System FR: Formalized Foundations for the Stainless Verifier

JAD HAMZA, EPFL, Switzerland
NICOLAS VOIROL, EPFL, Switzerland
VIKTOR KUNČAK, EPFL, Switzerland

We present the design, implementation, and foundation of a verifier for higher-order functional programs with generics and recursive data types. Our system supports proving safety and termination using preconditions, postconditions and assertions. It supports writing proof hints using assertions and recursive calls. To formalize the soundness of the system we introduce System FR, a calculus supporting System F polymorphism, dependent refinement types, and recursive types (including recursion through contravariant positions of function types). Through the use of sized types, System FR supports reasoning about termination of lazy data structures such as streams. We formalize a reducibility argument using the Coq proof assistant and prove the soundness of a type-checker with respect to call-by-value semantics, ensuring type safety and normalization for typeable programs. Our program verifier is implemented as an alternative verification-condition generator for the Stainless tool, which relies on the Inox SMT-based solver backend for automation. We demonstrate the efficiency of our approach by verifying a collection of higher-order functional programs comprising around 14000 lines of polymorphic higher-order Scala code, including graph search algorithms, basic number theory, monad laws, functional data structures, and assignments from popular Functional Programming MOOCs.

CCS Concepts: • Theory of computation → Logic and verification; Programming logic; Type theory; • Software and its engineering → Software verification; • Computing methodologies → Theorem proving algorithms.

Additional Key Words and Phrases: software verification, dependent types, recursive types, System F, SMT

ACM Reference Format:

1 INTRODUCTION

Automatically verifying the correctness of higher-order programs is a long-standing problem that arises in most programming languages and proof assistants. Despite extensive research in program verifiers and proof assistants [Abel 2010; Bertot and Castéran 2004a; Brady 2013; Gordon and Melham 1993; Harrison 2009, 2017; Leino 2010; Nipkow et al. 2002a; Norell 2007; Swamy et al. 2013; Vazou et al. 2014] there remain significant challenges and trade-offs in checking safety and termination. A motivation for our work are implementations that verify polymorphic functional programs using SMT solvers [Suter et al. 2011; Vazou et al. 2014]. To focus on foundations, we look at simpler verifiers that do not perform invariant inference and are mostly based on unfolding recursive definitions and encoding of higher-order functions into SMT theories [Blanc and Kunčak 2015; Suter et al. 2011; Voirol et al. 2015]. A recent implementation of such a verifier is the Stainless
system [LARA 2019], which handles a subset of Scala [Odersky et al. 2008]. The goal of Stainless is to verify that function contracts hold and that all functions terminate. It was shown [Hupel and Kuncak 2016] how to map certain patterns of specified Scala programs into Isabelle/HOL. Whereas this approach ensures soundness, it does not reuse the reasoning of Stainless and it can verify only some of the programs that Stainless verifier can prove. The present paper seeks to provide direct foundations for verification and termination checking of functional programs with a rich set of features for purely functional programming including non-monotonic data types. On the other hand, our calculus does not aspire to directly support effects for which there exist excellent other systems [Swamy et al. 2013].

The subtleties of ensuring function termination have been an initial impetus for the calculus we present. Termination is desirable for many executable functions in programs and is even more important in formal specifications. A non-terminating function definition such as \( f(x) = 1 + f(x) \) could be easily mapped to a contradiction and violate the conservative extension principle for definitions. Yet termination in the presence of higher-order functions and data types is challenging to ensure. For example, when using non-monotonic recursive types, terms can diverge even without the explicit use of recursive functions, as illustrated by the following snippet of Scala code:

```scala
case class D(f: D => Unit) // non-monotonic recursive type
def g(d: D): Unit = d.f(d) // non-recursive function definition
g(D(g)) // diverging term, reduces to D(g).f(D(g)) and then again to g(D(g))
```

Furthermore, even though the concept of termination for all function inputs is an intuitively clear property, its modular definition is subtle: a higher order function \( g \) taking another function \( f \) as an argument should terminate when given any terminating function \( f \), which, in turn, can be applied to expressions involving further calls to \( g \). The quest for solid foundations for termination led us to type theoretic techniques, where reducibility method has long been used to show strong normalization of expressive calculi [Tait 1967], [Girard 1990, Chapter 6], [Harper 2016]. As a natural framework for analyzing support for first-class functions with preconditions and post-conditions we embraced the ideas of refinement dependent types similar to those in Liquid Haskell [Vazou et al. 2014] with refinement-based notion of subtyping. To explain proof obligation generation in the higher-order case (including the question of which assumptions should be visible when checking a given assertion), we resorted to well-known dependent (II) function types. To support parametric polymorphism we incorporated type quantifiers, as in System F [Girard 1971, 1990]. We found that the presence of refinement types allowed us to explain soundness of well-founded recursion based on user-defined measures. The recursion in programs is thus not syntactically restricted as in, e.g., System F. To provide expressive support for iterative unfolding of recursive functions, we introduced rules to make function bodies available while type checking of recursive functions. For recursive type definitions, many existing systems introduce separate notions of inductive and co-inductive definitions. We found this distinction less natural for developers and chose to support expressive recursive types (without a necessary restriction to positive recursion) using sized types [Abel 2010]. We draw inspiration from a number of existing systems, yet our solution is a new sound combination of features that work nicely together.

We combined these features into a new type system, System FR, which we present as a bidirectional type checking algorithm. The algorithm generates type checking and type inference goals by traversing terms and types, until it reaches a point where it has to check that a given term evaluates to \texttt{true}. This typically arises when we want to check that a term \( t \) has a refinement type \( \{ x : T \mid b \} \), which is the case when \( t \) has type \( T \), and when the term \( b \) evaluates to \texttt{true} in the context where \( x \) equals \( t \). Following the tradition of SMT-based verifiers [Barnett et al. 2004; Detlefs et al. 1998], we use the term verification condition (VC) to refer to a term that should evaluate to \texttt{true}.
We prove the soundness of our type system using a reducibility interpretation of types. The goal of our verification system is to ensure that a given term belongs to the semantic denotation of a given type. For simple types such as natural numbers, this denotation is the set of untyped lambda calculus terms that evaluate, in a finite number of steps, to a non-negative integer. For function types the denotation are, as is typical in reducibility approaches, terms that, when applied to terms in denotation of argument type, evaluate to terms in the denotation of the result type. Such denotation gives us a unified framework for function contracts expressed as refinement types. The approach ensures termination of programs because the semantics of types only contain terms that are terminating in call-by-value semantics.

We have formally proven using the Coq proof assistant \cite{Bertot2004} the soundness of our typing algorithm, implying that when verification conditions generated for checking that a term $t$ belongs to a type $T$ are semantically valid, the term $t$ belongs to the semantic denotation of the type $T$. The bidirectional typing algorithm handles the expressive types in a deterministic and predictable way, which enables good and localized error reporting to the user. To solve generated verification conditions, we use existing implementation invoking the Inox solver\footnote{https://github.com/epfl-lara/inox} that reduces higher-order queries to the first-order language of SMT solvers \cite{Voirol2015}. Our semantics of types provides a definition of soundness for such solvers; any solver that respects the semantics can be used with our verification condition generator. Our bidirectional type checking algorithm thus becomes a new, trustworthy verification condition generator for Stainless. We were successful in verifying many existing Stainless benchmarks using the new approach.

We summarize our contributions as follows:

- We present a rich type system, called System FR, that combines System F with dependent types, refinements, equality types, and recursive types (Sections 3 and 4).
- We define a bidirectional type-checking algorithm for System FR (Section 5). Our algorithm generates verification conditions that are solved by the (existing) SMT-based solver Inox.
- We prove\footnote{https://github.com/epfl-lara/SystemFR/tree/oopsla2019} soundness of our bidirectional type-checking algorithm that reduces program correctness to proving that certain formulas always evaluate to true (Section 6). Our formalization also supports additional expressive notions, such as infinite intersections and unions as well as refinement conditions given by non-emptiness of an arbitrary type.
- We built a verification condition generator based on these foundations and evaluated it on around 14k lines of benchmarks (Section 7), showing that generating proof obligations using type checking is effective in practice.

```
def f(x: τ₁): τ₂ = {
  require(pre[x])
  decreases(m[x])
  E[x, f]
} ensuring { res ⇒ post[x, res] }
```

Fig. 1. Template of a recursive function with user-given contracts and a decreasing measure.

2 EXAMPLES OF PROGRAM VERIFICATION AND TERMINATION CHECKING

Our goal is to verify correctness and termination of pure Scala functions written as in Figure 1. $pre[x]$ is the precondition of the function $f$, and is written by the user in the same language as...
the body of $f$. The precondition may contain arbitrary expressions and calls to other functions. Similarly, the user specifies in $post$ the property that the results of the function should satisfy. To ensure termination of $f$ (which might call itself recursively), the user may also provide a measure using the $decreases$ keyword, which is also an expression (of type $Nat$, the type of natural numbers) written in the same language. $\tau_1$ and $\tau_2$ may be arbitrary types, including function types or algebraic data types. Informally, the function is terminating and correct, if, for every value $v$ of type $\tau_1$ such that $pre[v]$ evaluates to true, $f(v)$ returns (in a finite number of steps) a value $res$ of type $\tau_2$ such that $post[v, res]$ evaluates to true. By using dependent and refinement types, this can be summarized by saying that the function $f$ has type: $\Pi x : \{x : \tau_1 \mid pre[x]\}. \{res : \tau_2 \mid post[x, res]\}$.

As an example, consider the list type as defined in Figure 2. We use $Z$ to denote the type of integers (corresponding to Scala’s BigInt in actual source code). The function $filter$ filters elements from a list, while $count$ counts the number of occurrences of an integer in the list. These two functions have no pre- or postconditions. The $decreases$ clauses specify that the functions terminate because the size of the list decreases at each recursive call.

Using these functions we define $partition$ in Figure 3, which takes a list $l$ of integers and partitions it according to a predicate $p$: $Z \Rightarrow Bool$. We prove in the postcondition that partitioning coincides with applying $filter$ to the list with $p$ and its negation.

As an example, consider the list type as defined in Figure 2. We use $Z$ to denote the type of integers (corresponding to Scala’s BigInt in actual source code). The function $filter$ filters elements from a list, while $count$ counts the number of occurrences of an integer in the list. These two functions have no pre- or postconditions. The $decreases$ clauses specify that the functions terminate because the size of the list decreases at each recursive call.

Using these functions we define $partition$ in Figure 3, which takes a list $l$ of integers and partitions it according to a predicate $p$: $Z \Rightarrow Bool$. We prove in the postcondition that partitioning coincides with applying $filter$ to the list with $p$ and its negation.

As an example, consider the list type as defined in Figure 2. We use $Z$ to denote the type of integers (corresponding to Scala’s BigInt in actual source code). The function $filter$ filters elements from a list, while $count$ counts the number of occurrences of an integer in the list. These two functions have no pre- or postconditions. The $decreases$ clauses specify that the functions terminate because the size of the list decreases at each recursive call.

Using these functions we define $partition$ in Figure 3, which takes a list $l$ of integers and partitions it according to a predicate $p$: $Z \Rightarrow Bool$. We prove in the postcondition that partitioning coincides with applying $filter$ to the list with $p$ and its negation.

As an example, consider the list type as defined in Figure 2. We use $Z$ to denote the type of integers (corresponding to Scala’s BigInt in actual source code). The function $filter$ filters elements from a list, while $count$ counts the number of occurrences of an integer in the list. These two functions have no pre- or postconditions. The $decreases$ clauses specify that the functions terminate because the size of the list decreases at each recursive call.

Using these functions we define $partition$ in Figure 3, which takes a list $l$ of integers and partitions it according to a predicate $p$: $Z \Rightarrow Bool$. We prove in the postcondition that partitioning coincides with applying $filter$ to the list with $p$ and its negation.

As an example, consider the list type as defined in Figure 2. We use $Z$ to denote the type of integers (corresponding to Scala’s BigInt in actual source code). The function $filter$ filters elements from a list, while $count$ counts the number of occurrences of an integer in the list. These two functions have no pre- or postconditions. The $decreases$ clauses specify that the functions terminate because the size of the list decreases at each recursive call.

Using these functions we define $partition$ in Figure 3, which takes a list $l$ of integers and partitions it according to a predicate $p$: $Z \Rightarrow Bool$. We prove in the postcondition that partitioning coincides with applying $filter$ to the list with $p$ and its negation.

As an example, consider the list type as defined in Figure 2. We use $Z$ to denote the type of integers (corresponding to Scala’s BigInt in actual source code). The function $filter$ filters elements from a list, while $count$ counts the number of occurrences of an integer in the list. These two functions have no pre- or postconditions. The $decreases$ clauses specify that the functions terminate because the size of the list decreases at each recursive call.

