Pipit on the Post: Proving Pre- and Post-conditions of Reactive Systems

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Reactive languages such as Lustre and Scade are used to implement safety-critical control systems; proving such programs correct and having the proved properties apply to the compiled code is therefore equally critical. We introduce Pipit, a small reactive language embedded in F*, designed for verifying control systems and executing them in real-time. Pipit includes a verified translation to transition systems; by reusing F*'s existing proof automation, certain safety properties can be automatically proved by k-induction on the transition system. Pipit can also generate executable code in a subset of F* which is suitable for compilation and real-time execution on embedded devices. The executable code is deterministic and total and preserves the semantics of the original program.

2012 ACM Subject Classification Computer systems organization → Real-time languages; Theory of computation → Program verification; Software and its engineering → Specialized application languages

Keywords and phrases Lustre, streaming, reactive, verification

1 Introduction

Safety-critical control systems, such as the anti-lock braking systems that are present in most cars today, need to be correct and execute in real-time. One approach, favoured by parts of the aerospace industry, is to implement the controllers in a high-level language such as Lustre [**?**] or Scade [**?**], and verify that the implementations satisfy the high-level specification using a model-checker, such as Kind2 [**?**]. These model-checkers can prove many interesting safety properties automatically, but do not provide many options for manual proofs when the automated proof techniques fail. Additionally, the semantics used by the model-checker may not match the semantics of the compiled code, in which case properties proved do not necessarily hold on the real system. This mismatch may occur even when the compiler has been verified to be correct, as in the case of Vélus [**?**]. For example, in Vélus, integer division rounds towards zero, matching the semantics of C; however, integer division in Kind2 rounds to negative infinity, matching SMT-lib [**?**, **?**].

To be confident that our proofs hold on the real system, we need a single ~~semantics that is shared between~~ shared semantics for the compiler and the ~~model-checker or~~ prover. In this paper we introduce Pipit[1], an embedded domain-specific language for implementing and verifying controllers in F*. Pipit aims to provide a high-level language based on Lustre, while reusing F*'s proof automation and manual proofs for verifying controllers [**?**], and using Low*'s C-code generation for real-time execution [**?**]. To verify programs, Pipit translates its expression language to a transition system for k-inductive proofs, which is verified to be an abstraction of the original semantics. To execute programs, Pipit can generate executable code, which is total and semantics-preserving.

In this paper, we make the following contributions:

[1] Implementation available at (anonymised)

- we motivate the need to combine manual and automated proofs of reactive systems with a strong specification language (Section 2);
- we introduce Pipit, a minimal reactive language that supports rely-guarantee contracts and properties; crucially, proof obligations are annotated with a status — *valid* or *deferred* — allowing proofs to be delayed until more is known of the program context (Section 3);
- we describe a *checked semantics* for Pipit~~, which is parameterised by the property status~~; after checking deferred properties, programs ~~can be~~ are *blessed*, ~~and their properties lifted to valid status~~ which marks their properties as valid (Subsection 3.2);
- we describe an encoding of transition systems that can express under-specified rely-guarantee contracts as functions rather than relations; composing functions results in simpler transition systems (**??**);
- we identify the invariants and lemmas required to prove that the abstract transition system is an abstraction of the original semantics (**??**, **??**);
- similarly, we offer a mechanised proof that the executable transition system preserves the original semantics (**??**);
- finally, we evaluate Pipit by implementing the high-level logic of a ~~time-triggered~~ Time-Triggered Controller Area Network (~~CAN~~TTCAN) bus driver and verifying an abstract model of a key ~~operation~~ component (**??**).

2 Pipit for time-triggered networks

To introduce Pipit, we consider a ~~driver with~~ *time-triggered* network driver, which has a static schedule ~~of *triggers*, or actions tobe performed at a particular time; this~~ dictating the network traffic, and which all nodes on the network must adhere to. This driver is a simplification of the ~~time-triggered~~ Time-Triggered Controller Area Network (~~CAN~~TTCAN) bus specification [**?**] which we will discuss further in **??**.

2.1 ~~Deferring and proving properties~~

~~The schedule of our time-triggered driver is determined by a constant array of triggers,~~ At a high level, the network schedule is described by a *system matrix* which consists of rows of *basic cycles*. Each basic cycle consists of a sequence of actions to be performed at specific time-marks. Actions in the schedule may not be relevant to all nodes; the node's *node matrix* contains only the relevant actions. The node matrix is represented in memory by a *triggers array* containing triggers sorted by their ~~associated~~ time-marks; trigger actions include sending and receiving application-specific messages, sending reference messages, and triggering 'watch' alerts. Reference messages start a new basic cycle; a subset of nodes, designated as leaders, send reference messages to synchronise the network. Watch alerts are generally placed after an expected reference message to signal an error if no reference message is received.

Figure 1 (left) shows an example node matrix for a non-leader node. The matrix consists of two basic cycles C0 and C1 with messages sent at time-marks 0, 1 and 2. The node expects to receive a reference message at time-mark ~~. The driver~~ 7; the watch at time-mark 9 allows a grace period before triggering an error if the reference message is not received. Figure 1 (right) shows the corresponding triggers array.

The network has strict timing requirements which prohibit the driver from looping through the entire triggers array at each time-mark. Instead, the driver maintains an index that refers to the current trigger. At each ~~instant in time~~time-mark, the driver checks if the current trigger has expired or is inactive, and if so, it increments the index. ~~We first~~

	TM0	TM1	TM2	···	TM9
C0	SEND A	SEND B	-	···	WATCH
C1	SEND A	-	SEND C	···	WATCH

```
0:{ time = 0; enabled = {C0,C1}; action = SEND(A); }
1:{ time = 1; enabled = {C0};    action = SEND(B); }
2:{ time = 2; enabled = {C1};    action = SEND(C); }
3:{ time = 9; enabled = {C0,C1}; action = WATCH;   }
```

Figure 1 Left: node matrix; right: corresponding triggers array configuration

2.1 Deferring and proving properties

We implement a streaming function *count_when* to maintain the index into the triggers array; the function takes a constant natural number *max* and a stream of booleans *inc*. At each ~~time~~ step, *count_when* checks whether the current increment flag is true; if so, it increments the previous counter, saturating at the maximum; otherwise, it leaves the ~~previous~~ counter as-is.

```
let count_when (max: ℕ) (inc: stream 𝔹): stream ℕ =
  rec count.
     check[?] (0 ≤ count ≤ max);
     let count' = (0 fby count) + (if inc then 1 else 0) in
     if  count' ≥ max then max else count'
```

The implementation of *count_when* first defines a recursive stream, *count*, which states an invariant about the count before defining the incremented stream *count'*. Inside *count'*, the syntax 0 `fby` *count* is read as "the initial value of zero *followed by* the previous count".

The syntax `check`$_{?}$ $(0 \leq count \leq max)$ asserts that the count is within the range $[0, max]$. The subscript ? on the check is the *property status*, which in this case denotes that the assertion has been stated, but it is not yet known whether it holds. A property status of ☑, on the other hand, denotes that a property has been proved to hold. These property statuses are used to defer checking properties until enough is known about the environment, and to avoid rechecking properties that have already been proven. In practice, the user does not explicitly specify property statuses in the source language. The stated property $(0 \leq count \leq max)$ is a stream of booleans which must always be true. Non-streaming operations such as $\leq$ are implicitly lifted to streaming operations, and non-streaming values such as 0 and *max* are implicitly lifted to constant streams.

We defer the proof of the property here because, at the point of stating the property inside the `rec` combinator, we don't yet have a concrete definition for the count variable. In this case, we could have instead deferred the *statement* of the property by introducing a let-binding for the recursive count and putting the `check` outside of the `rec` combinator. However, it is not always possible to defer property statements: for example, when calling other streaming functions that have their own preconditions, it may not be possible to move the function call outside of its enclosing `rec`.

Pipit is an embedded domain-specific language. The program above is really syntactic sugar for an F* program that takes a natural number and constructs a Pipit core expression with a free boolean variable. We will discuss the details of the core language in Section 3, but for now we focus on the source program with some minor embedding details omitted.

To actually prove the property above, we use the meta-language F*'s tactics to translate the program into a transition system and prove the property inductively on the system. Finally, we *bless* the expression, which marks the properties as valid ($[? := ☑]$). Blessing is an intensional operation that traverses the expression and updates the internal metadata, but does not affect the runtime semantics.

```
let count_when☑ (max: ℕ): stream 𝔹 → stream ℕ =
  let system = System.translate1(count_when max) in
  assert (System.inductive_check system) by (pipit_simplify ());
  bless1 (count_when max)
```

The subscript 1 in the translation to transition system and blessing operations refers to the fact that the stream function has one stream parameter. The *pipit_simplify* tactic in the assertion performs normalisation-by-evaluation to simplify away the translation to a first-order transition system; F*'s proof-by-SMT can then solve the inductive check directly.

Callers of *count_when* can now use the validated variant without needing to re-prove the count-range property. In a dedicated model-checker such as Kind2 [?] or Lesar [?], this kind of bookkeeping would all be performed under-the-hood. By embedding Pipit in a general-purpose theorem prover, we move some of the bookkeeping burden onto the user; however, we have increased confidence that the compiled code matches the verified code and, as we shall see, we also have access to a rich specification language.

2.2 ~~The time-triggered system matrix~~Restrictions on the triggers array

~~The schedule of the time-triggered network is abstractly described by a *system matrix*, consisting of rows of *basic cycles*, columns of *transmission columns*, and cells of optional messages. Each basic cycle is identified by its cycle index and each transmission column has an associated time-mark.~~

~~(left) shows an example system matrix with cycles and and transmission columns at time-marks 0~~Our driver may fall behind when trying to execute certain schedules, as the driver only processes one trigger per time-mark. To ensure that the schedule can be executed on time, ~~1 and 2. For this example, we assume that one message can be sent per clock cycle. To execute this system matrix, we synchronise the local time to zero at the start of basic cycle . After a basic cycle completes, the nodes on the network synchronise before execution continues to the next basic cycle.~~

~~(right) shows the corresponding configuration for the triggers array. The enabled set denotes the basic cycles for which a trigger is active.~~

~~The system has strict timing requirements which restrict how triggers can be defined. In this example, each trigger has a unique time; in general, trigger times can overlap, but they need to be enabled on distinct cycles. Additionally, the schedule~~ the triggers array must allow sufficient time for the driver to skip over ~~the disabled triggers . Concretely, we~~ any disabled triggers before the next enabled trigger starts.

Recall our concrete triggers array from Figure 1, which contained trigger 1 (SEND B at time-mark 1 on cycle C0), and trigger 2 (SEND C at time-mark 2 on cycle C1). We could postpone trigger 1 to send ~~message~~ B at time-mark 2, as ~~triggers 1 and 2 have distinct cycles~~the corresponding cell in the node matrix is empty. However, we ~~could not bring forward trigger~~ *cannot* bring the trigger at index 2 forward to send message C at time-mark 1~~: the driver can only process one trigger per tick, and~~ , as it takes two steps to reach trigger 2 from the start of the array.

~~TM0 TM1 TM2 C0 MSG A MSG B - C1 MSG A - MSG C~~

```
~~0: { time_mark = 0; enabled = {C0,C1}; msg = A; }~~
~~1: { time_mark = 1; enabled = {C0};     msg = B; }~~
~~2: { time_mark = 2; enabled = {C1};     msg = C; }~~
```

~~Left: system matrix; right: corresponding triggers array configuration~~

We impose three restrictions on ~~the triggers array~~*valid* triggers arrays: the time-marks must be sorted; there must be an adequate time-gap between any two triggers that are enabled on the same cycle index; and each trigger's time-mark must be greater-than-or-equal to its index, so that it is reachable in time from the start of the array.

