recursive calls with the arguments being $(d_1(x), \overline{t_1}(x, \overline{y}))$ and $(d_2(x), \overline{t_2}(x, \overline{y}))$ where $\overline{t_1}$ and $\overline{t_2}$ are tuples of terms over $x, \overline{y}$. This definition is in the *FLUID* fragment, i.e., provably acyclic as the first parameter decreases in the subterm ordering on the ADT sort. Finally, let us assume a postcondition $post_f(x, \overline{y}, \rho)$ for f, where x of type ADT and $\overline{y}$ are the input parameters for f and ρ is a variable denoting the return value of f on $(x, \overline{y})$. We assume that there is no precondition for ease of exposition.

Let φ be a property that we want to prove. The VC is then $DEF \rightarrow \varphi$ where *DEF* contains the definition of f. Without loss of generality, let f occur in φ as the term $f(z, \overline{w})$. In order to model Leon's procedure, when proving φ valid we need to be able to assume that $post_f(x, \overline{y}, f(x, \overline{y}))$ holds on arguments obtained by unfolding the definition of f on $(z, \overline{w})$ arbitrarily many times.

Let us define a new recursive predicate $Contract(x, \overline{y})$ with input parameters identical to f and the following definition:

$$\forall x.\ Contract(x, \overline{y}) = \big(post_f(x, \overline{y}, f(x, \overline{y})) \wedge \big(x \neq Nil \rightarrow (Contract(d_1(x), \overline{t_1}(x, \overline{y})) \wedge Contract(d_2(x)), \overline{t_2}(x, \overline{y}))\big)\big)$$

The above declares $Contract(x, \overline{y})$ to be true iff f satisfies its contract on the input $x, \overline{y}$ and further, if x is not *Nil*, *Contract* also holds for the recursive calls $(d_1(x), \overline{t_1}(x, \overline{y}))$ and $(d_2(x), \overline{t_2}(x, \overline{y}))$ that occur in the definition of f. Therefore, asserting $Contract(x, \overline{y})$ can be seen as asserting that the contract of f holds for $(x, \overline{y})$ as well as all the tuples that occur when unfolding the definition of f on $(x, \overline{y})$ *ad infinitum*, i.e., the computational closure of $(x, \overline{y})$ (see Section 5.2).

Finally, in order to simulate Leon-style reasoning we want to assert contracts for all tuples occurring in the unfolding of $f(z, \overline{w})$ (but not the contract for the arguments themselves). We do this by explicitly asserting *Contract* for the 'first level' of terms in the unfolding of f and adding *Contract* to the *DEF*:

$$DEF' \rightarrow \big(\big(z \neq Nil \rightarrow Contract(d_1(z), \overline{t_1}(z, \overline{w})) \wedge Contract(d_2(z), \overline{t_2}(z, \overline{w}))\big) \rightarrow \varphi\big)$$

where DEF' is the union of *DEF* and the definition for *Contract* as above. As argued earlier, given the definition of *Contract*, asserting *Contract* on the first level tuples amounts to asserting the contract for all tuples that occur in the unfolding of $f(z, \overline{w})$.

Observe that the new VC is also in the *FLUID* fragment. Furthermore, when we unfold definitions, unfolding the definition of *Contract* naturally leads to assuming (in the instantiated quantifier-free formula) that $post_f$ holds on the arguments that occur when unfolding the definition of f on $(z, \overline{w})$. Hence applying UQFR to the above constructed formula essentially simulates the procedure that Leon performs. It is easy to see that the above construction can be generalized to multiple ADT sorts with different signatures as well as multiple mutually recursively defined functions with their respective contracts.

A.5 Continued from Section 8

A.5.1 Proof of Theorem 8.1. We provide a reduction from the non-halting problem for two-counter machines. A two-counter machine [Hopcroft et al. 2006] is a machine with two registers that can contain unbounded integers. The machine can only increment or decrement these counters, or check whether they are equal to zero. Two-counter machines are computationally equivalent to Turing machines [Hopcroft et al. 2006], and checking the halting/non-halting of a two-counter machine is

undecidable (assuming, without loss of generality, an initial configuration where counters are set to zero).

It is tempting to try to find a simple reduction that encodes executions of the machine using ADTs (say, as lists of configurations), defining a recursive predicate that identifies *halting* executions (which are finite), and stating the theorem that no ADT element encodes a halting execution of the machine. However, note that we are seeking validity with respect to the *combined theory* and not validity in the standard model. In fact, since validity over the combined theory is recursively enumerable, we cannot reduce non-halting problem of two counter machines (which is co-r.e. hard) to it. Our reduction reduces the non-halting problem to the complement of validity, i.e., satisfiability. We provide the proof below.

Proof. Let us fix a two-counter machine M. Let us consider ADTs that are lists of triples of integers: ADT List, with two constructors Nil and Cons:

```
data List = Nil | Cons (state : Int) (fst : Int) (snd : Int) (tail :
    List)
```

Each element of the list represents a configuration of a two-counter machine– the state of the two-counter machine and the value of the two counters. We can write quantifier-free logical formulae $\mathsf{init}(x)$ representing the initial configuration, $\mathsf{halt}(x)$ representing any halting configuration, and $\mathsf{nextconfig}(x, y)$ representing that y is the successor configuration of x. Now consider the following recursive definition def_{nonhalt}:

$$\begin{aligned}\forall x : \mathsf{List}.\, \mathsf{nonhalt}(x) = \mathsf{ite}(x = \mathit{Nil}, \mathit{False},\\ \mathsf{ite}(\mathsf{halt}(x), \mathit{False}\\ \mathsf{nextconfig}(x, \mathit{tail}(x)) \wedge \mathsf{nonhalt}(\mathit{tail}(x))\end{aligned}$$

Note that this function recurses on $\mathit{tail}(x)$, which is a strict syntactic subterm of x. Thus its definition meets the requirement of the *FLUID* fragment.