Using these functions we define $partition$ in Figure 3, which takes a list $l$ of integers and partitions it according to a predicate $p$: $Z \Rightarrow Bool$. We prove in the postcondition that partitioning coincides with applying $filter$ to the list with $p$ and its negation.
def isSorted(l: List): Boolean = {
  decreases(size(l))
  l match {
    case Nil() ⇒ true
    case Cons(x, Nil()) ⇒ true
    case Cons(x, Cons(y, ys)) ⇒
      x ≤ y && isSorted(Cons(y, ys))
  }
}

def merge(l1: List, l2: List): List = {
  require(isSorted(l1) && isSorted(l2))
  decreases(size(l1) + size(l2))
  (l1, l2) match {
    case (Cons(x, xs), Cons(y, ys)) ⇒
      if (x ≤ y) Cons(x, merge(xs, l2))
      else Cons(y, merge(l1, ys))
    case (Cons(_, _), Nil) ⇒ l1
    case _ => l2
  }
}

ensuring { res => isSorted(res) }

Fig. 5. A function that checks whether a list is sorted and a function that merges two sorted lists

Figure 4 shows a theorem that partition also preserves the multiplicity of each element. We use here count to state the property, but we could have used multisets instead (a type which is natively supported in Stainless). The holds keyword is a shorthand for ensuring { res => res }. The @induct annotation instructs the system to add a recursive call to partitionMultiplicity on the tail of 1 when 1 is not empty. This gives us access to the multiplicity property for the tail of 1, which the system can then use automatically to prove that the property holds for 1 itself. This corresponds to a proof by induction on 1.

Figure 5 shows a function isSorted that checks whether a list is sorted, and a function merge that combines two sorted lists in a sorted list. When given the above input, the system proves the termination of all functions, establishes that postconditions of functions hold, and shows that the theorem holds, without any user interaction or additional annotations. For the merge function, the postcondition might seem too weak to establish that e.g. Cons(x, merge(xs, l2)) is sorted just based on the fact that merge(xs, l2) is sorted. However, since we put the definition of merge while type-checking the body of merge in the context, it is possible to establish that x is smaller than the head of merge(xs, l2). We give more details on this feature of our system, called body-visible recursion, in Section 5.3.

2.1 Reasoning about Streams

Our system also supports reasoning about infinite data structures, including streams that are computed on demand. These data structures are challenging to deal with because even defining termination of an infinite stream is non-obvious, especially in absence of a concrete operation that uses the stream. Given some type X, Stream[X] represents the type of infinite streams containing elements in X. In a mainstream call-by-value language such as Scala, this type can be defined as:

case class Stream[X](head: X, tail: () ⇒ Stream[X])

For the sake of concise syntax, we typeset a function taking unit, (u: Unit) => e, using Scala’s syntax () => e for a function of zero parameters. Given a stream s: Stream[X], we can call s.head to get the head of the stream (which is of type X), or s.tail to get the tail of the stream (which is of type () => Stream[X]). We can use recursion to define streams, as shown in figures 6, 8, 7. The @erasable annotation is used to mark the erasable parameters n of these functions. These parameters are used as annotations to guide our type-checker, but they do not influence the computation and can be erased at runtime. For instance, an erased version of constant (without erasable code and without type annotation) looks like:

def constant(x) = Stream(x, () => constant(x))
Informally, we can say that the constant stream is terminating. Indeed, it has the interesting property that, despite the recursion, for every \( n \in \mathbb{N} \), we can take the first \( n \) elements in finite time (no divergence in the computation). We say that constant\((x)\) is an \( n \)-non-diverging stream. Moreover, when a stream is \( n \)-non-diverging for every \( n \in \mathbb{N} \), we simply say that it is non-diverging, which means that we can take as many elements as we want without diverging, which is the case for constant\((x)\). Note that non-divergence of constant cannot be shown by defining a measure on its argument \( x \) that strictly decreases on each recursive call, because constant is called recursively on the exact same argument \( x \). Instead, we define a measure on the erasable argument \( n \) of the annotated version. This corresponds to using type-based termination [Abel 2007, 2008; Barthe et al. 2008], where the type of the function for the recursive call is smaller than the type of the caller. We expand on that technique in Section 5.2.

In the annotated version of constant from Figure 6, the notation Stream\([X]\)(n) stands for streams of elements in \( X \) which are \( n \)-non-diverging. The type of constant then states that constant can be called with any (erasable) parameter \( n \) to build an \( n \)-non-diverging stream. Since parameter \( n \) is computationally irrelevant, this proves that the erased version of constant returns a non-diverging stream.

We now give a variant of constant which is diverging:

```python
def badConstant(x) = Stream(x, (⇒ badConstant(x).tail())
```

Indeed, given some \( x \), a call to badConstant\((x)\).tail() again, and would diverge.

The zipWith function in Figure 7 takes two streams and a function \( f \). It creates a new stream by applying \( f \) to pairs of elements taken from each stream. For zipWith, we can verify that as long as \( f \) terminates on every input, and \( s1 \) and \( s2 \) are non-diverging streams, then zipWith returns a non-diverging stream. We can then use zipWith to define the well-known Fibonacci stream (Figure 8), an infinite stream containing the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, etc. We make use of a function plus:\( Z \Rightarrow Z \Rightarrow Z \) that computes the sum of two integers. Just like the constant
streams, \( \text{fib} \) is a non-diverging stream. For instance, calling \( \text{fib} \).\text{tail}().\text{tail}().\text{tail}().\text{head} \) returns (in finite time) the number 2 from the Fibonacci sequence.

The important property that the type signature of \( \text{zipWith} \) ensures is that, for every \( n \in \text{Nat} \), if \( s_1 \) and \( s_2 \) are \( n \)-non-diverging streams, then \( \text{zipWith} f s_1 s_2 \) is \( n \)-non-diverging as well. Our type system can check this property and then use it to make sure that the definition of \( \text{fib} \) type-checks.

We can also prove further properties of interest, e.g., that zipping two streams \( s_1 \) we keep it in the paper for presenting examples.

The set \( \text{Val} \) of values, a pattern matching operator \( \text{match} \) includes the unit term \( () \), pairs, booleans, natural numbers, a recursor \( \text{rec} \) for iterating over natural numbers, a pattern matching operator \( \text{match} \) for natural numbers, a recursion operator \( \text{fix} \), an error term \( \text{err} \) to represent crashes. The recursor \( \text{rec} \) can be simulated using \( \text{fix} \) and \( \text{match} \) but we keep it in the paper for presenting examples.

The terms \( \text{fold}(t) \) and \( \text{unfold} t \_ \in x \Rightarrow t \) are used to represent data structures (such as lists or streams), where \( \text{fold}(t) \) plays the role of a constructor, and \( \text{unfold} \_ \in x \) the role of a deconstructor. The terms \( \Lambda t \_ \in t[] \) are used to represent the erasure of type abstractions and type instantiation terms (for polymorphism) of the form \( \Lambda \alpha t \_ \in t[\tau] \), where \( \alpha \) is a type variable and \( \tau \) is a type. These annotated terms will be introduced in a further section.

The term \( \text{size}(t) \) is a special term to internalize the sizes of syntax trees of values (ignoring lambdas) of our language. It is used for measure of recursive functions such as the map examples on lists shown in Section 2.

Given a term \( t \) we denote \( \text{fv}(t) \) the set of all free variables of \( t \). Terms are considered up to renaming of locally bound variables (alpha-renaming).

3 SYNTAX AND OPERATIONAL SEMANTICS

We now give a formal syntax for terms and show call-by-value operational semantics. This untyped lambda calculus with pairs, tagged unions, integers, booleans, and error values models programs that our verification system supports. It is Turing complete and rather conventional.

3.1 Terms of an Untyped Calculus

Let \( \mathcal{V} \) be a set of variables. We let Terms be the set of all (untyped) terms (see Figure 9) which includes the unit term \( () \), pairs, booleans, natural numbers, a recursor \( \text{rec} \) for iterating over natural numbers, a pattern matching operator \( \text{match} \) for natural numbers, a recursion operator \( \text{fix} \), an error term \( \text{err} \) to represent crashes. The recursor \( \text{rec} \) can be simulated using \( \text{fix} \) and \( \text{match} \) but we keep it in the paper for presenting examples.

The terms \( \text{fold}(t) \) and \( \text{unfold} t \_ \in x \Rightarrow t \) are used to represent data structures (such as lists or streams), where \( \text{fold}(t) \) plays the role of a constructor, and \( \text{unfold} \_ \in x \) the role of a deconstructor. The terms \( \Lambda t \_ \in t[] \) are used to represent the erasure of type abstractions and type instantiation terms (for polymorphism) of the form \( \Lambda \alpha t \_ \in t[\tau] \), where \( \alpha \) is a type variable and \( \tau \) is a type. These annotated terms will be introduced in a further section.

The term \( \text{size}(t) \) is a special term to internalize the sizes of syntax trees of values (ignoring lambdas) of our language. It is used for measure of recursive functions such as the map examples on lists shown in Section 2.

Given a term \( t \) we denote \( \text{fv}(t) \) the set of all free variables of \( t \). Terms are considered up to renaming of locally bound variables (alpha-renaming).

3.2 Call-by-Value Operational Semantics

The set \( \text{Val} \) of values of our language is defined (inductively) to be \( \text{zero} \), \( () \), \( \text{true} \), \( \text{false} \), every variable \( x \), every lambda term \( \lambda x. t \) or \( \Lambda t \), the terms of the form \( \text{succ}(v) \) or \( \text{fold}(v) \) where \( v \in \text{Val} \), and the terms of the form \( (v_1, v_2) \) where \( v_1, v_2 \in \text{Val} \).

The call-by-value small-step relation between two terms \( t_1, t_2 \in \text{Terms} \), written \( t_1 \leftarrow t_2 \), is standard for the most part and given in Figure 10. Given a term \( t \) and a value \( v \), \( t[x \leftarrow v] \) denotes the term \( t \) where every free occurrence of \( x \) has been replaced by \( v \).
\[ \begin{align*}
\forall \pi_1, \pi_2 : v_1 : v_2 \rightarrow v_1 & \quad (\beta_1) \\
\forall \pi_2 : v_2 = v_2 & \quad (\beta_2) \\
\forall v : (\lambda x . t \leftrightarrow t[x \mapsto v]) & \quad (\beta_3) \\
(\Delta t)[] \rightarrow t & \quad (\beta_4) \\
\text{if true then } t_1 \text{ else } t_2 \rightarrow t_1 & \quad (\beta_5) \\
\text{if false then else } t_2 \rightarrow t_2 & \quad (\beta_6) \\
\text{rec (zero, } t_0, _) \rightarrow t_0 & \quad (\beta_7) \\
\text{rec (succ (v), } t_0, (n, y) \Rightarrow t_s) \rightarrow t_s[n \mapsto v, y \mapsto \lambda u . \text{rec (v, } t_0, (n, y) \Rightarrow t_s)] & \quad (\beta_8) \\
\text{fix (y \Rightarrow t)} \rightarrow t[y \mapsto \lambda u . \text{fix (y \Rightarrow t)}] & \quad (\beta_9) \\
\text{match (zero, } t_0, _) \rightarrow t_0 & \quad (\beta_{10}) \\
\text{match (succ (v), } _, n \Rightarrow t_s) \rightarrow t_s[n \mapsto v] & \quad (\beta_{11}) \\
\text{or (latch (v), } x \Rightarrow t_1) \rightarrow t[x \mapsto v] & \quad (\beta_{12}) \\
\text{or (right (v), } _, x \Rightarrow t) \rightarrow t[x \mapsto v] & \quad (\beta_{13}) \\
\text{let } x = v \text{ in } t \rightarrow t[x \mapsto v] & \quad (\beta_{14}) \\
\text{unfold fold (v) in } x \Rightarrow t \rightarrow t[x \mapsto v] & \quad (\beta_{15}) \\
\text{size (v) \rightarrow build Nat (size (v))} & \quad (\beta_{16}) \\
E[t] \rightarrow E'[t'] & \quad (\text{Evaluation Context})
\end{align*} \]

**Fig. 10.** Small-step call-by-value operational semantics of untyped terms

\[
\begin{align*}
\text{size \_sem (\text{succ} (v))} = 1 + \text{size \_sem (v)} & \quad \text{size \_sem (\text{fold} (v))} = 1 + \text{size \_sem (v)} \\
\text{size \_sem (right (v))} = 1 + \text{size \_sem (v)} & \quad \text{size \_sem (left (v))} = 1 + \text{size \_sem (v)} \\
\text{size \_sem (v_1, v_2)} = \text{size \_sem (v_1)} + \text{size \_sem (v_2)} & \quad \text{size \_sem (v)} = 0, \text{ in all other cases}
\end{align*} \]

**Fig. 11.** Definition of \text{size \_sem (v)} for a value \text{v}.

To evaluate the fixpoint operator \text{fix}, we use the rule \text{fix (y \Rightarrow t)} \rightarrow t[y \mapsto \lambda (). \text{fix (y \Rightarrow t)}], which substitutes the \text{fix} under a lambda with unit argument. We do this wrapping of \text{fix} in a lambda term because we wanted all substitutions to be values for our call-by-value semantics, and \text{fix} is not. This also means that, to make a recursive call within \text{t}, one has to use \text{y()} instead of \text{y}.

To define the semantics of \text{size (v)}, we use a (mathematical) function \text{size \_sem (v)} that returns the size of a value, ignoring lambdas for which it returns 0. The precise definition is given in Figure 11.

We make use in the operational semantics of an \textit{evaluation context} \text{E}, which specifies through a \text{hole} \text{H} the next place where reduction can occur in a term. Inductively, an evaluation context \text{E} must be of one of the following forms:

\[
\begin{align*}
\text{H} & \quad \text{E e} \quad v E \quad (E, e) \quad (v, E) \quad \pi_1 E \quad \pi_2 E \quad \text{succ (E)} \quad \\
\text{rec (E, _, _)} \quad \text{match (E, _, _)} \quad \text{if E then t else } t \quad \text{let } x = E \text{ in } t \quad \text{size (E)} \quad \\
\text{fold (E)} \quad \text{unfold E in _} \quad E[] \quad \text{left (E)} \quad \text{right (E)} \quad \text{or (match (E, _, _))}
\end{align*} \]

Given a term \text{t}, we denote by \text{E}[\text{t}] the context \text{E} where the \text{hole} \text{H} has been replaced by \text{t}.