With these restrictions in place, we prove a lemma *lemma_can_reach_next*, which states that for all valid cycle indices and trigger indices, if the current trigger is enabled in the current cycle and there is another enabled trigger scheduled to occur somewhere in the array after the current one, then there is an adequate time-gap to allow the driver to skip over any disabled triggers in-between. These properties are straightforward in a theorem prover, but are difficult to state in a model-checker with a limited specification language.

2.3 Instantiating lemmas and defining contracts

We can now implement the trigger-fetch logic, which keeps track of the current trigger. ~~The trigger-fetch logic uses~~ We use the *count_when* streaming function to define the index of the current trigger; we tell *count_when* to increment the index whenever the previous index has expired or is inactive in the current basic cycle. We simplify our presentation here and only consider a ~~single cyclein isolation~~constant cycle: the real system presented in **??** has some extra complexity such as resetting the index, incrementing the cycle index at the start of a new cycle, and using machine integers.

```
let trigger_fetch (cycle: ℕ) (time: stream ℕ): stream ℕ =
  rec index.
    let inc = false fby ((time_mark index) ≤ time ∨ ¬(enabled index cycle)) in
    let index = count_when☑ trigger_count inc in
    ~~pose1 (lemma_can_reach_next cycle) index~~pose (lemma_can_reach_next cycle index);
    check? (can_reach_next_active cycle time index);
    index
```

The *trigger_fetch* function takes a static cycle index and a stream denoting the current time. The increment flag and the index are mutually dependent — the increment flag depends on the previous value of the index, while the index depends on the current value of the increment flag — so we introduce a recursive stream for the index. We allow the index to go one past the end of the array to denote that there are no more triggers.

We use the ~~*pose*$_1$~~ *pose* helper function to lift the *lemma_can_reach_next* lemma to a streaming context and instantiate it~~; the subscript 1 indicates that the lemma is being applied to one streaming argument (the index)~~. We then state an invariant as a deferred property. Informally, the invariant states that, either the current active trigger is not late, or the next active trigger after the current index is in the future and we can reach it in time.

With the explicitly instantiated lemma, we can prove the streaming invariant by straightforward induction on the transition system. To help compose this function with the rest of the system, we also abstract over the details of the trigger-fetch mechanism by introducing a rely-guarantee contract for *trigger_fetch*. The contract we state is that if ~~the environment ensures that the time doesn't skip — that is,~~ we are called once per ~~microsecond —~~ time-mark then we guarantee that we never encounter a late trigger.

```
let trigger_fetch☑ (cycle: ℕ): stream ℕ → stream ℕ =
  let contract = Contract.contract_of_stream1 {
    rely = ~~(λtime. time_no_skips time);~~ (λtime. time = 0 fby (time + 1));
    guar = (λtime index. (index_valid index ∧ enabled index cycle)
```

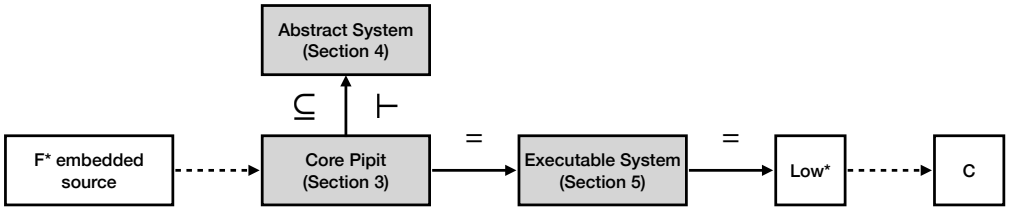

Figure 2 Architecture of Pipit. The gray boxes and solid arrows are defined in this paper. The white boxes and dashed arrows are trusted components. The labels ~~next to the arrows~~ denote verified properties of the translation: abstraction (⊆), entailment of proof obligations (⊢), and equivalence (=).

$$\implies (\text{time\_mark } \mathit{index}) \geq \mathit{time});$$
$$\texttt{body} = (\lambda \mathit{time}.\ \text{trigger\_fetch } \mathit{cycle}\ \mathit{time});$$
$$\} \texttt{ in}$$
$$\texttt{assert } (\text{Contract.inductive\_check } \mathit{contract}) \texttt{ by } (\text{pipit\_simplify } ());$$
$$\text{Contract.stream\_of\_contract}_1\ \mathit{contract}$$

In the implementation of the validated variant of *trigger_fetch*, we first construct the contract from streaming functions. The Contract.contract_of_stream$_1$ combinator describes a contract with one input (the time stream), and takes stream transformers for each of the rely, guarantee and body. The combinator transforms the surface syntax into core expressions. The assertion (Contract.inductive_check *contract*) then translates the expressions into a transition system, and checks that if the rely always holds then the guarantee always holds, and that the as-yet-unchecked subproperties hold. Finally, Contract.stream_of_contract$_1$ blesses the core expression and converts it back to a stream transformer, so it can be easily used by other parts of the program.

~~When this function is~~ The key distinction between our streaming rely-guarantee contracts and imperative pre-post contracts is that the rely and guarantee are both *streams* of booleans, rather than instantaneous predicates. In this case, the rely ($\mathit{time} = 0$ `fby` ($\mathit{time} + 1$)) checks that the current time is exactly one time-mark after the time at the previous *tick* of computation. Expressing such a rely in an imperative setting would require extra encoding, as preconditions in imperative languages do not generally have an innate notion of the previous value with respect to a global shared clock.

When *trigger_fetch* is used in other parts of the program, the caller must ensure that the environment satisfies the rely clause. In the core language, this is tracked by another deferred property status attached to the contract; we will discuss this further in Section 3.

3 ~~Core~~ Pipit core language

We now introduce the core Pipit language. Note that this form differs slightly from the surface syntax presented earlier in Section 2, which used the syntax of the metalanguage F*, as well as including proofs in F* itself.

Figure 2 shows the high-level architecture of Pipit. On the left-hand-side, the surface syntax embedded in F* is shown; this includes some Pipit-specific syntactic sugar. The translation from the surface syntax to the core language is trusted. There are two targets from the core language: abstract transition systems for verification, and executable transition systems for extraction to C. The translation to abstract systems is verified to be an abstraction

$$
\begin{array}{lcll}
e, e' & := & v \mid x \mid p(\overline{e}) & \text{(values, variables and operations)} \\
& \mid & v \ \texttt{fby}\ e \mid \texttt{rec}\ x.\ e[x] & \text{(delayed and recursive streams)} \\
& \mid & \texttt{let}\ x = e\ \texttt{in}\ e'[x] & \text{(let-expressions)} \\
& \mid & \texttt{check}_{\pi}\ e_{\text{prop}} & \text{(checked properties)} \\
& \mid & \texttt{contract}_{\pi}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} & \text{(rely-guarantee contracts)} \\
\\
v & := & n \in \mathbb{N} \mid b \in \mathbb{B} \mid r \in \mathbb{R} \mid \ldots & \text{(values)} \\
p & := & (+) \mid (-) \mid (\times) \mid \texttt{if-then-else} \mid \ldots & \text{(primitives)} \\
\\
\pi & := & \boxtimes \mid \boxed{?} & \text{(property statuses: valid or unknown)} \\
\\
V & := & \cdot \mid V; v & \text{(streams of values)} \\
\sigma & := & \{\overline{x \mapsto v}\} & \text{(heaps)} \\
\Sigma & := & \cdot \mid \Sigma; \sigma & \text{(streaming history environments)} \\
\tau, \tau' & := & \mathbb{N} \mid \mathbb{B} \mid \tau \times \tau \mid \ldots & \text{(value types)} \\
\Gamma & := & \cdot \mid x : \tau, \Gamma & \text{(type environments)}
\end{array}
$$

Figure 3 ~~Pipit core language~~ Core grammar~~, which contains~~: expressions e, values v, primitive operations p, and property statuses π.

according to the dynamic semantics (Subsection 3.1). ~~One property that still remains to be proven is that the proof obligations~~The translation to abstract systems also generates proof obligations, which are verified to correspond to the proof obligations on the ~~abstract transition system entail the original proof obligations (); this pending proof is denoted as negated entailment in the figure (⊬)~~original program. The translation to executable transition ~~system~~ systems is proven to be ~~semantics preserving~~semantics-preserving, as is the subsequent translation to Low*. The translation from Low* to C is external to this paper and forms part of our trusted computing base.

Figure 3 defines the grammar of Pipit. The expression form e includes standard syntax for values (v), variables (x) and primitive applications ($p(\overline{e})$). Most of the expression forms were introduced informally in Section 2 and correspond to the clock-free expressions of Lustre [**?**].

The expression syntax for delayed streams (v `fby` e) denotes the previous value of the stream e, with an initial value of v when there is no previous value.

Recursive streams ~~, which can refer to previous values of the stream itself,~~ are defined using the fixpoint operator (`rec` $x.\ e[x]$); the syntax $e[x]$ means that the variable x can occur in e. As in Lustre, recursive streams can only refer to their previous values and must be *guarded* by a delay: the stream (`rec` $x.\ 0$ `fby` $(x + 1)$) is well-defined ~~, but~~ and counts from zero up, but the stream (`rec` $x.\ x + 1$) is invalid and has no computational interpretation. This form of recursion differs slightly from standard Lustre, which uses a set of mutually-recursive bindings. Although we cannot express mutually-recursive bindings in the core syntax here, we can express them as a notation on the surface syntax by combining the bindings together into a record or tuple.

Checked properties and contracts are annotated with their property status π, which can either be valid ($\boxtimes$) or unknown ($\boxed{?}$). For checked properies $\texttt{check}_{\pi}\ e$, the property status denotes whether the property has been proved to be valid.

Contracts $\texttt{contract}_{\pi}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$ allow modular reasoning by replacing the implementation with an abstract specification. Contracts involve two verification conditions.

Firstly, when a contract is *defined*, the definer must prove that the body satisfies the contract: roughly, if e_{rely} is always true, then $e_{\text{guar}}[x := e_{\text{body}}]$ is always true. Secondly, when a contract is *instantiated*, the caller must prove that the environment satisfies the precondition: that is, e_{rely} is always true. Conceptually, then, a contract could have two property statuses: one for the definition and one for the instantiation. However, in practice, it is not useful to defer the proof of a contract definition — one could achieve a similar effect by replacing the contract with its implementation. For this reason, we only annotate contracts with one property status, which denotes whether the instantiation has been proved to satisfy the precondition.

For ~~the~~ example, the core expression ($\texttt{rec}\ \mathit{sum}.\ (0\ \texttt{fby}\ \mathit{sum}) + \mathit{ints}$) computes the sum of values from a stream of integers *ints* by defining a recursive stream *sum*, which is delayed and given an initial value of zero. If we were to use this sum in a context that required a strictly positive integer, we could give it a contract that states that if the input stream is always positive, then the resulting sum is also positive:

$$\texttt{contract}_{\boxed{?}}\ \{\mathit{ints} > 0\}\ (\texttt{rec}\ \mathit{sum}.\ (0\ \texttt{fby}\ \mathit{sum}) + \mathit{ints})\ \{\mathit{sum}.\ \mathit{sum} > 0\}$$

To be considered a valid program, we must prove that the contract definition itself holds, as with our earlier contract (Subsection 2.3). The unknown property status here allows us to defer the caller's proof that the input stream is always positive until the contract is used.