Consider the following property to prove valid under the above definition:

$$\varphi \equiv \forall x.(\mathsf{init}(x) \rightarrow \neg\, \mathsf{nonhalt}(x))$$

We claim $def_{\mathsf{nonhalt}} \rightarrow \varphi$ is valid in the combined theory if and only if the two-counter machine *halts*.

If the formula is not valid, then there is a model and an ADT element x such that $\mathsf{init}(x)$ and $\mathsf{nonhalt}(x)$ hold. Then there are two cases: x corresponds to a finite list (reaching Nil in finitely many destructions) or it is a nonstandard element corresponding to an infinite list. The former case is impossible, as no finite list can have nonhalt to be true on it. In the second case, the recursive definition of nonhalt ensures that the list pointed to by x encodes an execution of the two-counter machine, and hence the machine does not halt.

Conversely, assume the machine does not halt. It turns out that we can build a nonstandard model where x points to a nonstandard ADT element encoding an infinite list that corresponds to the non-halting execution of the machine. Formally, we use the compactness theorem instead of constructing this model explicitly. Note that for any $k \in \mathbb{N}$, there is a *standard* ADT element that encodes a finite list corresponding to a partial execution of the two-counter machine for k steps. Hence any *unfolding* of the definition of nonhalt on x, $\mathit{tail}(x)$, etc. up to k destructions is satisfiable. By the compactness theorem, the unfolding for ω number of steps is also satisfiable. One can now see that $\mathsf{nonhalt}(x)$ will hold in this model. □

A.5.2 *Proof of Theorem 8.2.*

Proof. We construct an instance of the validity problem without provably acyclic definitions that is unprovable for UQFR. We use the familiar ADT of lists over integers as our foreground sort:

```
data List = Nil | Cons (head : Int) (tail : List)
```

as well as the following definitions:

$$def_{std} \equiv \forall x : \mathsf{List}.\, std(x) = \mathsf{ite}(x = Nil, True, std(tail(x)))$$
$$def_{R} \equiv \forall x : \mathsf{List}.\, R(x) = \mathsf{ite}(std(x), True, \mathsf{ite}(head(x) = 0, R(x), \neg R(x)))$$

Both functions are well-defined definitions as they terminate on the standard model. The termination of *std* is apparent: it simply destructs the input term recursively until *Nil* and then returns *True*. *R* is also terminating on the standard model since $std(x)$ is always true on standard elements, therefore the *else* branch of the outer ite is never taken.

std is also provably acyclic since the arguments to the recursive call are smaller according to the subterm ordering. However, *R* is not provably acyclic. We demonstrate this indirectly by proving the incompleteness of UQFR and defer the discussion of why it is not provably acyclic.

Consider the theorem $\varphi \equiv \forall y : \mathsf{List}.\, R(y)$. We claim: (1) $\mathcal{T}_{comb} \models DEF \rightarrow \varphi$, and (2) UQFR does not terminate on (DEF, φ). We do not prove the latter here as it can be deduced easily by following the algorithm in Figure 1 and only show the former.

Suppose the claim is not true. Note that the antecedent is not vacuous since it is possible to satisfy the definitions on the standard model. Therefore, for the claim to be false there must exist a model where the definitions are satisfied, but there is a y such that $R(y)$ does not hold. From the definition of *R*, we know that this is only possible when $\neg std(x)$ and $head(x) = 0$. Any other path in the definition either leads to $R(x)$ being true or the impossibility $R(x) = \neg R(x)$. Now, consider the element $Cons(1, y)$. From the definition of *std*, we have $std(Cons(1, y)) = std(y) = False$. Then, following the definition of *R* yields $R(Cons(1, y)) = \neg R(Cons(1, y))$, which is impossible. Therefore, it must be the case that there is no model where the definitions are satisfied but φ does not hold. In other words, $\mathcal{T}_{comb} \models (DEF, \varphi)$.

UQFR never terminates on the algorithm because unfolding the definitions only ever produce terms that are destructions of y, whereas we proved the validity above by instantiating the definitions on a superterm of y. This shows that UQFR is incomplete for this instance. □

Discussion. Since we prove completeness for provably acyclic definitions, the above shows that *R* is not provably acyclic. More specifically, this is because in order to prove that the absurd recursive call $\neg R(x)$ is unreachable, we must essentially prove that $std(x)$ always holds. However, this not true in the combined theory, and one would typically use induction to establish this.

Consequently, in models where $std(y)$ does not hold for some y, *R* is in fact unrealizable as the element $Cons(1, y)$ cannot be given a valuation that is consistent with the definition of *R*. This does not happen with provably acyclic definitions because such functions can always be given a valuation consistent with their definitions on any model (see Theorem 4.4).

The completeness of UQFR arises from the fact that unfolding the (provably acyclic) definition of some R ($R \in \mathcal{D}$) on x amounts to a simulation of the "computation" of $R(x)$ in any model. Without provable acyclicity, we have shown that it is possible to construct well-defined definitions that are unrealizable in some models, rendering UQFR incomplete.

Received 2023-04-14; accepted 2023-08-27