We denote by \rightarrow^* the reflexive and transitive closure of \rightarrow. A term \text{t} is \textit{normalizing} if there exists a value \text{v} such that \text{t} \rightarrow^* \text{v}.
We use this type as a building block for writing specifications (pre and postconditions).

We give in Figure 12 the grammar for the types \( \tau \) when we can just denote this set by

\[
\begin{align*}
\tau & ::= \text{Unit} \mid \text{Bool} \mid \text{Nat} \mid \top \mid \Pi \tau_1 : \tau_2 \times \tau_1 \mid \Sigma \tau_1 : \tau_2 \mid \tau_1 + \tau_2 \mid \forall \tau_1 : \tau_2 \mid \forall \alpha : \text{Type.} \tau \\
\text{Rec(n)}(\alpha \Rightarrow \tau) & | \{ x : \tau \mid t \} | t \equiv t
\end{align*}
\]

Fig. 12. Grammar for types \( \tau \), where \( x \in \mathcal{V} \) is a term variable, \( \alpha \in \mathcal{V} \) is a type variable \((t \text{ denotes annotated terms of Figure 16 that complete the mutually recursive definition)}\)

## 4 Types, Semantics and Reducibility

We give in Figure 12 the grammar for the types \( \tau \) that our verification system supports. Given two types \( \tau_1 \) and \( \tau_2 \), we use the notation \( \tau_1 \rightarrow \tau_2 \) for \( \Pi \tau_1 : \tau_2 \) when \( x \) is not a free variable of \( \tau_2 \). Similarly, we use the notation \( \tau_1 \times \tau_2 \) for \( \Sigma \tau_1 : \tau_2 \) when \( x \) is not a free variable of \( \tau_2 \).

For recursive types, we introduce the notation:

\[
\text{Rec}(\alpha \Rightarrow \tau) \triangleq \forall n : \text{Nat. Rec}(n)(\alpha \Rightarrow \tau)
\]

Then, the type of (non-diverging) streams informally introduced in Section 2 can be understood as a notation, when \( X \) is a type, for: \( \text{Stream}[X] \triangleq \text{Rec}(\alpha \Rightarrow X \times (\text{Unit} \rightarrow \alpha)) \). Similarly, for a natural number \( n \), the type of \( n \)-non-diverging streams \( \text{Stream}_n[X] \) is a notation for \( \text{Rec}(n)(\alpha \Rightarrow X \times (\text{Unit} \rightarrow \alpha)) \). Using this notation, we can also define finite data structures such as lists of elements from \( X \), as follows: \( \text{List}[X] \triangleq \text{Rec}(\alpha \Rightarrow \text{Unit} + X \times \alpha) \).

We show in Section 4.3 that these types indeed correspond to streams and lists respectively.

Let Type be the set of all types. We define a (unary) logical relation on types to describe terms that do not get stuck (e.g. due to the error term \( \text{err} \), or due to an ill-formed application such as ‘true zero’) and that terminate to a value of the given type. Our definition is inspired by the notion of reducibility or hereditary termination (see e.g. [Girard 1990; Harper 2016; Tait 1967]), which we use as a guiding principle for designing the type system and its extensions.

### 4.1 Reducibility for Closed Terms

For each type \( \tau \), we define in Figure 13 mutually recursively the sets of reducible values \( [\tau]_v^\theta \) and reducible terms \( [\tau]_t^\theta \). In that sense, a type \( \tau \) can be understood as a specification that some terms satisfy (and some do not).

These definitions require an environment \( \theta \), called an interpretation, to give meaning to type variables. Concretely, an interpretation is a partial map from type variables to sets of terms. An interpretation \( \theta \) has the constraint that for every type variable \( \alpha \in \text{dom}(\theta) \), \( \theta(\alpha) \) is a reducibility candidate \( C \), which, in our setting, means that all terms in \( \theta(\alpha) \) are (erased) values. The set of all reducibility candidates is denoted by \( \text{Candidates} \subseteq \mathcal{2}_{\text{Terms}} \), and an interpretation \( \theta \) is therefore a partial map in \( \mathcal{V} \mapsto \text{Candidates} \).

When the interpretation has no influence on the definition, we may omit it. For instance, for every \( \theta \in (\mathcal{V} \mapsto \text{Candidates}) \), we have \( [\text{Nat}]_v^\theta = \{ \text{zero}, \text{succ} (\text{zero}), \text{succ} (\text{succ} (\text{zero})), \ldots \} \), so we can just denote this set by \( [\text{Nat}]_v \).

By construction, \( [\tau]_v^\theta \) only contains (erased) values (of type \( \tau \)), while \( [\tau]_t^\theta \) contains (erased) terms that reduce to a value in \( [\tau]_v^\theta \). For example, a term in \( [\text{Nat} \rightarrow \text{Nat}]_t^\theta \) is not only normalizing as a term of its own, but also normalizes whenever applied to a value in \( [\text{Nat}]_v^\theta \).

The type \( \{ x : \tau \mid b \} \) represents the values \( v \) of type \( \tau \) for which \( b[x \mapsto v] \) evaluates to \( \text{true} \). We use this type as a building block for writing specifications (pre and postconditions).

The type \( \forall x : \tau_1 \rightarrow \tau_2 \) represents the values that are in the intersection of the types \( \tau_2[x \mapsto a] \) when \( a \) ranges over values of type \( \tau_1 \). This type differs from \( \Pi x : \tau_1 \rightarrow \tau_2 \) in the sense that a value in \( \forall x : \tau_1 \rightarrow \tau_2 \) belongs to every \( \tau_2[x \mapsto a] \) for \( a \) in \( \tau_1 \), while a value in \( \Pi x : \tau_1 \rightarrow \tau_2 \) is a function that,
\[ [\alpha]^{\theta} \triangleq \theta(\alpha) \]
\[ [\tau]^{\theta} \triangleq \text{Val} \]
\[ [\text{Unit}]^{\theta} \triangleq \{()\} \]
\[ [\text{Bool}]^{\theta} \triangleq \{\text{true, false}\} \]
\[ [\text{Nat}]^{\theta} \triangleq \{\text{zero, succ(zero), succ(succ(zero)), \ldots}\} \]
\[ [\Pi x : \tau_1. \tau_2]^{\theta} \triangleq \{f \in \text{Val} \mid \forall a \in [\tau_1]^{\theta}. \ f \ a \in [\tau_2[x \mapsto a]]^{\theta}\} \]
\[ [\forall x : \tau_1. \tau_2]^{\theta} \triangleq \{b \in \text{Val} \mid \forall a \in [\tau_1]^{\theta}. \ b \in [\tau_2[x \mapsto a]]^{\theta}\} \]
\[ [\Sigma x : \tau_1. \tau_2]^{\theta} \triangleq \{(a, b) \mid a \in [\tau_1]^{\theta} \land b \in [\tau_2[x \mapsto a]]^{\theta}\} \]
\[ [\{x : \tau | b\}]^{\theta} \triangleq \{a \in [\tau]^{\theta} \mid b[x \mapsto a] \mapsto^* \text{true}\} \]
\[ [\tau_1 + \tau_2]^{\theta} \triangleq \{\text{left}(v) \mid v \in [\tau_1]^{\theta}\} \cup \{\text{right}(v) \mid v \in [\tau_2]^{\theta}\} \]
\[ [t_1 \equiv t_2]^{\theta} \triangleq \{(()) \mid t_1 \approx t_2, \text{ and } \theta \text{ otherwise, where} \]
\[ (t_1 \approx t_2) \triangleq (\forall v \in \text{Val}. \ (t_1 \mapsto^* v \iff t_2 \mapsto^* v)) \]
\[ [\forall \alpha : \text{Type}. \tau]^{\theta} \triangleq \{v \in \text{Val} \mid \forall C \in \text{Candidates}. \ v[] \in [\tau]^{\theta[\alpha \mapsto C]}\} \]
\[ [\text{Rec}(t)(\alpha \Rightarrow \tau)]^{\theta} \triangleq \{\text{fold}(v) \mid (t \mapsto^* \text{zero} \land v \in [\text{basetype}_{\alpha}(\tau)]^{\theta}) \lor \]
\[ \exists n \in [\text{Nat}]. \ t \mapsto^* \text{succ}(n) \land v \in [\tau]^{\theta[\alpha \mapsto [\text{Rec}(n)(\alpha \Rightarrow \tau)]^{\theta}]}) \}
\[ \tau^\theta_{1} \triangleq \{t \mid \exists v \in [\tau]^{\theta}. \ t \mapsto^* v\} \]

basetype_{\alpha}(\Sigma x : \tau_1. \tau_2) \triangleq \Sigma x : \text{basetype}_{\alpha}(\tau_1). \ \text{basetype}_{\alpha}(\tau_2)

basetype_{\alpha}(\tau_1 + \tau_2) \triangleq \text{basetype}_{\alpha}(\tau_1) + \text{basetype}_{\alpha}(\tau_2)

basetype_{\alpha}(\tau) \triangleq \text{if } \alpha \in \text{fv}(\tau) \text{ then } \top \text{ else } \tau

Fig. 13. Definition of reducibility for values and for terms for each type. The function basetype() is an auxiliary function, used in the base case of the definition for recursive types.

when applied to some a in τ1, produces a value in τ2[x \mapsto a]. From a value b in ∀x : τ1, τ2, we can build a value in Πx : τ1. τ2 (namely, λx. b), while the other way around is not always possible.

The sum type τ1 + τ2 represents values that are either of the form left(v) where v is a reducible value of τ1, or of the form right(v) where v is a reducible value of τ2.

The set of reducible values for the equality type [t1 \equiv t2]^{\theta} makes use of a notion of equivalence on terms which is based on operational semantics. More specifically, we say that t1 and t2 are equivalent, denoted t1 \approx t2, if for every value v, we have t1 \mapsto^* v \iff t2 \mapsto^* v. Note that this equivalence relation is defined even if we do not know anything about the types of terms t1 and t2, and it ensures that if one of the terms reduces to a value, then so does the other.

The type ∀\alpha : \text{Type}. \tau is the polymorphic type from System F. The set [\forall \alpha : \text{Type}. \tau]^{\theta} is defined by using the environment \theta to bind the type variable \alpha to an arbitrary reducibility candidate.

We use the recursive type \text{Rec}(n)(\alpha \Rightarrow \tau) as a building block for representing data structures such as lists of streams. The definition of reducibility for the recursive type makes use of an auxiliary
function `basetype()` that can be seen as an (upper) approximation of the recursive type. Note that `basetype_1(τ)` (defined at the bottom of Figure 13) removes the type variable α from τ.

Our reducibility definition respects typical lemmas that are needed to prove the soundness of typing rules, such as the following substitution lemma (see [Girard 1971] for the lemma on System F), which we have formally proven (see also Section 6 below).

**Lemma 4.1.** Let τ₁ and τ₂ be two types, and let α be a type variable that may appear in τ₁ but not in τ₂. Let θ be a type interpretation. Then, we have:

\[
\left[\left[\tau_1\right]\right]_{\theta} \equiv \left[\left[\tau_2\right]\right]_{\theta}
\]

**4.1 Well-Foundedness of the Reducibility Definition.** We can show that the definition given Figure 13 is well-founded by defining a lexicographic measure \((\text{typeNodes}(\tau), \text{index}(\tau)) \in \mathbb{N} \times \mathbb{N}\) on terms of type \(\tau\). The function `typeNodes` returns the size of the syntactic tree of type \(\tau\), ignoring the terms that appear inside. This size roughly corresponds to the number of (top-level) type constructors in the tree of \(\tau\). For example, `typeNodes({x : τ | p}) = 1 + `typeNodes(`τ`) (ignoring \(p\)). `typeNodes(τ₁ + τ₂) = 1 + `typeNodes(`τ₁`) + `typeNodes(`τ₂`), and `typeNodes(τ₁ ≡ τ₂) = 0. Ignoring the size of terms inside types ensures that given a type \(\tau\), a term variable \(x\), and a term \(a\), we have: `typeNodes(τ[x ↦ a]) = `typeNodes(`τ`). As a result, the measure `typeNodes(τ)` in the definition of reducibility decreases for indexed types such as \(Σ_1 : τ_1, τ_2 \) or \(Π_1 : τ_1, τ_2\).

The number `typeNodes(τ)` decreases in every case of Figure 13, except for recursive types `Rec(τ)(α ⇒ τ)` where the measure stays the same in the recursive call to the denotation on `Rec(n)(α ⇒ τ)` with \(t \xrightarrow{*} \text{succ}(n)\). This is where we use the second component of the lexicographic measure, `index(τ)`. We define `index(Rec(t)(α ⇒ τ)) = t` and `index(τ) = err` for every other type. Then, given \(t_1, t_2 \in \text{Terms}\), we consider \(t_1\) to be (strictly) smaller than \(t_2\) if there exist \(v_1, v_2 \in \text{Nat}\), such that \(t_1 \xrightarrow{*} v_1, t_2 \xrightarrow{*} v_2\), and \(v_1\) is strictly smaller than \(v_2\) when seen as a natural number. Therefore, the second component `index(Rec(n)(α ⇒ τ)) = n` is strictly smaller than `index(Rec(t)(α ⇒ τ)) = t` in the definition (because \(t \xrightarrow{*} \text{succ}(n)\)), which ensures that the overall lexicographic measure decreases.