The remaining grammatical constructs of Figure 3 describe streams, value environments, types and type environments. Streams V are represented as a sequence of values; streaming history environments Σ are streams of heaps. Types τ and type environments Γ are standard. For the presentation of the formal grammar here, we consider only a fixed set of values and primitives; in practice, the implementation is parameterised by a primitive table which we extend with immutable array operations for the TTCAN driver logic in **??**.

~~Streams V are represented as a sequence of values; streaming history environments Σ are streams of heaps. Types τ and type environments Γ are standard.~~

We define the typing judgments for Pipit in Figure 4. Most of the typing rules are standard for an unclocked Lustre. The typing judgment $\Gamma \vdash e : \tau$ denotes that, in an environment of streams Γ, expression e denotes a stream of type τ. This core typing judgment differs from the surface syntax used in Section 2, which used an explicit stream type; for the core language, we instead assume that everything is a stream.

~~For values, we~~ We use an auxiliary ~~definition~~ function prim-value-type$(v) = \tau$ to denote that value v has type τ. ~~Likewise, for primitives we use~~ ; for primitives prim-type$(p) = (\tau_1 \times \cdots \times \tau_n) \to \tau'$ ~~to denote that primitive~~ denotes that p takes arguments of type τ_i and returns a result of type τ'. Primitives are pure, non-streaming functions.

Rules TVALUE, TVAR, TPRIM and TLET are standard.

Rule TFBY states that expression v `fby` e requires both v and e to have equal types~~; the result is the same type~~.

Rule TREC states that a recursive stream `rec` $x.\ e$ has the recursive stream bound inside e. The recursion must also be guarded, in that any recursive references to x are delayed, but this requirement is performed as a separate syntactic check described in **??**.

Rule TCHECK states that ~~statically checking a property~~ checked properties $\texttt{check}_\pi\ e$ ~~requires~~ require a boolean property e~~and returns unit~~.

Finally, rule TCONTRACT applies for a contract $\texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$ with a body expression of ~~some~~ type τ. The overall expression has result type τ. Both rely and guarantee ~~clauses~~ must be boolean expressions~~. Additionally, the guarantee clause~~ , and the guarantee can refer to the result ~~value~~ as x.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\text{prim-value-type}(v) = \tau}{\Gamma \vdash v : \tau}\ (\textsc{TValue}) \qquad \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}\ (\textsc{TVar})$$

$$\frac{\text{prim-type}(p) = (\tau_1 \times \cdots \times \tau_n) \to \tau' \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash p(\overline{e}) : \tau'}\ (\textsc{TPrim})$$

$$\frac{\text{prim-value-type}(v) = \tau \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash v\ \texttt{fby}\ e' : \tau}\ (\textsc{TFby}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}\ x.\ e[x] : \tau}\ (\textsc{TRec})$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e'[x] : \tau'}\ (\textsc{TLet}) \qquad \frac{\Gamma \vdash e : \mathbb{B}}{\Gamma \vdash \texttt{check}_\pi\ e : \texttt{unit}}\ (\textsc{TCheck})$$

$$\frac{\Gamma \vdash e_{\text{rely}} : \mathbb{B} \qquad \Gamma \vdash e_{\text{body}} : \tau \qquad \Gamma, x : \tau \vdash e_{\text{guar}} : \mathbb{B}}{\Gamma \vdash \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} : \tau}\ (\textsc{TContract})$$

Figure 4 Typing rules for Pipit; the judgment $\Gamma \vdash e : \tau$ denotes that expression e describes a *stream* of values of type τ. ~~Two auxiliary~~ Auxiliary functions are used for values and primitive operations~~; their definitions are standard and are omitted~~.

3.1 Dynamic semantics

The dynamic semantics of Pipit are defined in Figure 5. We present our semantics in a big-step form. This differs somewhat from traditional *reactive* semantics of Lustre [**?**]. Our big-step semantics emphasises the equational nature of Pipit, as it is substitution-based and syntax-directed, while the reactive semantics emphasises the finite-state streaming execution of the system. We use transition systems for reasoning about the finite-state execution (**??**), which is fairly standard [**?**, **?**, **?**]. Previous work on the W-CALCULUS [**?**] for linear digital-signal-processing filters makes a similar distinction and provides a non-streaming semantics for reasoning about programs and a streaming semantics for executing programs.

The judgment form $\Sigma \vdash e \Downarrow v$ denotes that expression e evaluates to value v under streaming history Σ. The streaming history is a stream of heaps; in practice, we only evaluate expressions with a non-empty streaming history.

~~Rule Valuestates that evaluating a value results in the value itself.~~ At a high level, evaluation unfolds recursive streams to determine a value. For example, to evaluate the earlier sum example with input $ints = [1; 2]$, we start with the judgment:

$$\{ints \mapsto 1\}; \{ints \mapsto 2\} \vdash (\texttt{rec}\ sum.\ (0\ \texttt{fby}\ sum) + ints) \Downarrow v$$

~~Rule~~ First, we unfold the recursive stream one step to get $(0\ \texttt{fby}\ (\texttt{rec}\ sum.\ (0\ \texttt{fby}\ sum) + ints)) + ints$. Evaluation of primitives is standard. To evaluate variables, we look for the variable in the current (rightmost) heap:

$$\frac{}{\{ints \mapsto 1\}; \{ints \mapsto 2\} \vdash ints \Downarrow 2}\ (\textsc{Var})$$

$$\boxed{\Sigma \vdash e \Downarrow v}$$

$$\frac{}{\Sigma; \sigma \vdash x \Downarrow \sigma(x)}\ (\textsc{Var}) \qquad \frac{}{\Sigma \vdash v \Downarrow v}\ (\textsc{Value}) \qquad \frac{\Sigma \vdash e'[x := e] \Downarrow v}{\Sigma \vdash \texttt{let}\ x = e\ \texttt{in}\ e'[x] \Downarrow v}\ (\textsc{Let})$$

$$\frac{\Sigma \vdash e_1 \Downarrow v_1 \quad \ldots \quad \Sigma \vdash e_n \Downarrow v_n}{\Sigma \vdash p(\overline{e}) \Downarrow \text{prim-sem}(p, \overline{v})}\ (\textsc{Prim})$$

$$\frac{}{\sigma \vdash v\ \texttt{fby}\ e' \Downarrow v}\ (\textsc{FbyS}_1) \qquad \frac{\text{length}(\Sigma) > 0 \qquad \Sigma \vdash e' \Downarrow v'}{\Sigma; \sigma \vdash v\ \texttt{fby}\ e' \Downarrow v'}\ (\textsc{FbyS}_S)$$

$$\frac{\Sigma \vdash e[x := \texttt{rec}\ x.\ e] \Downarrow v}{\Sigma \vdash \texttt{rec}\ x.\ e[x] \Downarrow v}\ (\textsc{Rec}) \qquad \frac{}{\Sigma \vdash \texttt{check}_\pi\ e \Downarrow ()}\ (\textsc{Check})$$

$$\frac{\Sigma \vdash e_{\text{body}} \Downarrow v}{\Sigma \vdash \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} \Downarrow v}\ (\textsc{Contract})$$

$$\boxed{\Sigma \vdash e \Downarrow^* V} \qquad \boxed{\Sigma \vdash e \Downarrow^\square \top}$$

$$\frac{}{\cdot \vdash e \Downarrow^* \cdot}\ (\textsc{Steps}_0) \qquad \frac{\Sigma \vdash e \Downarrow V \qquad \Sigma; \sigma \vdash e \Downarrow v}{\Sigma; \sigma \vdash e \Downarrow V; v}\ (\textsc{Steps}_S)$$

$$\frac{\Sigma \vdash e \Downarrow^* \top; \ldots}{\Sigma \vdash e \Downarrow^\square \top}\ (\textsc{Always})$$

Figure 5 Dynamic semantics for Pipit; the judgment form $\Sigma \vdash e \Downarrow v$ denotes that evaluating expression e under streaming history Σ results in value v.

For delays, we discard the current heap and continue evaluation with the history prefix:

$$\frac{\{ints \mapsto 1\} \vdash (\texttt{rec}\ sum.\ (0\ \texttt{fby}\ sum) + ints) \Downarrow 1}{\{ints \mapsto 1\}; \{ints \mapsto 2\} \vdash 0\ \texttt{fby}\ (\texttt{rec}\ sum.\ (0\ \texttt{fby}\ sum) + ints) \Downarrow 1}\ (\textsc{FbyS}_S)$$

Returning to Figure 5, rule Var ~~states that to evalute~~ evalutes a variable x under some non-empty stream history $\Sigma; \sigma$, where σ is the most recent heap~~, we look up the variable in σ.~~

. Rules Value and Let are standard. Rule Prim ~~states that to evaluate~~ evaluates a primitive p applied to many arguments e_1 to e_n ~~, we evaluate~~ by evaluating each argument separately; we then apply the primitive with prim-sem metafunction.

~~Rule FbyS$_1$ evaluates a followed-by expression when the streaming history contains only a single element. Here,~~ For delay expressions v fby e~~evaluates to v, as there is no previous value~~of e to use.

~~Rule FbyS$_S$ evaluates a followed-by expression when~~ , we have two cases depending on whether there is a previous value. When there is no previous value – the streaming history ~~contains multiple entries. In this case, v fby e evaluates~~ only contains the current heap –

rule FBY$_1$ evaluates to the default value v. Otherwise, rule FBY$_S$ applies; we evaluate the previous value of e by discarding the most recent entry from the streaming history.

Rule REC evaluates a recursive stream `rec` x. e by unfolding the recursion one step. For causal expressions (**??**), where each recursive occurrence of x is guarded by a followed-by, this unfolding eventually terminates as each followed-by shortens the history.

Rule ~~LET IS STANDARD.~~

~~RULE~~ CHECK ~~states that check expressionsalways evaluate to unit~~ignores the property when evaluating check expressions. We do not ~~perform a dynamic check that the property is true here; checking the truth of properties is dealt with~~ dynamically check the property here; this is done in the checked semantics (Subsection 3.2).

~~Rule~~ Similarly, rule CONTRACT ~~states that contracts evaluate by just evaluating their body. Like with checks, we do not perform a dynamic check that the precondition andpostcondition hold.~~ ignores preconditions and postconditions when evaluating contracts. From an abstraction perspective, it would be valid to return an arbitrary value that satisfies the contract. However, such an abstraction would make evaluation non-deterministic and, for contracts with unsatisfiable postconditions, non-total. The deterministic and total nature of evaluation is key to our proofs and metatheory.

We also define two auxiliary judgment forms: $\Sigma \vdash e \Downarrow^* V$ and $\Sigma \vdash e \Downarrow^\square \top$.

Judgment form $\Sigma \vdash e \Downarrow^* V$ denotes that, under history Σ, expression e evaluates to the *stream* V. This judgment performs iterated application of single-value evaluation.

Judgment form $\Sigma \vdash e \Downarrow^\square \top$ denotes that a boolean expression e evaluates to the stream of trues under history Σ. Informally, it can be read as "e is always true in history Σ".

3.2 Checked semantics

~~(CHKCONTRACT)~~

~~Checked semantics for Pipit; the judgment form $\Sigma \vdash_\pi e$ valid denotes that evaluating expression e under streaming history Σ satisfies the checks and rely-guarantee contract requirements that are labelled with property status π.~~

In addition to the big-step semantics above, we also define a judgment form for checking that the properties and contracts of a program hold for a particular streaming history. We call these the *checked* semantics~~. Unlike an axiomatic semantics, the checked semantics operate on a concrete set of input streams.~~ ; they are comparable to checking runtime assertions.

The checked semantics have the judgment form $\Sigma \vdash_\pi e$ valid, which denotes that under streaming history Σ, the properties and contracts of e with status π hold. The property status dictates which properties should be checked and which should be ignored.

~~To show that an expression e's unknown properties hold , we prove that for all streaming histories~~ We consider a program to be *valid* if its checks hold for all histories ($\forall \Sigma.\ \Sigma \vdash_{\boxed{\checkmark}} e$ valid). The checked semantics are a specification describing what it means to be a valid program. We do not generally verify programs directly using the checked semantics; instead, we translate to an abstract transition system and construct the proofs there (**??**).

To check a property (`check`$_\pi$ e) in history Σ~~, assuming the valid properties hold ($\Sigma \vdash_{\boxed{\checkmark}} e$ valid), then the unknown properties ($\Sigma \vdash_{\boxed{?}} e$ valid)hold. The assumption here means that we do not have to re-check properties after proving them once.~~ , we check that e is always true ($\Sigma \vdash e \Downarrow^\square \top$).

~~Contracts involve two proofs~~Checking contracts is more involved. For whole-program correctness, it would suffice to check that a contract's rely and guarantee both hold. However, the purpose of contracts is to enable modular reasoning about parts of the program: we need to be able to check contracts independently of their context. Conceptually, then, contracts

involve two kinds of checks: one for the definition and one for the ~~instantiation. To prove that~~ call-site. To check a contract definition~~$\texttt{contract}_\pi\ \{c_{\text{rely}}\}\ c_{\text{body}}\ \{x.\ c_{\text{guar}}[x]\}$ is valid, we show that for all streaming histories Σ, assuming the rely is always true under the history $(\Sigma \vdash c_{\text{rely}} \Downarrow^{\square} \top)$, then the body always~~ , we check that the body satisfies the guarantee ~~$(\Sigma \vdash c_{\text{guar}}[x := c_{\text{body}}] \Downarrow^{\square} \top)$. Additionally, we can also assume that the valid properties in all three components hold, and we must also show that the unknown properties are valid. The fact that the checked semantics refers to a particular Σ is significant here: it allows the proof of contract validity to only consider streaming histories where the rely actually holds~~for all *valid* contexts – that is, those where the rely holds. Then, to check a contract instance, we just need to check that the call-site satisfies the rely.

~~To prove that a contract *instantiation* (a call-site) is valid, we show that, under the calling environment, the rely clause is always true . Crucially, the proof can also use the fact that, if the rely~~ For example, recall our earlier contract that the sum of strictly positive integers is positive:

$$\texttt{let}\ \text{sum}\ i = \texttt{contract}_{\boxed{?}}\ \{i > 0\}\ (\texttt{rec}\ sum.\ (0\ \texttt{fby}\ sum) + i)\ \{sum.\ sum > 0\}$$

To check the contract definition on a concrete input $i = [1; 2]$, we first evaluate the body:

$$\{i \mapsto 1\}; \{i \mapsto 2\} \vdash (\texttt{rec}\ sum.\ (0\ \texttt{fby}\ sum) + i) \Downarrow^{*} [1; 3]$$

We then check that, assuming all inputs are positive, then all results are positive:

$$\{i \mapsto 1\}; \{i \mapsto 2\} \vdash i > 0 \Downarrow^{\square} \top \implies \{i \mapsto 1, sum \mapsto 1\}; \{i \mapsto 2, sum \mapsto 3\} \vdash sum > 0 \Downarrow^{\square} \top$$

It is critical that the rely is ~~always true , then the guarantee is always true . This sort of feedback is necessary for proving properties of mutually-dependent calls. This circular dependency is well-founded as our causality check ensures that recursive streams are guarded by delays () .~~ true *at all points* in the stream. Consider if we had instead used the input stream $i = [-10; 1]$; the rely is false at the first step, but is instantaneously true at the second step. In this case, the sum is -10 at the first step, and -9 at the second step. At both steps the output is negative and the guarantee is false, even though the rely becomes true at the second step. The contract itself remains valid, however, as the assumption is invalid: the input did not satisfy the rely at all steps.

~~We define the~~

The checked semantics of Pipit is defined in **??**. ~~The checked semantics mostly follows the structure of the dynamic semantics, additionally checking any properties and contracts as they are encountered.~~

Rules ChkValue and ChkVar state that values and variables are always valid.

Rule ChkPrim checks a primitive application by descending into the subexpressions. Similarly, rule ChkFby descends into followed-by expressions.

~~Rules ChkFby$_1$ and ChkFby$_S$ are derived from the structure of the big-step rules Fby$_1$ and Fby$_S$. At an input stream of length one, ChkFby$_1$ asserts that all subproperties hold for the (non-existent) previous values in the stream. At subsequent parts of the stream, ChkFby$_S$ discards the most recent element of the stream history and checks the subexpression with the previous inputs.~~

~~Rules~~ Rule ChkRec ~~and~~ checks a recursive-expression $\texttt{rec}\ x.\ e$ by evaluating the overall expression to a stream of values V. The rule then extends the streaming environment Σ with x bound to the values from V; this extended environment is used to descend into the recursive expression.

$$\boxed{\Sigma \vdash_\pi e \text{ valid}}$$

$$\frac{}{\Sigma \vdash_\pi v \text{ valid}}\ (\textsc{ChkValue}) \qquad \frac{}{\Sigma \vdash_\pi x \text{ valid}}\ (\textsc{ChkVar})$$

$$\frac{\Sigma \vdash_\pi e_1 \text{ valid} \quad \ldots \quad \Sigma \vdash_\pi e_n \text{ valid}}{\Sigma \vdash_\pi p(\bar{e}) \text{ valid}}\ (\textsc{ChkPrim}) \qquad \frac{\Sigma \vdash_\pi e' \text{ valid}}{\Sigma \vdash_\pi v \texttt{ fby } e' \text{ valid}}\ (\textsc{ChkFby})$$

$$\frac{\Sigma \vdash \texttt{rec } x.\ e \Downarrow^* V \qquad \Sigma[x \mapsto V] \vdash_\pi e \text{ valid}}{\Sigma \vdash_\pi \texttt{rec } x.\ e[x] \text{ valid}}\ (\textsc{ChkRec})$$

$$\frac{\Sigma \vdash_\pi e \text{ valid} \qquad \Sigma \vdash e \Downarrow^* V \qquad \Sigma[x \mapsto V] \vdash_\pi e' \text{ valid}}{\Sigma \vdash_\pi \texttt{let } x = e \texttt{ in } e'[x] \text{ valid}}\ (\textsc{ChkLet})$$

$$\frac{(\pi = \pi' \implies \Sigma \vdash e \Downarrow^\square \top) \qquad \Sigma \vdash_\pi e \text{ valid}}{\Sigma \vdash_\pi \texttt{check}_{\pi'}\ e \text{ valid}}\ (\textsc{ChkCheck})$$

$$\frac{\begin{array}{c}\Sigma \vdash e_{\text{body}} \Downarrow^* V \\ (\pi = \pi' \implies \Sigma \vdash e_{\text{rely}} \Downarrow^\square \top) \\ (\pi = \boxdot \implies \Sigma \vdash e_{\text{rely}} \Downarrow^\square \top \implies \Sigma[x \mapsto V] \vdash e_{\text{guar}} \Downarrow^\square \top) \\ \Sigma \vdash_\pi e_{\text{rely}} \text{ valid} \\ (\Sigma \vdash e_{\text{rely}} \Downarrow^\square \top \implies \Sigma \vdash_\pi e_{\text{body}} \text{ valid} \ \wedge\ \Sigma[x \mapsto V] \vdash_\pi e_{\text{guar}} \text{ valid})\end{array}}{\Sigma \vdash_\pi \texttt{contract}_{\pi'}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} \text{ valid}}\ (\textsc{ChkContract})$$

Figure 6 Checked semantics for Pipit; the judgment form $\Sigma \vdash_\pi e$ valid denotes that evaluating expression e under streaming history Σ satisfies the checks and rely-guarantee contract requirements that are labelled with property status π.

Rule ChkLet ~~both perform the same unfolding as the corresponding big-step rules and check the resulting expression~~checks a let-expression $\texttt{let } x = e \texttt{ in } e'$ descends into both sub-expressions. To check the body e', the rule first evaluates e and extends the streaming environment.

Finally, the heavy lifting is performed by rules ChkCheck and ChkContract.

Rule ChkCheck ~~applies when checking property status~~ checks the properties marked π ~~of~~ in an expression $\texttt{check}_{\pi'}\ e$. If the check-expression has the same status as what we are checking ($\pi = \pi'$), then we ~~perform the actual check by evaluating the~~ evaluate the expression e and ~~requiring it to evaluate to a stream of trues. Otherwise, we do not need to evaluate the check-expression. In both cases, we~~ require it to be true at all steps. We then unconditionally descend into the ~~expression and check its subexpressions, as they may have~~ subexpression to check any nested properties. Such nested properties are unlikely to be written directly by the user, but might occur after ~~program transformations such as~~ inlining.

Rule ChkContract applies when checking property status π of a contract with expression $\texttt{contract}_{\pi'}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$. ~~Although we only include~~ This rule checks both the contract definition and the call-site. We evaluate the body to a stream V; these values are used to check that the body satisfies guarantee. Although the contract only has one property status~~on the contract~~, conceptually there are two distinct properties: one for the caller (π') and one for the definition ~~itself~~ (assumed to be $\boxdot$). To check the caller property when $\pi = \pi'$, we evaluate the rely e_{rely} and require it to ~~be true~~hold. To check the definition property when $\pi = \boxdot$, we assume that the rely holds, and check that the body satisfies the guarantee. We

also descend into the subexpressions to check them; when checking the body and guarantee, we can assume that the rely holds.

3.2.1 Blessing expressions and contracts

Blessing is a meta-operation that replaces the property statuses in an expression so that all checks and contracts are marked as valid (☑). Blessing an expression requires a proof that~~the checked semantics hold~~, for all input streams, assuming the valid checks hold, then the unknown checks hold:

$$\frac{\forall \Sigma.\ \Sigma \vdash_{\boxed{\checkmark}} e \text{ valid} \implies \Sigma \vdash_{\boxed{?}} e \text{ valid}}{\text{bless } e} \text{ (BlessExpression)}$$

We generally prove the required properties by first translating the program to an abstract transition system, as described in **??**.