**4.2 Reducibility for Open Terms**

Having defined reducibility for closed terms, we now define what it means for a term \(t\) with free term and type variables to be reducible for a type \(\tau\). Informally, we want to ensure that for every interpretation of the type variables, and for every substitution of values for the term variables, the term \(t\) reduces in a finite number of steps to a value in type \(\tau\). This is formalized by a (semantic) typing relation \(\Theta; Γ \vdash_{\text{Red}} t : \tau\) which is defined as follows.

First, a context \(\Theta; Γ\) is made of a finite set \(\Theta \subseteq V\) of type variables and of a sequence \(Γ\) of pairs in \(V \times \text{Type}\). The domain of \(Γ\), denoted \(\text{dom}(Γ)\), is the list of variables (in \(V\)) appearing in the left-hand-sides of the pairs. We implicitly assume throughout the paper that all variables appearing in the domains are distinct. This enables us to use \(Γ\) as a partial map from \(V\) to Type. We use a sequence to represent \(Γ\) as the order of variables is important, since a variable may have a (dependent) type which refers to previous variables in the context.

Given a partial map \(γ \in V \mapsto \text{Terms}\), we write \(γ(t)\) for the term \(t\) where every variable \(x\) is replaced by \(γ(x)\). We use the same notation \(γ(τ)\) for applying a substitution to a type \(τ\).

Given a context \(\Theta; Γ\), a reducible substitution for \(\Theta; Γ\) is a pair of partial maps \(θ \in V \mapsto \text{Candidates}\) and \(γ \in V \mapsto \text{Terms}\) where: \(\text{dom}(θ) = Θ\), \(\text{dom}(γ) = \text{dom}(Γ)\), and \(∀ x \in \text{dom}(Γ). γ(x) \in [γ(Γ(x))]_θ\).

Note that the substitution \(γ\) is also applied to the type \(Γ(x)\), since \(Γ(x)\) may be a dependent type with free term variables. The set of all pairs of reducible substitutions for \(Θ; Γ\) is denoted \([Θ; Γ]_γ\).
Finally, given a context $\Theta; \Gamma$, a term $t$ and a type $\tau$, we say that $\Theta; \Gamma \models_{\text{red}} t : \tau$ holds when for every pair of substitutions $\theta, \gamma$ for the context $\Theta; \Gamma$, $\gamma(t)$ belongs the reducible values at type $\gamma(\tau)$. Formally, $\Theta; \Gamma \models_{\text{red}} t : \tau$ is defined to hold when:

$$\forall \theta, \gamma \in [\Theta; \Gamma]_v, \gamma(t) \in [\gamma(\tau)]^{\theta}_1$$

Our bidirectional type checking and inference algorithm in Section 5 is a sound (even if incomplete) procedure to check $\Theta; \Gamma \models_{\text{red}} t : \tau$.

### 4.3 Recursive Types

We explain in this section how to interpret the type $\text{Rec}(\alpha \Rightarrow \tau)$ (see reducibility definition in Figure 13) and how the $\text{Stream}[X]$ and $\text{List}[X]$ types represent streams and lists.

#### 4.3.1 Infinite Streams

For a natural number $n$, consider the type $S_n \triangleq \text{Stream}[\text{Nat}] \triangleq \text{Rec}(\alpha \Rightarrow \text{Nat} \times (\text{Unit} \rightarrow \alpha))$. Let us first see what $S_n$ represents for small values of $n$. As a shortcut, we use the notations $0, 1, 2, \ldots$ for zero, $\text{succ}(\text{zero})$, $\text{succ}(\text{succ}(\text{zero}))$, $\ldots$

The definition $[S_0]_v$ refers to basetype $\alpha(\text{Nat} \times (\text{Unit} \rightarrow \alpha))$, which is $\text{Nat} \times \text{Unit}$ by definition. This means that $[S_0]_v$ is the set of values of the form $\text{fold}(a, v)$, where $a \in [\text{Nat}]_v$ and $v \in \text{Val}$.

By unrolling the definition, we get that $[S_1]_v$ is the set of values of the form $\text{fold}(v)$ where $v$ is in $[\text{Nat} \times (\text{Unit} \rightarrow \alpha)]_v^{[\text{Nat}]_v} \alpha \mapsto [S_0]_v$, which is the same (by Lemma 4.1) as $[\text{Nat} \times (\text{Unit} \rightarrow S_0)]_v$. Therefore, $[S_1]_v$ is the set of values of the form $\text{fold}(a, f)$ where $a \in [\text{Nat}]_v$ and $f \in [\text{Unit} \rightarrow S_0]_v$. This means that when $f$ is applied to $()$, $f$ terminates and returns a value in $[S_0]_v$. Similarly, $[S_2]_v$ is the set of values of the form $\text{fold}(a, f)$ where $n \in [\text{Nat}]_v$ and $f \in [\text{Unit} \rightarrow S_1]_v$.

To summarize, we can say that for every $n \in [\text{Nat}]_v$, $S_n$ represents values of the language that behave as streams of natural numbers, as long as they are unfolded at most $n + 1$ times. This matches the property we mentioned in Section 2, as $S_n$ represents the streams that are $n + 1$-non-diverging. We can show that as $n$ grows, $S_n$ gets more and more constraints: $[S_0]_v \supseteq [S_1]_v \supseteq [S_2]_v \supseteq \ldots$ In the limit, a value $v \in [\forall n : \text{Nat}. S_n]_v$ (which is in every $S_n$ for $n \in [\text{Nat}]_v$), represents a stream of natural numbers, that, regardless of the number of times it is unfolded, does not diverge, i.e. a non-diverging stream. Equivalently, we have $v \in [\text{Stream}[\text{Nat}]]_v$.

#### 4.3.2 Finite Lists

Types of the form $\text{Rec}(\alpha \Rightarrow \tau)$ can also be used to represent finite data structures such as lists. We let $\text{List}_n[X]$ be a notation for $\text{Rec}(\alpha \Rightarrow \text{Unit} + X \times \alpha)$, so that:

$$\text{List}[X] \triangleq \forall n : \text{Nat}. \text{List}_n[X].$$

Here are some examples to show how lists are encoded:

- The empty list is $\text{fold}(\text{left}())$,
- A list with one element $n$ is $\text{fold}(\text{right}(n, \text{fold}(\text{left}())))$,
- Given an element $n$ and a list $l$, we can construct the list $n :: l$ by writing: $\text{fold}(\text{right}(n, l))$.

Let us now see why $\text{List}[X]$ represents the type of all finite lists of elements in $X$. The first thing to note is that given $n \in [\text{Nat}]_v$, $\text{List}_n[X]$ does not represent the lists of size $n$. For instance, we know that $[\text{List}_0[X]]_v$ is the set of values of the form $\text{fold}(v)$ where $v \in [\text{basetype}_\alpha(\text{Unit} + X \times \alpha)]_v$, i.e. $v \in [\top]_v = \text{Val}$. Therefore, $\text{List}_0[X]$ contains lists of all sizes (and also all values that do not represent lists, such as $\text{fold}(\text{zero})$ or $\text{fold}(\lambda x. ())$).

Instead, $\text{List}_n[X]$ can be understood as the values that, as long as they are unfolded no more than $n$ times, behave as lists. As for streams, we have: $[\text{List}_0[X]]_v \supseteq [\text{List}_1[X]]_v \supseteq [\text{List}_2[X]]_v \supseteq \ldots$ where the monotonicity follows because $\alpha$ only appears in positive positions in the definitions of the recursive types for streams and lists. In the limit, we can show that $\text{List}[X]$ contains all finite lists, and nothing more.
Lemma 4.2. Let \( v \in \text{Val} \) be a value and \( X \) be some type. Then, \( v \in [\text{List}[X]]_v \) if and only if there exists \( k \geq 0 \) and \( a_1, \ldots, a_k \in [X]_v \) such that \( v = \text{fold}(\text{right}(a_1, \ldots, \text{fold}(\text{right}(a_k, \text{fold}(\text{left}())))) \ldots)) \).

It may seem surprising that the type of streams \( \text{Rec}(\alpha \Rightarrow X \times (\text{Unit} \rightarrow \alpha)) \) contains infinite streams while the type of lists \( \text{Rec}(\alpha \Rightarrow \text{Unit} + X \times \alpha) \) only contains finite lists. The reason is that, in a call-by-value language, a value representing an infinite list would need to have an infinite syntax tree, with infinitely many \( \text{fold}() \)'s (which is not possible). On the other hand, we can represent infinite streams by hiding recursion underneath a lambda term as shown in Section 2.

5 A BIDIRECTIONAL TYPE-CHECKING ALGORITHM

In this section, we give procedures for inferring a type \( \tau \) for a term \( t \) in a context \( \Theta; \Gamma \), denoted \( \Theta; \Gamma \vdash t \Downarrow \tau \), as well as for checking that the type of a term \( t \) is \( \tau \), denoted \( \Theta; \Gamma \vdash t \Downarrow t \). We introduce rules of our procedures throughout this section; the full set of rules is given in figures 14 and 15.

Our inference and checking rules give rise to conditions of the form \( \Theta; \Gamma \vdash t_1 \equiv t_2 \). We call such checks verification conditions (in the rules, they are boxed and appear in blue color). The \( \equiv \) sign is part of the judgment form, and does not describe a formula. We rely on an external solver to perform these checks, and assume that when the verification condition is considered valid by the solver, then: \( \forall \theta, \gamma \in [\Theta; \Gamma], _v (\gamma(\text{erase}(t_1))) \equiv (\gamma(\text{erase}(t_2))). \) This is an equivalent way of saying that \( [\gamma(\text{erase}(t_1))]_v \equiv [\gamma(\text{erase}(t_2))]_v \) is non-empty. Under these conditions, we have the following theorem.

Theorem 5.1 (Soundness of the Bidirectional Type-Checker). If \( \Theta; \Gamma \vdash t \Downarrow \tau \) holds or if \( \Theta; \Gamma \vdash t \Downarrow t \) holds, then \( \Theta; \text{erase}(\Gamma) \models_{\text{Red}} \text{erase}(t) : \text{erase}(\tau) \) holds.

### 5.1 Annotated Terms

In order to guide our type-checking algorithm, we require terms to be annotated. We give in Figure 16 the grammar for annotated terms. The term \( \text{inst}(t_1, t_2) \) is used to instantiate a term \( t_1 \) which has a type of the form \( \forall x : \tau_2. \tau \) to a particular term \( t_2 \) of type \( \tau_2 \), in the (Infer Forall Instantiation) type inference rule of Figure 14. The term \( \text{unfold}_\text{pos} t_1 \) in \( t_2 \) is an annotated variant of \( \text{unfold} t_1 \) in \( t_2 \) (see rules (Infer Unfold) and (Infer Unfold Positive) in Figure 14). We discuss the difference between these rules in Section 5.6.

The type \( \text{let} \ x \ = \ t_2 \ in \ \tau \) represents the type \( \tau \) where the variable \( x \) is bound to \( t_2 \) by using \( \text{let} \)'s in each term that appears in \( \tau \). The formal definition is given in Section 5.

Annotations such as \( \lambda x : \tau. \ t \) or \( \text{inst}(t_1, t_2) \) have no runtime influence and are erased (respectively to \( \lambda x. \ t \) and \( t_1 \)). We write \( \text{erase}(t) \) to refer to the erase of \( t \), where every annotation has been erased. The full definition is given in the long version of our paper [Hamza et al. 2019].

When a type \( \tau \) has annotated terms inside, we write \( \text{erase}(\tau) \) to erase their annotations. For instance \( \text{erase}(\{x : \tau \mid b\}) \) refers to \( \{x : \tau \mid \text{erase}(b)\} \). Moreover, for a context \( \Gamma \), we write \( \text{erase}(\Gamma) \) to refer to the context \( \Gamma \) where each type \( \tau \) has been replaced by \( \text{erase}(\tau) \).

### 5.2 Contracts and Measures

The syntax we support in our verification tool translates into our core calculus presented above. In our tool we support named functions with contracts and measures which are desugared into \( \text{fix} \) terms. To compare natural numbers and express the fact that measures decrease, we use functions ‘<’, ‘<=’ and ‘==’ on natural numbers. These functions can be defined using the recursor \( \text{rec} \) (see the long version of our paper for definitions [Hamza et al. 2019]).