Blessing is different for contract definitions, as we need to separate the definition of the contract from the instantiation. To check that a contract definition is valid, we show that if the rely clause is always true for a particular input, then the body satisfies the guarantee for the same inputs. We also assume that the valid properties in the rely, body and guarantee hold, and show the corresponding unknown properties:

$$\begin{aligned}&\texttt{let}\ \text{contract\_valid}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{e_{\text{guar}}\} : \text{prop} = \\ &\quad \forall \Sigma.\ \ (\Sigma \vdash_{\boxed{\checkmark}} (e_{\text{rely}}, e_{\text{body}}, e_{\text{guar}}[x := e_{\text{body}}]) \text{ valid} \ \wedge\ \Sigma \vdash e_{\text{rely}} \Downarrow^{\square} \top) \\ &\qquad \implies (\Sigma \vdash_{\boxed{?}} (e_{\text{rely}}, e_{\text{body}}, e_{\text{guar}}[x := e_{\text{body}}]) \text{ valid} \ \wedge\ \Sigma \vdash e_{\text{guar}}[x := e_{\text{body}}] \Downarrow^{\square} \top)\end{aligned}$$

After proving that the contract is valid for all inputs, we can bless the contract definition. Blessing the contract definition blesses the subexpressions for the rely, body and guarantee, but leaves the contract's *instantiation* property status as unknown:

$$\frac{\text{contract\_valid}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{e_{\text{guar}}\}}{\text{bless\_contract}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{e_{\text{guar}}\}} \text{ (BlessContract)}$$

3.3 Causality and metatheory

To ensure that recursive streams have a computational interpretation, we ~~require~~ implement a causality restriction, similar to standard Lustre [**?**]. This restriction checks that all recursive streams are guarded by a followed-by delay. We implement this as a simple syntactic check: each `rec` x. e can only mention x inside a followed-by. This check ~~is stricter than necessary~~ensures productivity of recursive streams, but can be too strict: for example, the expression `rec` x. (`let` $x' = x + 1$ `in` 0 `fby` x') ~~does mention~~ mentions the recursive stream x outside of the delay and is outlawed, but after inlining the let, it would be causal. We hope to relax this restriction ~~somewhat~~ in future work.

The causality restriction gives us some important properties about the metatheory. The most important property is that the dynamic semantics form a total function: given a streaming history and a causal expression, we can evaluate the expression to a value. These properties are mechanised in F*.

▶ **Theorem 1** (bigstep-is-total). *For any non-empty streaming history* Σ *and causal expression* e*, there exists some value* v *such that* e *evaluates to* v $(\Sigma \vdash e \Downarrow v)$.

The relationship between substitution and the streaming history is also important. In general, we have a substitution property that states that evaluating a substituted expression $e[x := e']$ under some context Σ is equivalent to evaluating e' and adding it to the context Σ:

```
type system (~~input~~input: Γ) (~~result~~result: τ) = {
    state:  Γ;
    free:   Γ;
    init:   heap ~~state~~state;
    step:   heap ~~input~~ input → heap ~~free~~ free → heap ~~state~~ state → step_result ~~state result~~state result;
}

type step_result (~~state~~state: Γ) (~~result~~result: τ) = {
    update: heap ~~state~~state;
    value:  ~~result~~result;
    rely:   prop;
    guar:   prop;
}
```

Figure 7 Abstract transition system type definitions

▶ **Theorem 2** (bigstep-substitute). *For a streaming history* Σ *and causal expressions* e *and* e'*, if* $e[x := e']$ *evaluates to a value* v $(\Sigma \vdash e \Downarrow v)$*, then we can evaluate* e' *to some stream* V $(\Sigma \vdash e' \Downarrow^* V)$ *and extend the streaming history to evaluate* e *to the original value* $(\Sigma[x \mapsto V] \vdash e \Downarrow v)$*. The converse is also true.*

The big-step semantics in Figure 5 for a recursive expression `rec` $x.\ e$ performs one step of recursion by substituting x for the recursive expression. An alternative non-syntax-directed semantics would be to have the environment outside the semantics supply a stream V such that if we extend the streaming history with $x \mapsto V$, then e evaluates to V itself. The above substitution theorem can be used to show that, for causal expressions, these two semantics are equivalent. We can additionally show that, when evaluating e with $x \mapsto V$, the most recent value in V does not affect the result. This fact can be used to "seed" evaluation by starting with an arbitrary value:

▶ **Theorem 3** (bigstep-rec-causal). *For a streaming history* $\Sigma; \sigma$ *and a causal recursive expression* `rec` $x.\ e$*, if* $(\Sigma; \sigma \vdash e \Downarrow v)$*, then updating* $\sigma[x]$ *with any value* v' *results in the same value:* $(\Sigma; \sigma[x \mapsto v'] \vdash e \Downarrow v)$*.*

4 Abstract transition systems

To prove properties about Pipit programs, we translate to an *abstract* transition system, so-called because it abstracts away the implementation details of contract instantiations. For extraction we also translate to *executable* transition systems, which we discuss in **??**.

?? shows the types of transition systems. A transition system is parameterised by its input context and the result type. It also contains two internal contexts: firstly, the state context describes the private state required to execute the machine; secondly, the free context contains any extra input values that the transition system would like to existentially quantify over. The free context is used to allow the system to ask for arbitrary values from the environment, when it would not otherwise be able to return a concrete value.

For recursive streams and contract instantiations, which ~~abstract over the~~ hide their implementation, the natural translation to a transition system would involve ~~an existential quantifier: "there exists some value~~ existentially quantifying a result that satisfies the specification~~"~~. Unfortunately, ~~such~~ using an existential quantifier requires a step *relation* rather

than a step *function*. Using a step relation complicates the resulting transition system, as other operations such as primitive application must also introduce existential quantifiers; such quantifiers block simplifications such as partial-evaluation and result in a more complex transition system. Instead, the free context provides the step function with a fresh unconstrained value of the desired type, which the step function can then constrain.

Back to **??**, the step-result contains the updated state for the transition system, as well as the result value. The step-result additionally contains two propositions; one for the 'rely', or assumptions about the execution environment, and another for the 'guarantee', or obligations that the transition system must show. For the transition system corresponding to an expression e, these propositions are roughly analogous to the known checked semantics $\Sigma \vdash_{\boxed{\checkmark}} e$ valid and unknown checks $\Sigma \vdash_{\boxed{?}} e$ valid respectively.

~~Our implementation includes a mechanised proof that , for~~ For example, recall again the sum contract:

$$\texttt{let}\ \text{sum}\ \mathit{ints}\ =\ \texttt{contract}_{\boxed{?}}\ \{\mathit{ints} > 0\}\ (\texttt{rec}\ \mathit{sum}.\ (0\ \texttt{fby}\ \mathit{sum}) + \mathit{ints})\ \{\mathit{sum}.\ \mathit{sum} > 0\}$$

To verify the contract definition, we first translate it to an abstract transition system whose input environment contains an integer *ints*, and whose result type is also an integer. The followed-by delay results in a local state variable called sum_fby, and we encode the existentially-quantified recursive stream as a free context variable called sum:

```
let sum_def: system (ints: ℤ) ℤ = {
    state  = (sum_fby: ℤ);
    free   = (sum: ℤ);
    init   = { sum_fby = 0 };
    step   = λi f s. {
               update = { sum_fby = f .sum };
               value  = f .sum;
               rely   = (f .sum = s .sum_fby + i .ints) ∧ i .ints > 0;
               guar   = f .sum > 0; } }
```

The initial state of 0 corresponds to the initial value of the followed-by. In the step function, argument i refers to the input heap containing i.ints, f refers to the free heap containing the recursive stream f.sum, and s refers to the state heap containing s.sum_fby. In the rely of the step result, f.sum is constrained to be the translated body of the recursive stream. The translated rely also includes the contract's rely that the input integer is positive. Finally, the translated guarantee includes the contract's guarantee that the output is positive.

To verify the transition system, we prove inductively that if the rely always holds, then the guarantee holds; we discuss proofs of system validity further in **??**.

The translation for contract instantiations is similar, except that the contract body is replaced by an arbitrary value from the free context. For example, we can use the sum contract to implement the Fibonacci sequence with `rec` *fib*. sum (1 `fby` *fib*). This program does not require any input values, so we leave the input context empty. The state context includes an entry for the 1 `fby` *fib* followed-by expression, but does not include the followed-by expressions inside the contract definition. Similarly, the free context includes an entry for the recursive stream, and an entry for the abstract, underspecified value of the contract:

```
let fib_def: system () ℤ = {
```

$$
\begin{array}{rcll}
[\![v]\!]_{\text{state}} & = & \cdot & \\
[\![x]\!]_{\text{state}} & = & \cdot & \\
[\![p(\overline{e})]\!]_{\text{state}} & = & \bigcup_i [\![e_i]\!]_{\text{state}} & \\
[\![v \texttt{ fby } e]\!]_{\text{state}} & = & x_{\texttt{fby}(e)} : \tau, [\![e]\!]_{\text{state}} & (\text{fresh } x_{\texttt{fby}(e)}) \\
[\![\texttt{rec } x.\ e]\!]_{\text{state}} & = & [\![e]\!]_{\text{state}} & \\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{state}} & = & [\![e]\!]_{\text{state}} \cup [\![e']\!]_{\text{state}} & \\
[\![\texttt{check}_\pi\ e]\!]_{\text{state}} & = & [\![e]\!]_{\text{state}} & \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{state}} & = & [\![e_r]\!]_{\text{state}} \cup [\![e_b]\!]_{\text{state}} &
\end{array}
$$

$$
\begin{array}{rcl}
[\![v]\!]_{\text{free}} & = & \cdot \\
[\![x]\!]_{\text{free}} & = & \cdot \\
[\![p(\overline{e})]\!]_{\text{free}} & = & \bigcup_i [\![e_i]\!]_{\text{free}} \\
[\![v \texttt{ fby } e]\!]_{\text{free}} & = & [\![e]\!]_{\text{free}} \\
[\![\texttt{rec } x.\ e]\!]_{\text{free}} & = & x : \tau, [\![e]\!]_{\text{free}} \\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{free}} & = & [\![e]\!]_{\text{free}} \cup [\![e']\!]_{\text{state}} \\
[\![\texttt{check}_\pi\ e]\!]_{\text{free}} & = & [\![e]\!]_{\text{free}} \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{free}} & = & x : \tau, [\![e_r]\!]_{\text{free}} \cup [\![e_b]\!]_{\text{state}}
\end{array}
$$

Figure 8 Transition system typing contexts of expressions; for an expression e, $[\![e]\!]_{\text{state}} : \Gamma$ and $[\![e]\!]_{\text{free}} : \Gamma$ describe the heaps used to store the expression's internal state and extra inputs.

$$
\begin{array}{lll}
\text{state} & = (\text{fib\_fby: } \mathbb{Z}); & \\
\text{free} & = (\text{fib: } \mathbb{Z};\ \text{sum\_contract: } \mathbb{Z}); & \\
\text{init} & = \{\ \text{fib\_fby} = 1\ \}; & \\
\text{step} & = \lambda i\ f\ s.\ \{ & \\
 & \quad \text{update} = \{\ \text{fib\_fby} = f\ .\text{fib}\ \}; & \\
 & \quad \text{value} = f\ .\text{fib}; & \\
 & \quad \text{rely} = (f\ .\text{fib} = f\ .\text{sum\_contract}) & \\
 & \qquad \wedge\ (s\ .\text{fib\_fby} > 0 \implies f\ .\text{sum\_contract} > 0); & \\
 & \quad \text{guar} = s\ .\text{fib\_fby} > 0;\ \}\ \}
\end{array}
$$

As before, the translated rely includes the assumption that the recursive stream's value (f.fib) agrees with its body (f.sum_contract). Additionally, the rely includes the assumption that the contract's rely implies the guarantee: if sum's input (s.fib_fby) is positive, then its output (f.sum_contract) is positive too. Finally, the translated guarantee encodes the obligation that the environment satisfies the *contract's rely* – the input to sum is positive.

Note that the transition system requires the rely to hold *at the current step*, while the "true" semantics of contracts requires the rely to hold *at every step so far*. This minor optimisation is sound, as we define system validity to require all steps to satisfy the rely.

4.1 Translation

We now present the details of the translation. For causal expressions, the translated transition system is verified to be an abstraction of the original expression's dynamic semantics~~.~~ ~~The proof that~~ , and the generated proof obligations imply that the original expression satisfies the ~~rely and guarantee propositions correspond to the~~ checked semantics~~is future work~~.

?? defines the internal state and free contexts required for an expression. For most expression forms, the state and free contexts are defined by taking the union of the contexts of subexpressions. Followed-by delays introduce a local state variable $x_{\texttt{fby}(e)}$ in which to

$$
\begin{array}{lcl}
[\![v]\!]_{\text{init}} & = & () \\
[\![v]\!]_{\text{value}}(i,f,s) & = & v \\
\\
[\![x]\!]_{\text{init}} & = & () \\
[\![x]\!]_{\text{value}}(i,f,s) & = & (i \cup f).x \\
\\
[\![p(\overline{e})]\!]_{\text{init}} & = & \bigcup_i [\![e_i]\!]_{\text{init}} \\
[\![p(\overline{e})]\!]_{\text{value}}(i,f,s) & = & \text{prim-sem}(p, \overline{[\![e]\!]_{\text{value}}(i,f,s)}) \\
[\![p(\overline{e})]\!]_{\text{update}}(i,f,s) & = & \bigcup_i [\![e_i]\!]_{\text{update}}(i,f,s) \\
[\![p(\overline{e})]\!]_{\text{rely}}(i,f,s) & = & \bigwedge_i [\![e_i]\!]_{\text{rely}}(i,f,s) \\
[\![p(\overline{e})]\!]_{\text{guar}}(i,f,s) & = & \bigwedge_i [\![e_i]\!]_{\text{guar}}(i,f,s) \\
\\
[\![v \texttt{ fby } e]\!]_{\text{init}} & = & [\![e]\!]_{\text{init}} \cup \{x_{\texttt{fby}(e)} \mapsto v\} \\
[\![v \texttt{ fby } e]\!]_{\text{value}}(i,f,s) & = & s.x_{\texttt{fby}(e)} \\
[\![v \texttt{ fby } e]\!]_{\text{update}}(i,f,s) & = & [\![e]\!]_{\text{update}}(i,f,s) \cup \{x_{\texttt{fby}(e)} \mapsto [\![e]\!]_{\text{value}}(i,f,s)\} \\
[\![v \texttt{ fby } e]\!]_{\text{rely}}(i,f,s) & = & [\![e]\!]_{\text{rely}}(i,f,s) \\
[\![v \texttt{ fby } e]\!]_{\text{guar}}(i,f,s) & = & [\![e]\!]_{\text{guar}}(i,f,s) \\
\\
[\![\texttt{rec } x.\ e]\!]_{\text{init}} & = & [\![e]\!]_{\text{init}} \\
[\![\texttt{rec } x.\ e]\!]_{\text{value}}(i,f,s) & = & f.x \\
[\![\texttt{rec } x.\ e]\!]_{\text{update}}(i,f,s) & = & [\![e]\!]_{\text{update}}(i,f,s) \\
[\![\texttt{rec } x.\ e]\!]_{\text{rely}}(i,f,s) & = & [\![e]\!]_{\text{rely}}(i,f,s) \\
 & \wedge & f.x = [\![e]\!]_{\text{value}}(i,f,s) \\
[\![\texttt{rec } x.\ e]\!]_{\text{guar}}(i,f,s) & = & [\![e]\!]_{\text{guar}}(i,f,s) \\
\\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{init}} & = & [\![e]\!]_{\text{init}} \cup [\![e']\!]_{\text{init}} \\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{value}}(i,f,s) & = & [\![e']\!]_{\text{value}}(i \cup \{x \mapsto [\![e]\!]_{\text{value}}(i,f,s)\}, f, s) \\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{update}}(i,f,s) & = & [\![e']\!]_{\text{update}}(i \cup \{x \mapsto [\![e]\!]_{\text{value}}(i,f,s)\}, f, s) \\
 & \cup & [\![e]\!]_{\text{update}}(i,f,s) \\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{rely}}(i,f,s) & = & [\![e']\!]_{\text{rely}}(i \cup \{x \mapsto [\![e]\!]_{\text{value}}(i,f,s)\}, f, s) \\
 & \wedge & [\![e]\!]_{\text{rely}}(i,f,s) \\
[\![\texttt{let } x = e \texttt{ in } e']\!]_{\text{guar}}(i,f,s) & = & [\![e']\!]_{\text{guar}}(i \cup \{x \mapsto [\![e]\!]_{\text{value}}(i,f,s)\}, f, s) \\
 & \wedge & [\![e]\!]_{\text{guar}}(i,f,s) \\
\\
[\![\texttt{check}_\pi\ e]\!]_{\text{init}} & = & [\![e]\!]_{\text{init}} \\
[\![\texttt{check}_\pi\ e]\!]_{\text{value}}(i,f,s) & = & () \\
[\![\texttt{check}_\pi\ e]\!]_{\text{update}}(i,f,s) & = & [\![e]\!]_{\text{update}}(i,f,s) \\
[\![\texttt{check}_\pi\ e]\!]_{\text{rely}}(i,f,s) & = & (\pi = \boxtimes \implies [\![e]\!]_{\text{value}}(i,f,s)) \wedge [\![e]\!]_{\text{rely}}(i,f,s) \\
[\![\texttt{check}_\pi\ e]\!]_{\text{guar}}(i,f,s) & = & (\pi = \boxed{?} \implies [\![e]\!]_{\text{value}}(i,f,s)) \wedge [\![e]\!]_{\text{guar}}(i,f,s) \\
\\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{init}} & = & [\![e_r]\!]_{\text{init}} \cup [\![e_g]\!]_{\text{init}} \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{value}}(i,f,s) & = & f.x \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{update}}(i,f,s) & = & [\![e_r]\!]_{\text{update}}(i,f,s) \cup [\![e_g]\!]_{\text{update}}(i,f,s) \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{rely}}(i,f,s) & = & ([\![e_r]\!]_{\text{value}}(i,f,s) \implies [\![e_g]\!]_{\text{value}}(i,f,s)) \\
 & \wedge & (\pi = \boxtimes \implies [\![e_r]\!]_{\text{value}}(i,f,s)) \\
 & \wedge & [\![e_r]\!]_{\text{rely}}(i,f,s) \\
 & \wedge & ([\![e_r]\!]_{\text{value}}(i,f,s) \implies [\![e_g]\!]_{\text{rely}}(i,f,s) \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{guar}}(i,f,s) & = & (\pi = \boxed{?} \implies [\![e_r]\!]_{\text{value}}(i,f,s)) \\
 & \wedge & [\![e_r]\!]_{\text{guar}}(i,f,s) \wedge [\![e_g]\!]_{\text{guar}}(i,f,s)
\end{array}
$$

Figure 9 Transition system semantics; for an expression $\Gamma \vdash e : \tau$, $[\![e]\!]_{\text{init}} : \text{heap}\ [\![e]\!]_{\text{state}}$ is the initial state. For each field of the step-result type, we define a translation function that takes the input, free and state heaps: for example, we define the value-result of a step with type $[\![e]\!]_{\text{value}} : \text{heap}\ \Gamma \to \text{heap}\ [\![e]\!]_{\text{free}} \to \text{heap}\ [\![e]\!]_{\text{state}} \to \tau$.

store the most recent stream value. We generate a fresh variable here, ~~though~~ although the implementation uses de Bruijn indices. Recursive streams and contracts both introduce new bindings into the free context; we assume that their binders x are unique.

?? defines the translation for expressions. Values and ~~expressions~~ variables have no internal state. For variables, we look for the variable binding in either of the input or free heaps; bindings are unique and cannot occur in both. We omit the rely and guarantee definitions here; both are trivially true.

To translate primitives, we union together the initial states of the subexpressions; updating the state is similar. For the rely and guarantee definitions, we take the conjunction: we can assume that all subexpressions rely clauses hold, and must show that all guarantees hold.

To translate a followed-by v `fby` e, we initialise the ~~follow-by~~followed-by's unique binder $x_{\texttt{fby}(e)}$ to the followed-by's default value v. At each step, we return the value in the local state ~~, before~~ *before* updating the local state to the subexpression's new value. ~~The rely and guarantee differ from the checked semantics here: in the checked semantics, we check the subexpression on the previous inputs, but here we check the current subexpression. This means that a single step of the rely and guarantee do not exactly correspond to the checked semantics; however, we posit that they are equivalent for a rely and guarantee that has been proven to hold for any sequence of inputs.~~

To translate a recursive expression `rec` x. e of type τ, we require an arbitrary value $x : \tau$ in the free heap. The rely proposition constrains the free variable x to be the result of evaluating e with the binding for x passed along, thus closing the recursive loop.

To translate let-expressions `let` $x = e$ `in` e', we extend the input heap with the value of e before evaluating e'. The presentation here duplicates the computation of the value of e, but the actual implementation introduces a single binding.

To translate a check property, we inspect the property status. If the property is known to be valid, then we can assume the property is true in the rely clause. Otherwise, we include the property as an obligation in the guarantee clause. In either case, we also include the subexpression's rely and guarantee clauses.

Finally, to translate contract instantiations, we use the contract's rely and guarantee and ignore the body. As with recursive expressions, we require an arbitrary value $x : \tau$ in the free heap. The translation's rely allows us to assume that the contract definition holds: that is, the contract's rely implies the contract's guarantee. If the contract instantiation is known to be valid, we can also assume that the contract's rely holds. Otherwise, we include the contract's rely as an obligation by putting it in the translation's guarantee.

~~In the contract instantiation, we assume that if the contract rely is true *at the current step*, then the contract~~

4.2 Proof obligations and induction

To verify that the translated system satisfies its proof obligations – that is, the checked properties and contract relies hold — we can perform induction on the system's sequence of steps. A system satisfies its proof obligations if, for any sequence of steps that all satisfy its rely or assumptions, the system's guarantee also holds ~~at the current step. The real semantics of the contract, however, requires the contract rely to be true~~ for all of the steps.

Inductive proofs on Lustre programs generally use a non-standard definition of induction, as the property we wish to show is a function of the ~~*at every*~~ *step* ~~*so far. This difference is benign, as the inductive case of the proof of transition system validity assumes that both result*~~, rather than being a function of the *state*. This means that the base case must take

a single step from the initial state to be able to state the property that, if the step result's rely holds, then its guarantee holds:

```
let inductive_check_base (sys : system input τ) : prop =
  ∀(i : heap input)(f : heap sys.free).
  let stp = sys.step i f sys.init  in
  stp.rely ⟹ stp.guar
```

For the inductive step case, we allow the system to take *two* steps from an arbitrary state, assuming that both steps satisfy the rely and ~~guarantee held at previous steps; if the rely also holds now, then it has held at every step so far.~~ the first step satisfied the inductive property:

```
let inductive_check_step (sys : system input τ) : prop =
  ∀(i₀ i₁ : heap input)(f₀ f₁ : heap sys.free)(s₀ : heap sys.state).
  let stp₁ = sys.step i₀ f₀ s₀  in
  let stp₂ = sys.step i₁ f₁ stp₁.state  in
  stp₁.rely ⟹ stp₁.guar ⟹ stp₂.rely ⟹ stp₂.guar
```

This inductive scheme also generalises to *k-induction*, which allows the inductive case to assume the previous k steps satisfied the inductive property, rather than just assuming that the one previous step holds. K-induction is a fairly standard invariant strengthening technique; intuitively, it allows the proof to use more context of the history of execution [**?**, **?**, **?**].