Figure 17 shows how, thanks to refinement types, the \( \text{fix} \) term can encode recursive functions (such as the one given in Section 2) that feature user-defined pre- and post-conditions and whose
\[\Gamma(x) = \tau \quad \text{(Infer Var)} \quad \Theta; \Gamma \vdash \text{true} \# \text{Bool} \quad \text{(Infer True)} \quad \Theta; \Gamma \vdash \text{false} \# \text{Bool} \quad \text{(Infer False)} \]

\[\Theta; \Gamma \vdash \emptyset \# \text{Unit} \quad \text{(Infer Unit)} \quad \Theta; \Gamma \vdash \text{zero} \# \text{Nat} \quad \text{(Infer Zero)} \quad \Theta; \Gamma \vdash \text{succ}(t) \# \text{Nat} \quad \text{(Infer Succ)} \]

\[\Theta; \Gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 \# \text{if } t \text{ then } t_1 \text{ else } t_2 \quad \text{(Infer If)}\]

\[\Theta; \Gamma \vdash t_1 \# \text{Bool} \quad \Theta; \Gamma, p : t_1 \equiv t_2 \# \tau_2 \quad \Theta; \Gamma, p : t_1 \equiv \text{false} \# t_3 \# t_3 \quad \text{(Infer Left)} \quad \Theta; \Gamma \vdash \text{left} \# \tau_1 \quad \Theta; \Gamma \vdash \text{right} \# \\
\tau_2 \quad \Theta; \Gamma \vdash \text{Match}(t_1, \tau_1, \tau_2) \quad \text{(Infer Match)} \quad \Theta; \Gamma \vdash \text{either}\_\text{match}(t, x_1, x_2) \# \text{Either}\_\text{Match}(t, x_1, x_2) \quad \text{(Infer Either Match)} \quad \Theta; \Gamma \vdash \text{rec}[n \mapsto \tau](n, y) \Rightarrow t_1 \quad \text{Let } n = t_1 \text{ in } r \quad \text{(Infer Rec)} \]

\[\Theta; \Gamma, n : \text{Nat}, y : \text{Unit} \rightarrow \tau, p : y \equiv \lambda n : \text{Unit}. \text{rec}[n \mapsto \tau](n, y) \Rightarrow t_1 \# t_2 \# \text{fix}(n \mapsto m) \quad \text{(Infer Fix)} \]

\[\Theta; \Gamma \vdash t_1 \# \tau_1 \quad \Theta; \Gamma \vdash t_2 \# \tau_2 \quad \Theta; \Gamma \vdash (t_1, t_2) \# \text{Nat} \quad \text{(Infer Pair)} \quad \Theta; \Gamma \vdash \text{infl}(x_1 : \tau_1, \tau_2) \quad \text{(Infer Proj1)} \]

\[\Theta; \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \# \text{let } x = t_1 \text{ in } t_2 \quad \Theta; \Gamma \vdash x_1 : \tau_1 \# \tau_2 \quad \Theta; \Gamma \vdash \text{let } x_1 : \tau_1 \text{ in } t_2 \# \text{let } x_1 : \tau_1 \text{ in } t_2 \quad \text{(Infer Lett)} \quad \Theta; \Gamma, \lambda x : \tau \vdash t \# \tau \quad \Theta; \Gamma \vdash \text{infl}(x_1 : \tau_1, \tau_2) \quad \text{(Infer Proj2)} \]

\[\Theta; \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \quad \Theta; \Gamma, x : \tau_1 \# \tau_2 \quad \Theta; \Gamma \vdash \text{fix}(n \mapsto m) \quad \text{(Infer Fix)} \quad \Theta; \Gamma \vdash \text{let } x = \text{rec}[n \mapsto \tau](n, y) \Rightarrow t_1 \text{ in } t_2 \quad \text{(Infer App)} \]

\[\Theta; \Gamma \vdash \text{infl}(x_1 : \tau_1, \tau_2) \quad \Theta; \Gamma \vdash \text{let } x = \text{rec}[n \mapsto \tau](n, y) \Rightarrow t_1 \text{ in } t_2 \quad \Theta; \Gamma \vdash x_1 : \tau_1 \# \tau_2 \quad \Theta; \Gamma \vdash x_1 : \tau_1 \# \tau_2 \quad \text{(Infer Type App)} \quad \Theta; \Gamma \vdash x : \tau_1 \# \tau_2 \quad \Theta; \Gamma \vdash \text{inst}(x_1 : \tau_1, \tau_2) \quad \text{(Infer Forall Instantiation)} \]

\[\Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \text{(Infer Fold)} \quad \Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \text{(Infer Fold)} \]

\[\Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \text{(Infer Unfold)} \]

\[\Theta; \Gamma \vdash t_1 \# \text{Rec}(n)(\alpha \mapsto \tau) \quad \Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \text{(Infer Unfold Positive)} \]

\[\Theta; \Gamma \vdash \text{lessThan } 0 \# \text{true} \quad \Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \text{(Infer Unfold Positive)} \]

\[\Theta; \Gamma \vdash \text{true} \equiv \text{false} \quad \Theta; \Gamma \vdash \text{err} \# \tau \quad \text{(Infer Err)} \quad \Theta; \Gamma \vdash \text{refl}(t_1, t_2) \# t_1 \equiv t_2 \quad \text{(Infer Refl)} \]

\[\Theta; \Gamma \vdash t \# \tau \quad \Theta; \Gamma \vdash \text{fold}(\text{rec}(n)(\alpha \mapsto \tau))(\alpha \mapsto \tau) \quad \text{(Infer Refinement)} \]

Fig. 14. \(\Theta; \Gamma \vdash t \# \tau\) infers a type \(\tau\) for \(t\) in context \(\Theta; \Gamma\) based on the shape of \(t\). The (Infer Drop Refinement) rule is applied with low priority, only if no other rule is applicable, keeping type checking deterministic.
\[\Theta, \Gamma \vdash t \downarrow \tau \quad \Theta, \Gamma, p : t_1 \equiv \top \vdash t_2 \downarrow \tau \quad \Theta, \Gamma, p : t_1 \equiv \bot \vdash t_3 \downarrow \tau \quad (\text{Check If})\]

\[\Theta, \Gamma \vdash t_n \downarrow \text{Nat} \quad \Theta, \Gamma, p : t_n \equiv \mathsf{zero} \vdash t_0 \downarrow \tau \quad \Theta, \Gamma, n : \text{Nat}, p : t_n \equiv \mathsf{succ}(n) \vdash t_s \downarrow \tau \quad (\text{Check Match})\]

\[\Theta, \Gamma \vdash t \downarrow \tau_1 + \tau_2 \quad \Theta, \Gamma, x : r_1, p : t \equiv \mathsf{left}(x) \vdash t_1 \downarrow \tau \quad \Theta, \Gamma, x : r_2, p : t \equiv \mathsf{right}(x) \vdash t_2 \downarrow \tau \quad (\text{Check Either Match})\]

\[\Theta, \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \downarrow \tau \quad \Theta, \Gamma, x : r_1, p : x \equiv t_1 + t_2 \downarrow \tau \quad (\text{Check Let})\]

\[\Theta, \Gamma \vdash \Pi : x : r_1, r_2 \quad (\text{Check Pi})\]

\[\Theta, \Gamma \vdash \text{inst}(t, n) \downarrow \tau \quad \Theta, \Gamma \vdash \text{match}(t, n \Rightarrow t_1) \downarrow \tau \quad (\text{Check Forall})\]

\[\Theta, \Gamma \vdash \text{left}(x) \downarrow \tau \quad \Theta, \Gamma, x : r_1, p : x \equiv \pi_1 t_1 + \pi_2 t_2 \downarrow \tau \quad (\text{Check Sigma})\]

\[\Theta, \alpha; \Gamma \vdash t[\alpha] \downarrow \tau \quad \Theta, \Gamma \vdash \text{Rec}(n_2)(\alpha \Rightarrow \tau) \quad \Theta, \Gamma \vdash n_1 \equiv n_2 \quad (\text{Check Recursive})\]

\[\Theta, \Gamma \vdash t \downarrow \tau \quad \Theta, \Gamma \vdash t \downarrow \forall x : \text{Type, } \tau \quad (\text{Check Type Abs})\]

\[\Theta, \Gamma \vdash t \downarrow \tau \quad \Theta, \Gamma \vdash t \downarrow \forall x \downarrow \tau \quad (\text{Check Top 1})\]

\[\Theta, \alpha; \Gamma \vdash t[\alpha] \downarrow \tau \quad \Theta, \Gamma \vdash t[\alpha] \downarrow \tau \quad (\text{Check Top 2})\]

\[\Theta, \Gamma \vdash t \downarrow \tau \quad \Theta, \Gamma \vdash t \downarrow \tau \quad (\text{Check Reflective})\]

Fig. 15. \(\Theta, \Gamma \vdash t \downarrow \tau\) checks that term \(t\) indeed has type \(\tau\) under context \(\Theta; \Gamma\). When multiple rules are applicable, they are applied from a priority order from top to bottom, left to right. The (Check Forall) rule can be generalized to (non-empty) types other than \(\text{Nat}\), but we only need it for \(\text{Nat}\) (in type \(\text{Rec}(\alpha \Rightarrow \tau)\)).

\[
t := x \mid () \mid \lambda x : \tau, t \mid t t \mid (t, t) \mid \pi_1 t \mid \pi_2 t \mid \text{inst}(t, n) \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if } t \text{ then } t \text{ else } t \mid \mathsf{zero} \mid \mathsf{succ}(t) \mid \mathsf{rec}(x \Rightarrow \tau)(t, t, (n, y) \Rightarrow t) \mid \mathsf{fix}(n \Rightarrow \tau)(n, y) \Rightarrow t) \mid \mathsf{match}(t, n \Rightarrow t) \mid \mathsf{fold}(\tau)(t) \mid \mathsf{fold}(\tau)(t) \mid \mathsf{unfold}(t, n \Rightarrow t) \mid \mathsf{unfold}(t, n \Rightarrow t) \mid \Lambda \alpha. t \mid t[\tau] \mid \mathsf{err}(\tau) \mid \mathsf{refl}(t, t) \mid \text{let } x = t \text{ in } t \mid \mathsf{size}(t)
\]

Fig. 16. Grammar for annotated terms \(t\), where \(x, y\) and \(n\) are term variables and \(\alpha\) is a type variable.

termination arguments relies on a user-defined \textit{measure} function. The \texttt{fix} term shown on the right corresponds to the desugaring of the recursive function on the left whose contracts are given by the \texttt{require} and \texttt{ensuring} keywords, and whose measure is given by the \texttt{decreases} keyword. The contract terms \texttt{pre} and \texttt{post} are such that \(x : r_1 \vdash \texttt{pre} : \texttt{Bool}\) and \(x : r_1, \texttt{res} : r_2 \vdash \texttt{post} : \texttt{Bool}\), and the measure function \texttt{measure} satisfies \(x : r_1 \vdash \texttt{measure} : \texttt{Nat}\). The term \texttt{pred} is a function of type \(\{n : \text{Nat} \mid \text{zero} < n\} \rightarrow \text{Nat}\) that returns the predecessor of numbers greater than zero.

We now explain how our type-checking algorithm ensures termination of such a function. Our type inference rule for \texttt{fix} is (\textit{Infer Fix}). The side condition \(n \notin \mathsf{fv(erase(t))}\) ensures that \(n\) only appears in type annotations in \(t\), and is not part of the computation. The other check corresponds to a proof by strong induction (over \(n\)) that the \texttt{fix} term has type \(\forall n : \text{Nat}. \tau\). Indeed, we have to check that \(t\), the body of the \texttt{fix} term, has type \(\tau\) (for some \(n : \text{Nat}\)), under the assumption that \(y\) (which is the variable representing the recursion) has type \(\tau[n \mapsto m]\) for all \(m < n\). The ‘Unit →'
\textbf{def} f(x: \tau_1): \tau_2 = \\
\text{require}(\text{pre}[x])
\text{decreases}(\text{measure}[x])
\text{ensuring} \\
\{ res \Rightarrow \text{post}[x, \text{res}] \}
\begin{aligned}
\text{fix}[n] \Rightarrow \Pi x : \{ x : \tau_1 \mid \text{pre} \} \mid \text{measure} \leq n \}. \{ \text{res} : \tau_2 \mid \text{post} \}
\{ \text{res} \Rightarrow \text{post}[x, \text{res}] \}

f \equiv
\begin{aligned}
E[x, f]
(\text{pre}, \text{measure})
\lambda x : \{ x : \tau_1 \mid \text{pre} \} \mid \text{measure} \leq n).
(\text{res}, \tau_2) \Rightarrow \Pi x : \{ x : \tau_1 \mid \text{pre} \} \mid \text{measure} \leq n.
(\text{res}, \text{post})
\end{aligned}
\]

Fig. 17. Encoding named function with pre- and post-conditions are given by the \textbf{require} and \textbf{ensuring}, and measure given by the \textbf{decreases} keyword (left) into a terminating fixpoint recursion (right).

part of the type of \( y \) corresponds to the fact that the operational semantics of \texttt{fix} replaces variable \( y \) by the \texttt{fix} term under a lambda (as explained in Section 3).

The variable \( p \) is a witness that the variable \( y \) is equal to the \texttt{fix} term (under a lambda). This feature is useful for \textit{body-visible} recursion, and is explained in Section 5.3.

Back to the encoding presented in Figure 17, we explain how the \textit{(Infer Fix)} rule ensures that \textit{measure} decreases at each recursive call of function \( f \). Assume that the premise of the \textit{(Infer Fix)} rule holds, and that \( f \) is called with some value \( v \) of type \( \tau_1 \), such that \textit{measure}\([v]\) evaluates to some (term representing a) natural number \( n \). By instantiating the premise of the \textit{(Infer Fix)} rule for that particular \( n \), we get that \( E[x, \text{inst}(f, \text{pred}(n))()] \) is well-typed under the condition that \( f \) has type:

\[
\forall m : \{ m : \text{Nat} \mid m < n \}. \text{Unit} \Rightarrow \Pi x : \{ x : \tau_1 \mid \text{pre} \} \mid \text{measure} \leq m \}. \{ \text{res} : \tau_2 \mid \text{post} \}
\]

First, in order for \texttt{pred} to be applied to \( n \), we have to check that \( n \) is non-zero, meaning that the measure of \( v \) is strictly positive in the places where the recursive calls happen. This is ensured by the \textit{(Check Refinement)} rule for checking refinement types (see Figure 15), which generates a verification condition.