To reason about system validity in general, we define a predicate *system_holds_all* that formally defines a valid system as: for all sequences of inputs and their corresponding steps, if all of the steps' relies hold, then the guarantees also hold. Validity is implied by (k-)induction.

4.3 Translation correctness proofs

We prove that the transition system is an abstraction of the dynamic semantics: that is, if the expression evaluates to v under some context, then there exists some execution of the transition system that also results in v. The transition system itself is deterministic, but the free context provides the non-determinism which may occur from underspecified contracts; our theorem statement existentially quantifies the free heap.

The results presented here rely heavily on the totality and substitution metaproperties described in **??**. **??** defines the invariant for the abstraction proof; the judgment form $\Sigma \vdash e \sim s$ checks that s is a valid state heap. We use the invariant to state that, if executing the transition system for e on the entire streaming history Σ results in state heap s, then s is a valid state.

As most expressions do not modify the state heap, the invariant for most expressions simply descends into the subexpressions. Where new bindings are added, we use the dynamic semantics to extend the context with the new values. The invariant for ~~follow-by~~ followed-by expressions asserts that the initial state of the ~~follow-by~~ followed-by is the default value; on subsequent steps, the state corresponds to the dynamic semantics. With this invariant, we can prove abstraction:

▶ **Theorem 4** (translation-abstraction). *For a well-typed causal expression e and streaming history Σ, if e evaluates to stream V ($\Sigma \vdash e \Downarrow^* V$), then there exists a sequence of free heaps Σ_F such that repeated application of the transition system's step results in V.*

$$\boxed{\Sigma \vdash e \sim s}$$

$$\frac{}{\Sigma \vdash v \sim s}\ (\textsc{IValue}) \qquad \frac{}{\Sigma \vdash x \sim s}\ (\textsc{IVar})$$

$$\frac{\Sigma \vdash e_1 \sim s \quad \ldots \quad \Sigma \vdash e_n \sim s}{\Sigma \vdash p(\overline{e}) \sim s}\ (\textsc{IPrim}) \qquad \frac{s.x_{\texttt{fby}(e')} = v \qquad \cdot \vdash e' \sim s}{\cdot \vdash v\ \texttt{fby}\ e' \sim s}\ (\textsc{IFby}_0)$$

$$\frac{\Sigma;\sigma \vdash e' \Downarrow s.x_{\texttt{fby}(e')} \qquad \Sigma;\sigma \vdash e' \sim s}{\Sigma;\sigma \vdash v\ \texttt{fby}\ e' \sim s}\ (\textsc{IFby}_S)$$

$$\frac{\Sigma \vdash \texttt{rec}\ \ x.\ e \Downarrow^* V \qquad \Sigma[x \mapsto V] \vdash e \sim s}{\Sigma \vdash \texttt{rec}\ \ x.\ e[x] \sim s}\ (\textsc{IRec})$$

$$\frac{\Sigma \vdash e \Downarrow^* V \qquad \Sigma \vdash e \sim s \qquad \Sigma[x \mapsto V] \vdash e' \sim s}{\Sigma \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e'[x] \sim s}\ (\textsc{ILet})$$

$$\frac{\Sigma \vdash e \sim s}{\Sigma \vdash \texttt{check}_\pi\ e \sim s}\ (\textsc{ICheck})$$

$$\frac{\Sigma \vdash e_{\text{body}} \Downarrow^* V \qquad \Sigma \vdash e_{\text{rely}} \sim s \qquad \Sigma[x \mapsto V] \vdash e_{\text{guar}} \sim s}{\Sigma \vdash \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} \sim s}\ (\textsc{IContract})$$

Figure 10 Transition system state invariant

Finally, we can show the main entailment result that if the proof obligations hold on the system, then the original program is valid according to the checked semantics:

▶ **Theorem 5** (translation-entailment). *For a well-typed causal expression e and its translated system s, if the system holds (system_holds_all s), and the checked properties in e hold* ($\forall\Sigma.\ \Sigma \vdash_{\boxed{\checkmark}} e$ *valid*)*, then the unknown properties in e also hold* ($\forall\Sigma.\ \Sigma \vdash_{\boxed{?}} e$ *valid*)

The above theorem allows us to *bless* the expression and mark all properties as valid (**??**). Importantly, the assumption that the checked properties hold lets us re-use previously-verified properties without re-proving them, allowing for modular proofs.

5 Extraction

Pipit can generate executable code which is suitable for real-time execution on embedded devices. The code extraction uses a variation of the abstract transition system described in **??**, with two main differences to ensure that the result is executable without relying on the environment to provide values for the free context. Contracts are straightforward to execute by using the body of the contract rather than abstracting over the implementation.

To execute recursive expressions $\texttt{rec}\ \ x.\ e : \tau$, we require an arbitrary value of type τ to seed the fixpoint, as described in **??**. We first call the step function to evaluate e with x bound to $\bot_\tau$. This step call returns the correct value, but the updated state is invalid, as it may refer to the bottom value. To get the correct state, we call the step function again, this time with ~~e bound to~~ x bound to the correct value, v.

For example, for the *sum* contract with body (`rec` *sum*. (0 `fby` *sum*) + *ints*), we generate an executable system that takes an input context containing integer variable *ints*, with a single state variable for the followed-by, and returning an integer:

```
let sum_def: system (ints: ℤ) ℤ = {
    state  = (sum_fby: ℤ);
    init   = { sum_fby = 0; };
    step   = λi s.
               let (fby_0, s_0)   = (s.sum_fby, s {sum_fby = ⊥_ℤ})   in
               let (sum_0, s_0)   = (fby_0 + i.ints, s_0)   in
               let (fby_1, s_1)   = (s.sum_fby, s {sum_fby = sum_0})   in
               let (sum_1, s_1)   = (fby_1 + i.ints, s_1)   in
               (sum_0, s_1) }
```

Here, the step function takes heaps of the input and state contexts, and returns a pair of the result value and the updated state. The first two bindings correspond to the seeded evaluation with the recursive value for the sum set to $\bot_{\mathbb{Z}}$; as such, the resulting state s_0 is invalid. The last two bindings recompute the state, this time with the correct recursive value sum_0 used in the state. This duplication of work can often be removed by the partial evaluation and dead-code-elimination which we perform during code extraction.

This translation to transition systems is verified to preserve the original semantics. The invariant is very similar to that of **??**, except that the invariant descends into the implementations of contracts. For the abstract systems we only showed abstraction; to prove that executable systems are equivalent to the original semantics, we ~~rely on~~ use the fact that the original semantics and transition systems are both deterministic and total (**??**).

▶ **Theorem 6** (execution-equivalence). *For a well-typed causal expression e and streaming history Σ, e evaluates to stream V ($\Sigma \vdash e \Downarrow^* V$) if-and-only-if repeated application of the transition system's step on Σ also results in V.*

To extract the program, we use a *hybrid embedding* as described in [**?**], which is similar to staged-compilation. The hybrid embedding involves a deep embedding of the Pipit core language, while the translation to executable transition systems produces a shallow embedding. We use the F* host language's normalisation-by-evaluation and tactic support [**?**] to ~~specialise~~ partially-evaluate the application of the translation to a particular input program. This ~~specialisation~~ partial-evaluation results in a concrete transition system that fits in the Low* subset of F*, which can then be extracted to statically-allocated C code [**?**].

The ~~translation for recursive streams described above calls the step function of the sub-stream twice, which can duplicate work. The normalisation strategy used to partially-evaluate the translation inlines the two occurrences of the step function, and is often able to remove the duplicate work, but this removal is not guaranteed. Our~~ generated C code for *sum*[2] includes a struct type to hold the state information, as well as reset and step functions:

```
struct sum_state  uint32_t sum_fby;
void sum_reset(struct sum_state* state);
int sum_step(struct sum_state* state, uint32_t ints);
```

[2] This interface is for a variant of the sum contract with 32-bit integers instead of unbounded integers.

The reset function takes the pointer to the state struct and sets it to its initial values. The step function takes the pointer to the state struct and the inputs, and returns the result integer. The state struct is updated in-place. The implementations of these functions avoid dynamic (heap) allocation and are suitable for embedded systems. This interface is standard for Lustre compilers [?, ?] and other synchronous languages.

Unfortunately, our current approach is ~~also~~ unsuitable for generating imperative array code, as our ~~shallowly-embedded~~ pure transition system ~~requires~~ only supports pure arrays. In the future, we intend to ~~address~~ support efficient array computations and fix the above work duplication by introducing an intermediate imperative language such as Obc [?], a static object-based language suitable for synchronous systems. Even with an added intermediate language, we believe that a variant of our current translation and proof-of-correctness will remain useful as an intermediate semantics.

6 Evaluation

To evaluate Pipit, we have implemented the high-level logic of a ~~time-triggered~~ Time-Triggered Controller Area Network (~~CAN~~TTCAN) bus driver [?], described earlier in Section 2. The CAN bus is ~~commonly found~~ common in safety-critical automotive and industrial settings. The time-triggered network architecture defines a static schedule of network traffic~~. All~~; by having all nodes on the network ~~must~~ adhere to the ~~same schedule, which significantly increases~~ schedule, the reliability of periodic messages is significantly increased [?].

~~At a high level, the schedule is described by a system matrix which consists of rows of basic cycles. Each basic cycle consists of a sequence of actions to be performed at particular time-marks. Actions in the schedule may not be relevant to all nodes, so each node has its own local array containing the relevant triggers; trigger actions include sending and receiving application-specific messages, sending reference messages, and triggering 'watch' alerts. The trigger action for receiving an application-specific message checks that a particular message has been received since the trigger was last executed; depending on this, the driver increments or decrements a message-status-counter, which will in turn signal an error once the upper limit is reached. Reference messages start a new basic cycle and are used to synchronise the nodes. Watch alerts are generally placed after the expected end of the cycle and are used to signal an error if no reference message is received.~~

The TTCAN protocol can be implemented in two levels of increasing complexity. In the first level, reference messages, which perform synchronisation between nodes, contain the index of the newly-started cycle. In the second level, the reference messages also contain the value of a global fractional clock and whether any gaps have occurred in the global clock, which allows other nodes to calibrate their own clocks. We implement the first level as it is more amenable to software implementation [?].

The implementation defines a streaming function that takes a stream describing the current time, the state of the hardware, and any received messages. It returns a stream of commands to be performed, such as sending a particular reference message. The implementation defines a pure streaming function. To actually interact with the hardware we assume a small hardware-interop layer that reads from the hardware registers and translates the commands to hardware-register writes, but we have not yet implemented this. We package the driver's inputs into a record for convenience:

```
type driver_input = {
    local_time: network_time_unit;
    mode_cmd: option mode;
```

```
    tx_status:   tx_status;
    bus_status:  bus_status;
    rx_ref:      option ref_message;
    rx_app:      option app_message_index;
}
```

Here, the local-time field denotes the time-since-boot in *network time units*, which are based on the bitrate of the underlying network bus. The mode-command is an optional field which indicates requests from the application to enter configuration or execution mode. The transmission-status describes the status of the last transmission request and may be none, success, or various error conditions. The bus-status describes whether the bus is currently idle, busy, or in an error state. The two receive fields denote messages received from the bus; for application-specific messages the time-triggered logic only needs the message identifier.