Second, the rule \textit{(Infer Forall Instantiation)} ensures that \texttt{inst}(\( f \), \texttt{pred}(\( n \))()) takes arguments of type \( \{ x : \{ x : \tau_1 \mid \text{pre} \} \mid \text{measure} \leq \text{pred}(n) \} \). Therefore, if \( f \) is applied recursively to an argument \( v' \), the rule \textit{(Check Refinement)} ensures that \textit{measure}([\( v' \]) \leq \text{pred}(n) \) holds. Overall, we get \texttt{measure}([\( v' \]) \leq \text{pred}(n) < n = \text{measure}([\( v \]), which ensures that the measures of arguments always decrease on recursive calls to \( f \).

In our implementation, we do not go through the encoding with \texttt{fix} and forall types, but instead directly generate the verification conditions that correspond to the measure decreasing by using the left-hand-side form of Figure 17. Our system also supports mutually recursive functions (by requiring that the measure decreases for each call to a mutually recursive function), which can be encoded in the usual way by defining a \texttt{fix} term that returns a tuple of functions.

In the end, if the body of the function is well-typed, the \textit{(Infer Fix)} rule infers the type:

\[
\forall n : \text{Nat}. \Pi x : \{ x : \tau_1 \mid \text{pre} \} \mid m \leq n \}. \{ \text{res} : \tau_2 \mid \text{post} \}
\]

One should note that this encoding imposes a scoping restriction on the original program, namely precondition, postcondition, and measure of a function \( f \) cannot contain calls to \( f \). This restriction has not proved limiting in our experience with benchmarks.

5.2.1 Lexicographic Orderings. Functions whose termination arguments require lexicographic orderings can be encoded by using two levels of recursions, which is a known technique that shows expressive power of System T [Girard 1990, Section 7.3.2]. We review how this encoding works in our system in the long version of our paper [Hamza et al. 2019] and show an example of Ackerman’s function and its simple lexicographic measure. In our implementation, we support lexicographic measures directly.
5.3 Body-Visible Recursion

In this section, we give more details about the (Infer Fix) and (Infer Rec) typing rules for recursion. They allow body-visible recursion which gives the type-checker access to the definition of a recursive function while type-checking the body of the recursive function itself.

The first thing to note is that we introduce an equality type containing the definition (in the type of p) in the context, while we do not know yet whether the body of the recursion is well-typed. Since our equality type is defined (in Section 4) for all terms, regardless of whether they are well-typed, this is perfectly legal. We show in the merge example of Figure 5 how body-visible recursion relieves the user from writing excessive specification annotations.

Assume we want to prove, on paper, that merge indeed returns a sorted list when given two sorted lists l1 and l2, by induction over size(l1) + size(l2). Consider the first branch of the if then else statement, where we return Cons(x, merge(xs,l2)). By the induction hypothesis, we know that the recursive call merge(xs,l2) is sorted, but this mere fact is not enough to conclude that Cons(x, merge(xs,l2)) is sorted. By unfolding the definition of isSorted, we see that we need in addition to know that x is smaller than the head of the result merge(xs,l2).

Therefore, the property we prove by induction needs to be strengthened by saying that the head of the result, if non-empty, is equal to the (smallest) head of one of the input lists. From that, we will know by the induction hypothesis that the head h of merge(xs,l2) (if non-empty) is either the head of xs or the head of l2. In the first case, we can deduce that x is smaller than h by using the fact that l1 = Cons(x,xs) is sorted. In the second case, we have h = y, and we know from the condition of the if then else statement that x ≤ y. In both cases, we can conclude that the whole list Cons(x, merge(xs,l2)) is sorted.

If we are to type-check the program above, and if we only know the return type of merge(xs,l2), that is { l: List | isSorted(l) }, we will run into the same problem, and will not be able to conclude that Cons(x, merge(xs,l2)) is sorted. In our type system, we get in addition access to the definition of merge while type-checking it, thanks to the p variable of equality type in the (Infer Fix) rule. By unfolding the definition of merge(xs,l2), we conclude by case analysis that the head h of merge(xs,l2) (if non-empty) is either the head of xs or the head of l2 (which is y).

Without body-visible recursion, the developer would need to strengthen the postcondition to:

\[
\text{isEmpty}(\text{res}) \lor (\text{isEmpty}(\text{l1}) \land \text{head}\text{(res)} == \text{head}\text{(l1)}) \\
\lor (\text{isEmpty}(\text{l2}) \land \text{head}\text{(res)} == \text{head}\text{(l2)})
\]

In Inox, the external solver we use for verification conditions, definitions of recursive functions are unfolded automatically. Inox does incremental queries to SMT solvers. It first sends a query without unfolding at all, then a new query after unfolding once, and so on until a query succeeds or a timeout. Thanks to this approach, Inox does not rely on universal quantifiers to encode recursive functions. This feature is crucial to have such examples be verified without user intervention, and is here required to get the bodies of the calls to merge and isSorted when verifying the postcondition of merge.

5.4 Unification in Type Inference

In order to perform type inference, we expect a certain structure on the inferred types. For example, given term t_1 t_2, we expect term t_1 to have a function type when inferring the type of an application. Furthermore, type inference must perform least upper bound computation for if, match and either_match terms, which adds more complexity to the system. We handle these considerations in a generalized manner by performing type unifications, defined in Figure 18 by using the type-level notations If Then Else, Let, Either_Match, and Match.
strictly positively in $\tau$ respect to our reducibility definition. This lemma states that when a type variable

Conversely, we use the rule

Paper [Hamza et al. 2019] for the precise definition). This enables us, under some conditions, to

Beyond general rules of Figure 14, in Figure 19 we show two additional rules for ‘fold’ and ‘unfold’ in that ignore the indices hidden under the Rec type for strictly positive recursive types (which is the case for the List and Stream types). We write $\text{spos}(\alpha, \tau)$ when a type variable appears only strictly positively in $\tau$, meaning only to the right of $\Pi$ and $\forall$ types (see the long version of our paper [Hamza et al. 2019] for the precise definition). This enables us, under some conditions, to fold and unfold a strictly positive recursive type without worrying about indices.

Practically, given an element $l$ of type $\text{List}[X]$ (resp. $\text{Stream}[X]$), we can unfold it (with the (Infer Unfold Gen) rule) to get its head and its tail of type $\text{List}[X]$ (resp. $\text{Unit} \rightarrow \text{Stream}[X]$). Conversely, we use the rule (Infer Fold Gen) to build a list or a stream from an element and a tail.

Strict positivity gives us the following key lemma that ensures the soundness of our rules with respect to our reducibility definition. This lemma states that when a type variable $\alpha$ appears only strictly positively in $\tau$, then quantifying with a forall type outside $\tau$ or inside a substitution for $\alpha$ is

We draw attention here to the fact that our type checking and inference procedures are syntax directed and predictable, which enables a natural verification process through term-level hints. The algorithmic nature of our type checking precludes the verification of certain well-typed programs. However, our experience has shown that this limitation is largely inconsequential in practice and is outweighed by the predictable nature of the algorithm.

5.5 Hiding Recursive Type Indices

Beyond general rules of Figure 14, in Figure 19 we show two additional rules for ‘fold’ and ‘unfold’ in that ignore the indices hidden under the Rec type for strictly positive recursive types (which is the case for the List and Stream types). We write $\text{spos}(\alpha, \tau)$ when a type variable appears only strictly positively in $\tau$, meaning only to the right of $\Pi$ and $\forall$ types (see the long version of our paper [Hamza et al. 2019] for the precise definition). This enables us, under some conditions, to fold and unfold a strictly positive recursive type without worrying about indices.

Fig. 18. Recursion schema for unification of two types, where $E$ is a context with two holes.

\[
\Theta, \Gamma \vdash t_1 \Downarrow \text{Rec}(\alpha \Rightarrow \tau) \quad \Theta, \Gamma, x : \text{Rec}(\alpha \Rightarrow \tau) \vdash p : t_1 \equiv \text{fold}(x) + t_2 \Downarrow \tau' \quad \Theta - \text{spos}(\alpha, \tau) \quad \Theta, \Gamma \vdash \text{unfold} t_1 \text{ in } x \Rightarrow t_2 \Downarrow \tau' \quad \text{(Infer Unfold Gen)}
\]

\[
\Gamma \vdash t \Downarrow \text{Rec}(\alpha \Rightarrow \tau) \quad \Theta - \text{spos}(\alpha, \tau) \quad \Gamma \vdash \text{fold} \{ \text{Rec}(\alpha \Rightarrow \tau) \}(t) \Downarrow \text{Rec}(\alpha \Rightarrow \tau) \quad \text{(Infer Fold Gen)}
\]
the same (as long as we are quantifying over a non-empty type \( \tau \)). This property is similar to the notions \( \text{lim sup-pushable} \) and \( \text{lim inf-pullable} \) [Abel 2008] and implies the soundness of our rules.

**Lemma 5.2.** Let \( \tau \) and \( \forall x : \tau_1 \cdot \tau_2 \) be two types. Let \( \alpha \) be a type variable that appears strictly positively in \( \tau \). Let \( \theta \) be a type interpretation such that \( \llbracket \tau_1 \rrbracket_\theta \) is not empty. We have:

\[
[\tau][\alpha \mapsto \forall x : \tau_1 \cdot \tau_2]]_\theta = [\forall x : \tau_1 \cdot \tau][\alpha \mapsto \tau_2]]_\theta
\]

### 5.6 Type-Checking Algorithm Examples: Streams

#### 5.6.1 Constant Stream

The \( \text{fix} \) term and associated typing rules can also be used to express the kind of recursion used to define the streams in Section 2. We start by revisiting the constant stream, which in our notations can be written as an untyped term:

\[
\text{constant} \triangleq \text{fix}(\text{constant} \Rightarrow \Lambda \lambda x. \text{fold}(x, \lambda u. \text{constant}(\llbracket x \rrbracket(x)))
\]

Assume we want to prove, on paper, that for any value \( x \), \( \text{constant}(\llbracket x \rrbracket(x)) \) produces a non-diverging stream, i.e. a stream which is \( n \)-non-diverging for every \( n \in \mathbb{N} \). A natural proof could be done by induction on \( n \), as follows:

- \((n = 0)\) \( \text{constant}(\llbracket x \rrbracket(x)) \) is \( 0 \)-non-diverging, meaning that it reduces to a value of the form \( \text{fold}(x, v) \) where \( x \) and \( v \) are values. This is clear from the code of \text{constant}, as this expression evaluates in a few steps to \( \text{fold}(x, \lambda u. (\lambda u'. \text{constant})(\llbracket x \rrbracket(x))) \).
- \((n = n' + 1)\) Assume by induction that \( \text{constant}(\llbracket x \rrbracket(x)) \) is \( n' \)-non-diverging. By definition of \( n \)-non-diverging, we get that \( \text{fold}(x, \lambda u : \text{Unit}. \text{constant}(\llbracket x \rrbracket(x))) \) is \( n \)-non-diverging. Since this term is equivalent to the term to which \( \text{constant}(\llbracket x \rrbracket(x)) \) evaluates, we conclude that \( \text{constant}(\llbracket x \rrbracket(x)) \) is \( n \)-non-diverging as well.

Our type system and type-checking algorithm can be used to simulate this proof by using an annotated version of \text{constant}:

\[
\text{constant} \triangleq \text{fix}[n \Rightarrow \forall X : \text{Type}. X \rightarrow \text{Stream}_n[X]]((n, \text{constant}) \Rightarrow \text{body}(n, \text{constant}))
\]

where \( \text{body}(n, \text{constant}) \) is a shorthand for

\[
\forall X. \lambda x : X. \text{fold}[\text{Stream}_n[X]][x, \lambda u : \text{Unit}. \text{inst}(\text{constant}, \text{pred}(n)))(\llbracket X(x) \rrbracket)
\]

By applying the \( \text{(Infer Fix)} \) rule presented above, we get the type

\[
\vdash \text{constant} \upharpoonright \forall n : \text{Nat}. \forall X : \text{Type}. X \rightarrow \text{Stream}_n[X].
\]

The \( \text{(Infer Fix)} \) rule of our algorithm generates a check that corresponds to a (strong) induction that shows that for every \( n \in \llbracket \text{Nat} \rrbracket_\theta \), \( \text{constant}(\llbracket x \rrbracket(x)) \) in \( \llbracket \text{Stream}_n[X] \rrbracket_\theta \) assuming that it is in \( \llbracket \text{Stream}_m[X] \rrbracket_\theta \) for all \( m < n \):

\[
n : \text{Nat}, \text{constant} \vdash \forall m : \{m : \text{Nat} \mid m < n\}. \text{Unit} \rightarrow \forall X : \text{Type}. X \rightarrow \text{Stream}_n[X] \vdash \text{body}(n, \text{constant}) \upharpoonright \forall X : \text{Type}. X \rightarrow \text{Stream}_n[X]
\]

After applying standard rules related to \( \lambda \) and \( \Lambda \), our algorithm will attempt to infer, using the \( \text{(Infer Fold)} \) rule, a type for the term:

\[
\text{fold}[\text{Stream}_n[X]][x, \lambda u : \text{Unit}. \text{inst}(\text{constant}, \text{pred}(n)))(\llbracket X(x) \rrbracket)
\]

In addition to the type-check that \( n \) has type \( \text{Nat} \), this rule generates two checks to cover the cases where \( n \) is zero or non-zero. These correspond to the informal proof by induction given above for the non-divergence of \text{constant}. The first check reduces (after applying some straightforward rules) to checking that \( \lambda u : \text{Unit}. \text{inst}(\text{constant}, \text{pred}(n)))(\llbracket X(x) \rrbracket \) has type \( \tau \) (remember that \( \text{basetype}_{\alpha}(X \times (\text{Unit} \rightarrow \tau)) = X \times \tau \)), which goes through easily thanks to the rule \( \text{(Check Top 1)} \).
zipWith ≝ \text{fix}[n \Rightarrow \forall X, Y, Z : \text{Type}.(X \rightarrow Y \rightarrow Z) \rightarrow \text{Stream}_n[X] \rightarrow \text{Stream}_n[Y] \rightarrow \text{Stream}_n[Z]](n, \text{zipWith}) \\
\text{fold}[	ext{Stream}_n[Z]](\text{unfold s1 in } x_1 \Rightarrow \text{unfold s2 in } x_2 \Rightarrow f(\pi_1 x_1)(\pi_1 x_2), \lambda u : \text{Unit}. \text{unfold}_\text{pos} s_1 \text{ in } x_1 \Rightarrow \text{unfold}_\text{pos} s_2 \text{ in } x_2 \Rightarrow \text{inst}(\text{zipWith, pred(n))}(X)[Y][Z](\langle \pi_2 \text{ x_1} \rangle(\langle \pi_2 \text{ x_2} \rangle())))

Fig. 20. An annotated term of our calculus to define \text{zipWith}.

The second check amounts to checking that \text{inst(constant, pred(n))}(X)(x) has type \text{Stream}_n[X] under the assumption that \text{n} = \text{succ}(n'). By the context we know that \text{inst(constant, pred(n))}(X)(x) has type \text{Stream}_{\text{pred}(n)}[X]. Since \text{n'} and \text{pred(n)} are equivalent, we can use the rule (\text{Check Recursive}) (given in Figure 15) to convert between the two types.

Attempting to type-check the \text{badConstant} example from Section 2:

\text{fix}\[n \Rightarrow \forall X : \text{Type}. X \rightarrow \text{Stream}_n[X]](n, \text{badConstant} \Rightarrow \text{badBody}(n, \text{badConstant}))

where \text{badBody}(n, \text{badConstant}) stands for:

\text{\lambda X. \lambda x : X. fold}[	ext{Stream}_n[X]](x, \lambda u : \text{Unit}. \text{inst(badConstant, pred(n))}(X)(x).\text{tail}())

will lead to an error in the second check (corresponding to the inductive case), since the extra call to \text{tail()} decreases the index of the stream by one.

5.6.2 ZipWith Function on Streams. We now revisit the \text{zipWith} function from Figure 7. We said in Section 2 that for every \text{n} \in \mathbb{N}, when \text{s1} and \text{s2} are \text{n}-non-diverging streams, then so is \text{zipWith}(f, s_1, s_2). On paper, we can check by induction over \text{n} \in \mathbb{N} that when \text{s1} and \text{s2} are \text{n}-non-diverging streams and \text{f} is terminating, then \text{zipWith}(f, s_1, s_2) (as written in Figure 7, and ignoring type annotations and the erasable parameter for the moment) is also an \text{n}-non-diverging stream.

- \(n = 0\) It is indeed the case that we can access the head of \text{zipWith}(f, s_1, s_2) as long as we can access the heads \text{s1} and \text{s2} (and as long as \text{f} terminates).
- \(n = n' + 1\) Let \text{s1} and \text{s2} be two \text{n}-non-diverging streams. By definition of \text{n}-non-diverging, we know that \text{s1}.\text{tail}() and \text{s2}.\text{tail}() are \text{n'}-non-diverging. By induction hypothesis, \text{zipWith}(f, \text{s1}.\text{tail}(), \text{s2}.\text{tail}()) is \text{n'}-non-diverging as well. This means that \text{zipWith}(f, \text{s1}, \text{s2}) is \text{n}-non-diverging, which concludes the proof.

Accordingly, our type-checking algorithm is able to infer the type

\(\forall n : \text{Nat.} \forall X, Y, Z : \text{Type}. (X \rightarrow Y \rightarrow Z) \rightarrow \text{Stream}_n[X] \rightarrow \text{Stream}_n[Y] \rightarrow \text{Stream}_n[Z].\)

for the annotated \text{zipWith} term given in Figure 20. We use the term 'unfold \text{in}' to access the heads (with \text{\pi_1}). For the tails, we use instead 'unfold_pose \text{in}'. To understand the difference, we must first look at the (\text{Infer Fold}) rule for the fold term, which will generate two subgoals, one with \text{n} = 0 and one with \text{n} > 0. In the subgoal with \text{n} = 0, we must check that the lambda term \('\lambda u : \text{Unit}.[...]'\) has type \text{T}, which goes through directly thanks to (\text{Check Top 1}). Therefore, when
we are type-checking the body of the lambda, we know that $n > 0$. We can thus access the tails using (Infer Unfold Positive), a variant of (Infer Unfold) that discards a subgoal with $n = 0$ but requires proving $n > 0$ instead.

5.6.3 Fibonacci Stream. We now consider the Fibonacci stream function from Figure 8. We can prove by induction that for all $n \in \mathbb{N}$, $\text{fib}$ is an $n$-non-diverging stream. In our view, this corresponds to writing `fib` as in Figure 21.

When type-checking `fib`, we use the type of `zipWith` and instantiate it to `pred(pred(n))`, and get that: `inst(zipWith, pred(pred(n)))` has type:

$$
\forall X, Y, Z : \text{Type}. (X \rightarrow Y \rightarrow Z) \rightarrow \text{Stream}_{\text{pred(pred(n))}}[X] \rightarrow \text{Stream}_{\text{pred(pred(n))}}[Y] \rightarrow \text{Stream}_{\text{pred(pred(n))}}[Z]
$$

This is where we use the fact that `zipWith` returns an $n$-non-diverging stream when given $n$-non-diverging streams.

5.7 Verification of Properties

Finally, we show how to verify the property mentioned in Section 2: that zipping two streams $s_1$ and $s_2$ with the function $\lambda x. \lambda y. x$ returns a stream equivalent to $s_1$.

We define a function $\text{zipWith}_\text{fst}$ with the following type:

$$
\Pi n : \text{Nat}. \Pi X : \text{Type}. \Pi s_1 : \text{Stream}[X]. \Pi s_2 : \text{Stream}[X]. \Pi n s_1 s_2 \equiv \Pi n (\text{zipWith}(\lambda x. \lambda y. x, s_1, s_2))
$$

For short, we denote this type as $\Pi n : \text{Nat}. \tau$. The code of $\text{zipWith}_\text{fst}$ is given in Figure 22 and makes use of the recursor $\text{rec}$. Our (Infer Rec) rule for inferring the type of $\text{rec}$ is similar to $\text{fix}$. The difference is that $\text{rec}$ allows references to $n$ in the code, so the computation is allowed to
We represent terms and types using a representation, lambda and other binders can be seen as terms with holes, which can be filled recursively based on a well-founded measure.

\[
\begin{align*}
[\text{⊥}]^\theta_v & \triangleq \{\} \\
[\tau_1 \cap \tau_2]^\theta_v & \triangleq \left([\tau_1]^\theta_v \cap [\tau_2]^\theta_v\right) \\
[\tau_1 \cup \tau_2]^\theta_v & \triangleq \left([\tau_1]^\theta_v \cup [\tau_2]^\theta_v\right) \\
[\exists x : \tau_1. \tau_2]^\theta_v & \triangleq \{b \mid \exists a \in [\tau_1]^\theta_v. b \in \left([\tau_2[x \mapsto a]]^\theta_v\right)\} \quad \text{(existential)} \\
[[x : \tau_1 | \tau_2]]^\theta_v & \triangleq \{a \in [\tau_1]^\theta_v | \exists b \in \left([\tau_2[x \mapsto a]]^\theta_v\right)\} \quad \text{(refinement by type)} \\
[\text{Let } x = t \text{ in } \tau]^\theta_v & \triangleq \{v \mid \exists a \in \text{Val}. t \mapsto_a a \land v \in \left[[\tau[x \mapsto a]]^\theta_v\right)\}
\end{align*}
\]

Fig. 23. Definition of reducibility for intersection, union, singleton types, and refinement by types.

depend on \(n\). Moreover, \texttt{rec} does not use \(\forall\) types. Finally, \texttt{rec} is analogous to a simple induction, while \texttt{fix} is analogous to a strong induction.

In the \texttt{zipWith\_fst} example, applying this rule corresponds to a proof by induction over \(n\), that for any two streams \(s_1\) and \(s_2\), the \(n\)th element of \(s_1\) is equal to the \(n\)th element of \(zipWith(\lambda x. \lambda y. x, s_1, s_2)\).

In the base case, to type-check the term \texttt{refl}[\texttt{nth 0 s1, nth 0 (zipWith(\lambda x. \lambda y. x, s1, s2))}], we must make sure that the following equality holds: \(\texttt{nth 0 s1} \equiv \texttt{nth 0 (zipWith(\lambda x. \lambda y. x, s1, s2))}\).

The corresponding type inference rule is \(\texttt{(Infer Refl)}\), which generates a verification condition.

In the inductive case, we explicitly instantiate our inductive hypothesis on the tails of \(s_1\) and \(s_2\) by using the \texttt{let} binding on variable \texttt{pr}. The equality given by the type of \texttt{pr} is then sufficient to prove what we wanted: \(\texttt{nth (succ(n)) s1} \equiv \texttt{nth (succ(n)) (zipWith(\lambda x. \lambda y. x, s1, s2))}\).

This example also illustrates that developers can express the desired versions of what others might call extensional equality (here: same results when calling \texttt{nth}) using \(\Pi\) types and equi-reducibility \(\equiv\); our type system does not impose any preferred form of extensional equality.

6 FORMALIZATION IN THE COQ PROOF ASSISTANT

We here give more details about our formalization of Theorem 5.1 in Coq 8.9.1 (including the rules from Section 5.5). Our proofs are available from


We represent terms and types using a locally nameless representation [Charguéraud 2012], where free variables are named, and where local variables are bound using De Bruijn indices. Using this representation, lambdas and other binders can be seen as terms with holes, which can be filled with other terms (typically values). We use the Coq Equations library [Sozeau 2010] to define the reducibility logical relation \([\_\_\_party\text{._v}]. This library facilitates the use of functions which are defined recursively based on a well-founded measure.

We give an overview of our files (containing around 20k lines of code):

- \texttt{Trees.v} contains the definitions of types and terms,
- \texttt{Typing.v} gives all typing rules (containing rules from the paper and more),
- \texttt{SmallStep.v} contains the operational semantics of the language,
- \texttt{ReducibilityDefinition.v} contains the definition of reducibility,
- The \texttt{Reducibility\_star.v} files contain lemmas for the soundness of the rules from \texttt{Typing.v},
- \texttt{Reducibility.v} contains the proof that all typing rules from \texttt{Typing.v} are sound with respect to the reducibility definition (which implies that Theorem 5.1 holds).

6.1 Extensions of Formalization

Our approach to formalization proved flexible and allowed us to extend the system with additional types. In addition to formalizing the types and rules for our type checking algorithm (figures 15 and 15), our Coq formalization also defines (Figure 23) reducibility for the bottom type, singleton types,
union and intersection types, and an existential type more abstract than dependent pair (and dual to \( \forall x : \tau_1, \tau_2 \)). We also formalized refinement by type \( \{ x : \tau_1 | \tau_2 \} \), which generalizes refinements from computable terminating terms of our language to the condition of non-emptiness of arbitrary types in our system. Existential type and refinement by type can be viewed as second and first projections of dependent pairs. Together with existential and universal types, refinement by type allows us to refine types by quantified propositions, providing support for quantified preconditions and postconditions. Some of the type forms we present can be expressed using others; the set of type forms is not minimal.

We define further syntactic sugar, including precise \( \text{if} \; \text{then} \; \text{else} \) and \( \text{match} \) types (Figure 24), as an alternative to the \( \text{if} \; t_1 \; \text{then} \; t_2 \; \text{else} \; t_3 \) that relies on simplifications of Figure 18. We rely on \( \text{Let} \) type of Figure 23, and we use the ill-typed \( \text{err} \) term in the computations of (respectively) \( \text{pred} \), \( \text{unfold_left} \) and \( \text{unfold_right} \), to make sure that the denotations of the \( \text{Let} \) types are empty when \( t \) does not reduce (respectively) to a strictly positive natural number, \( \text{left}(t') \), or \( \text{right}(t') \) for some \( t' \). We proved soundness of typing rules that show that these types behave as expected. For instance, an \( \text{if} \; \text{then} \; \text{else} \) expression can be assigned an \( \text{if} \; \text{then} \; \text{else} \) type of the branches.

These types can be used to encode type level computation reminiscent to one present in Scala HList\(^3\), as illustrated by the following example. Given a list \( L : \text{List}[\text{Bool}] \) and types \( \tau_1 \) and \( \tau_2 \), we define a type \( \text{TypeFromList}(L, \tau_1, \tau_2) \) that represents nested pairs from \( \tau_1 \) and \( \tau_2 \) according to the boolean values (false for \( \tau_1 \) and true for \( \tau_2 \)) and with nesting given by the length of \( L \). For example, when \( L \) is the list \( \text{true, false, true} \) then \( \text{TypeFromList}(L, \text{Nat}, \text{Nat} \rightarrow \text{Nat}) \) represents nested tuples \( w \) of the form \( (a_0, (a_1, (a_2, \ldots))) \) where \( a_0 : \text{Nat}, a_1 : \text{Nat} \rightarrow \text{Nat}, a_2 : \text{Nat} \).

\[ \text{TypeFromList}(L, \tau_1, \tau_2) \triangleq \{ ([w : \tau | \forall n : \{ n : \text{Nat} | n < \text{size}(L) \}]. \text{get}(w, n) := (\text{if nth}(L, n) \; \text{then} \; \tau_1 \; \text{else} \; \tau_2))) \}
\]

The function \( \text{nth}(L, n) \) returns the \( n \)th element of the list \( L \) (defined as a recursive data type). The function \( \text{get}(w, n) \) is also a term-level function that operates on nested pairs \( w \) such that e.g. \( \text{get}(w, 2) \) expands to \( \text{pi}_1(\pi_2(\text{pi}_2(w))) \). The type \( x \equiv \text{get}(w, n) \) in the expansion of refinement by judgement-as-type \( \triangleq \)\( \triangleq \) (Figure 24) does not require type checking the get function; its semantics (see \( \triangleq \) in Figure 13) is that \( \text{get}(w, n) \) reduces to \( x \) in the (untyped) operational semantics.

Furthermore, we prove soundness of multiple rules (such as congruence rules) for establishing the equality judgment \( \Theta; \Gamma \vdash t_1 \equiv t_2 \) described in Section 5. We also prove soundness of a rule that unfolds the definition of a recursive function in the context, which is what is required by the solver in Section 5.3 to unfold the definition of \( \text{merge} \). Such rules are a step towards justifying not only the verification condition generation but also verification condition solving.

\(^3\)https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/hlists.scala
We have implemented our bidirectional type checking procedure by writing an alternative verification-condition generator for Stainless\(^4\) [LARA 2019]. The code is merged into master and available in e.g. release version 0.4.0. Its functionality can be invoked with the \--type-checker\ command line option in both scalac 2.12 and Dotty front end pipeline for Stainless. Thanks to these frontends, which provide a form of type inference, the type annotation burden is lessened for the user. Precise types such as indexed recursive types (which are not supported out of the box by these frontends) still need to be annotated manually.

The implementation was evaluated on benchmarks shown in Figure 25 totalling 14k LoC, collected from existing Stainless test suites and case studies. The proof/code ratio depends on the properties being proven. It can be 0 or close to 0 as in the merge example of Section 2, or higher than 1 when writing lemmas for proving detailed specifications (as in the Huffman coding example). The benchmarks reside in the frontends/benchmarks/typechecker/valid directory of Stainless.

The streams benchmark relies on more expressive annotations of the Dotty compiler that we use to write recursive types, and is available in frontends/benchmarks/dotty-specific/typechecker.

We use the following syntax to represent indexed types:

<table>
<thead>
<tr>
<th>notation in this paper</th>
<th>encoding in Dotty front-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>type (\text{Stream}_n[X])</td>
<td>(\text{Stream}[X] @\text{indexedAt}(n))</td>
</tr>
<tr>
<td>constructor (\text{fold}([\text{Stream}_n[X]])((\text{hd}, \text{tl})))</td>
<td>(\text{indexedAt}(n, \text{Stream}[X](\text{hd}, \text{tl})))</td>
</tr>
</tbody>
</table>

The suite of all benchmarks verifies in \(\sim 6\) minutes in total when using implementation based on our type checker. The table shows the number of lines of codes within each benchmark, and the number of verification conditions which were generated by the type checker and verified by the SMT-backed Inox solver. The time given includes the time for generating the verification conditions and solving them, but not the parsing and compilation which is done by the Scala compiler scalac (which we use for some initial type inference and to obtain a tree representation of the program), nor the transformations which are internal to Stainless and which happen before using the type-checker. To understand the impact of these time measurements, note that 11% of time of total verification time is spent in the bidirectional type-checking algorithm that generates verification conditions and that is the focus of this paper, 66% in checking the verification conditions using a SMT-backed solver Inox for recursive and higher-order functions, 13% for the parsing, name resolution and most of the type checking pipeline of Scala’s scalac compiler, and 10% for extraction of scalac abstract syntax trees to Stainless trees. All times were measured on a Lenovo X1 Carbon laptop with an Intel i7-7500U and 16GB RAM.

To illustrate the diversity of benchmarks, we note that InsertionSort, QuickSorts, MergeSorts and StableSort feature various implementations of the sorting algorithms with the typical properties shown. ListMonad and OptionMonad show that the monadic laws hold for the List\([X]\) and Option\([X]\) types. The List and ListWithSize benchmarks feature a collection of common higher-order functions on lists such as map, filter, forall, etc., as well as many properties shown about the implementations. In the GodelNumbering benchmark, we prove that the pairing function \(2^x(2y+1)−1\) is a bijection between natural numbers and pairs of natural numbers using a series of lemmas about linear and non-linear arithmetic. Similar non-linearity is featured in the MoreExtendedEuclidGCD benchmark where we show that an implementation of the extended Euclid’s algorithm indeed computes the greatest common divisor and the coefficients of Bézout’s identity. Benchmarks also include persistent data structures [Okasaki 1998], ConcTree and ConcRope [Prokopec and Odersky 2015], explicit state model checker ReachabilityChecker, small interpreters, dynamic programming

\(^{4}\)https://github.com/epfl-lara/stainless
### 8 RELATED WORK

In this paper, we tackle program verification using a type-theoretic approach. Other techniques, such as symbolic execution for verification (see e.g. [Hallahan et al. 2019; Nguyen et al. 2017]) or using term rewriting systems (see e.g. [Giesl et al. 2011, 2006, 2004]) can be used instead or in complement to our approach. The use of type checking in verification appears in program verifiers such as F* [Ahman et al. 2017; Swamy et al. 2016, 2013] and Liquid Haskell [Vazou et al. 2013, 2014, 2018]. Like Dafny [Leino 2010] and Stainless, these systems rely on SMT solvers to automatically discharge verification conditions.

F* is a dependently-typed programming language which supports a rich set of effects (such as divergence, mutation, etc.). To the best of our knowledge F* does not support proving termination of functions operating on infinite data structures such as streams. Stainless has a desugaring pass that handles local state and certain forms of unique references. Furthermore, for semantic modeling of global state, a user can use monad syntax (Scala’s `for` comprehensions) explicitly in the surface language, which Scala compiler desugars into higher-order functions that our system can handle. Overall, F* supports a more expressive class of effects than what Stainless currently handles. The goal of Liquid Haskell [Vazou et al. 2018] is to add refinement types to Haskell, a call-by-need language (while we focus here on call-by-value languages). The system supports a sound

---

**Fig. 25.** Summary of evaluation results for our verification tool, featuring lines of code, number of verification conditions (VCs), and type-checking times (including generation of VCs and checking of VCs by Inox).

### algorithms such as Viterbi and Knapsack, and benchmarks solving assignments from Scala MOOCs (see https://courseware.epfl.ch/ or https://www.coursera.org/specializations/scala).
verification procedure which relies on decidable theories of SMT. The supported language imposes certain restrictions on refinement occurrences and the metatheory does not feature recursive types (although they are supported by the implementation). Similarly to our body-visible recursion, Liquid Haskell allows the type checker to access the body of recursive functions (for recursive calls) while type-checking the function itself using a technique called refinement reflection [Vazou et al. 2018]. Dafny verifier [Leino 2010] supports imperative and object-oriented as well as functional programming, including inductive and coinductive types, recursion using ordinals, quantifiers. We are not aware of metatheory to justify the soundness of Dafny. In contrast, we provide a mechanized proof of soundness for System FR, which increases confidence in VC generation soundness. This is important because it is easy to construct paradoxes when combining features such as impredicativity and contravariant recursion. Being based on a reducibility relation, our proof can be used as a basis to add new language features in a compositional way. Several other systems have been recently formalized, such as MetaCoq [Sozeau et al. 2019].

Proof assistants with great expressive power include Isabelle [ Blanchette et al. 2017; Nipkow et al. 2002b], Coq [ Barras 2010; Bertot and Castéran 2004b; Sacchini 2013], Idris [Brady 2013], Agda [Abel 2010; Norell 2007], PML2 [Lepigre 2017], Lean [de Moura 2016] and Zombie [Casinghino et al. 2014]. When coinduction is supported in these systems (which is not always the case), it is typically through special constructs for coinductive types and their corresponding cofixpoint operators, dual to inductive types and their fixpoint operators. In System FR, we instead treat induction and coinduction in a uniform way (see e.g. Section 5.6). Our type system features a single kind of recursive types which allow uniform definition of inductive, coinductive and mixed recursive types. Our operational semantics further rely on the fix operator which roughly corresponds to general recursion in mainstream programming call-by-value languages. An alternative approach to uniform handling of recursion and corecursion is given in [Gianantonio and Miculan 2002]. We were able to encode the first example from this work (in our Streams benchmark) which generates a stream and whose termination proof requires a combination of inductive and coinductive reasoning. Using our system, we expressed that using a simple lexicographic measure combining the index of the type of the produced stream with the inductive measure. In a private communication, Andreas Abel showed us that Agda can also handle this example.

Our system follows Nuprl [ Constable et al. 1986] (and other computational type theories) style of starting out with an untyped calculus and then introducing various types to classify untyped terms based on their behaviors. Nuprl supports a very expressive type system, which covers the types we present in this paper, except impredicative polymorphism. In a large development, Nuprl’s metatheory has been formalized in Coq [Anand and Rahli 2014]. One part of that formalization relies on the use of the Coq axiom of functional choice FunctionalChoice_on, which gives a function $f$ from $A$ to $B$ when a formula of the form $\forall a : A. \exists b : B. \phi(a, b)$ holds. Nuprl does not use SMT solvers for automation, but relies instead on built-in and user-defined tactics similarly to Coq.

Our reducibility definition for recursive type is inspired from step-indexed logical relations [Ahmed 2006]. The main difference is that the indices in step-indexed logical relations do not appear at the level of types, but at the level of the logical relation that gives meaning to types. In System FR we internalize the indices at the level of recursive types in order to give more expressive power to the users, and let them specify decreasing measures for recursive functions that manipulate infinite data structures. This treatment of recursive types is similar to the TORES [Jacob-Rao et al. 2018] type system, where recursive types can be indexed by an arbitrary index language. We only support recursive types indexed by natural numbers. TORES provides a decision procedure for a rich type system with inductive types, coinductive types, and indexed recursive types, yet its metatheory does not handle polymorphism nor refinement types.
Our termination criterion (rule for fix) is inspired by type-based termination [Abel 2004, 2007, 2008, 2012; Abel and Pientka 2013; Barthe et al. 2004]. Such work also typically uses two different kinds of recursion, one for induction and one for coinduction. In type-based termination, instead of requiring that a measure on the arguments of recursive calls decreases, we require that recursive functions are called at a type which is strictly smaller than the type of the caller. Our fix operator produces a term with a forall type, which is similar to the implicit function type of the implicit calculus of constructions [Miquel 2001]. Our termination measure on types can then be understood as a measure on the implicit argument of a function with an implicit function type. Earlier work on type-based termination checking includes simpler type systems such as DML [Xi 2001].

A proof of concept of extension of Dotty compiler by a restricted set of (mostly numerical) refinement predicates was presented in [Schmid and Kunčak 2016], without soundness proof. Developments specific to Scala include dependent object types [Amin 2016] which focus on path dependency instead of predicate refinement and, by design, admit possibly non-terminating programs. System FR does not attempt to support records, let alone path dependency. Stainless relies on Scala compiler front ends to obtain a symbol-resolved syntax tree. For example, implicit parameters in Scala [Odersky et al. 2018] become ordinary explicit parameters by the time Stainless processes them. Conversely, many of the types we define in System FR do not have their counterparts in Scala. An upcoming PhD thesis of Nicolas Voirol [Voirol 2019] presents a system related to System FR that more closely follows the expressive power of Inox solver on which Stainless relies.

9 CONCLUSION

We have presented System FR, a formalized type system and a bidirectional type checking algorithm that can serve as a basis of a verifier for higher-order functional programs. We were able to verify correctness and safety of a wide range of benchmarks amounting to 14k lines of code and proofs. Our formalization suggests that lazy data structures and non-covariant recursion are tractable and that explicit indices required in the general framework can be eliminated in commonly occurring cases. Our type system incorporates Π and Σ types, yet it also supports their variants (∀ and ∃) that correspond to infinite intersections and unions. Along with support for singleton types and refinements, we obtained a rich framework to approximate program semantics, which can also help further type, invariant, and measure inference algorithms. Our experience confirms an advantage of the semantic-based soundness proof: once we adopt an approach of interpreting types as sets of terms, we are less dependent on a particular choice of syntactic rules; we can introduce new classes of types and new rules for verification condition generation and solving, as long as we can justify them semantically. When viewing types in our system as propositions, we obtain an expressive quantified logic. We have proven the soundness of this logic in Coq using countable interpretation of ground System FR types. The success in verification of our benchmarks suggests that the rules of System FR work well for many properties of functional programs. Furthermore, refinement by type along with intersections and unions (Section 6.1) allows System FR to describe types whose interpretations are undecidable countable sets from higher levels of arithmetical hierarchy.

ACKNOWLEDGMENTS

Work supported in part by Swiss National Science Foundation project 200021_175676 "Scaling Predicate Types". We thank Ravichandran Kandhadai Madhavan for numerous discussions about termination for higher-order functional programs; Romain Ruetschi for his valuable contributions to Stainless implementation; Paolo G. Giarrusso for interesting discussions and very relevant pointers on type theory; Andreas Abel for explaining us how to mix induction and co-induction in Agda. Thanks to Yoan Géran who implemented another prototype of our algorithm, found formalization discrepancies and proposed improvements to the typing rules.
REFERENCES