The driver-logic returns a stream of commands for the hardware-interop layer to perform:

```
type commands = {
    enable_acks: bool;
    tx_ref:      option ref_message;
    tx_app:      option app_message_index;
    tx_delay:    network_time_unit;
}
```

The enable-acknowledgements field denotes whether the hardware should respond to messages from other nodes with an acknowledgement bit; in the case of a severe error acknowledgements are disabled, as the node must not write to the bus at all. The transmit fields denote whether to send a reference message or an application-specific message. For application-specific messages, the hardware-interop layer maintains the transmission buffers containing the actual message payload. To meet the schedule as closely as possible, the driver anticipates the next transmission and includes a transmission delay to tell the hardware exactly when to send the next message.

6.1 Runtime

The implementation includes an extension of the trigger-fetch logic described in Section 2, as well as state machines for tracking node synchronisation, master status and fault handling. We generate real-time C code as described in **??**. We evaluated the generated C code by executing with randomised inputs and measuring the worst-case-execution-time on a Raspberry Pi Pico (RP2040) microcontroller. The runtime of the driver logic is fairly stable: over 5,000 executions, the measured worst-case execution time was ~~114μs~~140μs, while the average was ~~107μs~~ 90μs with a standard deviation of ~~2.3μs~~1.5μs. Earlier work on fault-tolerant TTCAN [**?**] describes the required slot sizes — the minimum time between triggers — to achieve bus utilisation at different bus rates. For a 125Kbit/s bus, a slot size of approximately 1,500μs is required to achieve utilisation above 85 per cent. For the maximum CAN bus rate of 1Mbit/s, the required slot size is 184μs. Further evaluation is required to ensure that the complete runtime including the hardware-interop layer is sufficient for full-speed CAN.

Our code generation can be improved in a few ways. A common optimisation in Lustre is to fuse consecutive if-statements with the same condition [**?**]; such an optimisation seems useful here, as our treatment of optional values introduces repeated unpacking and repacking. Some form of array fusion [**?**] may also be useful for removing redundant array operations.

```
let rec next (i: int) (c: cycle):
    Tot  (option int)
         (decreases (count - i)) =
    if trigger_enabled i c
    then Some i
    else if i ≥ count − 1
    then None
    else next (i + 1) c
```

```
  function next(index:  int; c: cycle)
        returns (result: int)
  var next_array: int ^ COUNT;
  let
    next_array[i] =
      if trigger_enabled(COUNT - 1 - i, c)
      then COUNT - 1 - i
      else if i <= 0
      then NO_NEXT_TRIGGER
      else next_array[i - 1];
    result =
      next_array[COUNT - 1 - index];
  tel
```

Figure 11 Left: next-trigger logic in F*; right: Kind2 encoding as array scan. In F*, the *Tot τ (decreases ...)* syntax declares a total function with the given termination measure. In Kind2, the `int^COUNT` syntax denotes the type of an array of integers of length `COUNT`, while the `next_array[i]` declaration defines the elements of the array as a function of the index `i`.

Our current extraction generates a transition-system with a step function which returns a tuple of the updated state and result. Composing these step functions together results in repeated boxing and unboxing of this tuple; we currently rely on the F* normaliser to remove this boxing. In the future, we plan to build on the current proofs to implement a more-sophisticated encoding that introduces less overhead.

6.2 Verification

We have verified a simplified trigger-fetch mechanism, as presented earlier (Section 2). For comparison, we implemented the same logic in the Kind2 model-checker [**?**]. The restrictions placed on the triggers array — that triggers are sorted by time-mark, that there must be an adequate time-gap between a trigger and its next-enabled, and that a trigger's time-mark must be greater-than-or-equal-to its index — are naturally expressed with quantifiers. The Kind2 model-checker includes experimental array and quantifier support [**?**]. Due to the experimental nature of these features, we had to work around some limitations: for example, the use of arrays and quantifiers disables IC3-based invariant generation; quantified variables cannot be used in function calls; and the use of top-level constant arrays caused runtime errors that rendered most properties invalid [**?**].

We were able to ~~reliably verify the~~ express equivalent properties in Kind2 ~~implementation of the simplified trigger-fetch mechanism for trigger arrays containing up to 16 elements; above that , we ran into intermittent runtime errors. For reference,~~ and in Pipit, aside from some encoding issues. For example, the specification-only function that finds the next trigger is naturally recursive. Kind2 does not support recursive functions, but we were able to encode it by introducing a temporary array and using Kind2's array comprehension syntax for scanning over arrays. Additionally, while the recursive call *increases* the index, the array scan can only depend on values with lower indices. **??** illustrates this encoding with a simplified version of the ~~M__TTCAN hardware implementation of TTCAN supports up to 64 triggers [?]~~next-trigger logic.

~~We made~~

	Kind2				Pipit	
	simple enable-set		full enable-set			
size	wall-clock	~~user-time~~ CPU time	wall-clock	~~user-time~~ CPU time	wall-clock	~~user-time~~ CPU time
1	~~1.51~~1.48s	~~1.33~~1.06s	~~1.61~~1.57s	2.26s	~~5.28~~5.25s	5.03s
2	~~1.50~~1.51s	~~1.29~~1.26s	~~1.68~~1.71s	~~2.85~~2.93s	~~5.28~~5.25s	5.03s
4	~~1.56~~1.57s	~~1.59~~1.62s	~~2.10~~2.08s	~~5.05~~4.78s	~~5.28~~5.25s	5.03s
8	~~1.80~~1.76s	~~3.12~~3.07s	~~4.29~~4.21s	~~18.12~~16.98s	~~5.28~~5.25s	5.03s
16	~~3.41~~3.36s	~~12.29~~11.91s	~~16.41~~13.82s	~~71.23~~65.57s	~~5.28~~5.25s	5.03s
32	~~11.24~~12.15s	~~67.27~~62.38s	~~941.64~~269.14s	~~3853.16~~1230.05s	~~5.28~~5.25s	5.03s
64	1701.01s	9096.99s	(timeout)		5.25s	5.03s
128	(timeout)		(timeout)		5.25s	5.03s

Figure 12 Verification time for trigger-fetch; simple enable-set uses a simplified version of the enable-set, while full enable-set uses bitwise arithmetic as in the TTCAN specification. The wall-clock time denotes the elapsed time that an engineer must spend waiting for the result; the CPU time denotes the total time spent computing by all of the CPU cores. The verification time for Pipit is a once-and-for-all proof that is parametric in the size of the array. ~~We were unable to verify arrays of size 64 with Kind2 as our verification timed out after three hours~~The time limit was one hour.

We compare against two Kind2 implementations: one corresponds closely to the Pipit development, while the other includes a critical simplification ~~in the Kind2 implementation, which was~~ to modify the trigger-enabled set to be a single cycle index. In ~~the specification~~TTCAN proper, the enabled set is implemented as a cycle-offset and repeat-factor. Checking if a trigger is enabled in the current cycle requires nonlinear arithmetic, which is difficult for SMT solvers. In our Pipit development, we can treat the definition of the cycle set abstractly. However, in the Kind2 development, ~~quantifiers~~ quantified formulas cannot contain function calls, which means that we cannot hide the implementation of the enabled-set check by providing an abstract contract. This limitation also makes the specification quite unwieldy, as ~~functions must be manually inlined~~we must manually inline any functions in quantified formulas.

?? shows the verification runtime for different sizes of arrays; the Pipit version is parametric in the array size, and is thus verified for all sizes of arrays. We ran these experiments in Docker on an Intel i5-12500 with 32GB of RAM. Both Kind2 and Pipit developments of the ~~simplified~~ trigger-fetch logic are roughly the same size, on the order of two-hundred lines of code including comments. Ignoring whitespace and comments, the Pipit implementation of trigger-fetch has 26 lines of actual executable code, while the Kind2 code has 32. The majority of the remaining code comprises the definition of valid schedules (34 for Pipit, 28 for Kind2), and the lemma statements and invariants (12 for Pipit, 31 for Kind2), as well as contract statements and boilerplate.

We were able to verify the Kind2 implementation of the complete trigger-fetch mechanism for up to 32 triggers; above that, our verification timed out after one hour. For the simplified trigger-fetch mechanism, we were able to verify up to 64 triggers. For reference, hardware implementations of TTCAN such as M_TTCAN support up to 64 triggers [?].

We plan to verify the remainder of the TTCAN implementation and publish it separately. Prior work formalising TTCAN has variously modeled the protocol itself [?, ?, ?], instances of the protocol [?], and abstract models of TTCAN implementations [?], but we are unaware of any prior work that has verified an *executable* implementation of TTCAN.

Separately, Pipit has also been used to implement and verify a real-time controller for a coffee machine reservoir control system [?]. The reservoir has a float switch to sense the

water level and a solenoid to allow the intake of water. The specification includes a simple model of the water reservoir and shows that the reservoir does not exceed the maximum level under different failure-mode assumptions.

7 Related work

Using existing Lustre tools to verify *and* execute the time-triggered CAN driver from Section 2 is nontrivial. Compiling the triggers array with an unverified compiler such as Lustre V6 [?] or Heptagon [?] is straightforward; however, the verified Lustre compiler Vélus [?] does not support arrays, records, or a foreign-function interface. Recent work on translation validation for LustreC [?] also does not yet support arrays.

Verifying the time-triggered CAN driver is trickier, as the restrictions placed on the triggers array — that triggers are sorted by time-mark, there must be an adequate time-gap between a trigger and its next-enabled, and a trigger's time-mark must be greater-than-or-equal-to its index — naturally require quantifiers. As described in **??**, Kind2 does include experimental array and quantifier support, but in our experiments was unable to verify the full logic for arrays up to the 64 triggers, which is the size supported by hardware implementations of TTCAN. Additionally, due to the limitations ~~on top-level array definitions~~that require the constant triggers array to be passed as an argument, compiling the program with Lustre V6 would result in the entire triggers array being copied to the stack each iteration, which is unlikely to result in acceptable performance.

Other model-checkers for Lustre such as Lesar [?], JKind [?] and the original Kind [?] do not support quantifiers. It may be possible to encode the quantifiers as fixed-size loops in those that support arrays, but ensuring that these loops do not affect the execution or runtime complexity of the generated code does not appear to be straightforward.

These model-checkers have definite usability advantages over the general-purpose-prover approach offered here: they can often generate concrete counterexamples and implement counterexample-based invariant-generation techniques such as ICE [?] and PDR [?, ?]. However, even when the problem can be expressed, these model-checkers do not provide much assurance that the semantics they use for proofs matches the compiled code. In the future, we would like to investigate integrating Pipit with a model-checker via an unverified extraction: such an extraction may allow some of the usability benefits such as counterexamples and invariant generation. If this integration were used solely for debugging and suggesting candidate invariants, then such a change would not necessarily expand the trusted computing base — that is, we could augment our end-to-end verified workflow with *unverified but validated* invariant generation.

Recent work has also introduced a form of refinement types for Lustre [?]. Rather than using transition systems, this work generates self-contained verification conditions based on the types of streams. Such a type-based approach promises to allow abstraction of the implementation details. However, for general-purpose functions such as *count_when* from Section 2, it is not clear how to give it a specification that actually *abstracts* the implementation: a simple specification that the result is within some range would hide too much and be insufficient for verifying the rest of the system. For such functions, the best specification is likely to include a re-statement of the implementation itself.

The embedded language Copilot generates real-time C code for runtime monitoring [?]. Recent work has used translation validation to show that the generated C code matches the high-level semantics [?]. Copilot supports model-checking via Kind2; however, the model-checking has a limited specification language and does not support contracts.

Early work embedding a denotational semantics of Lucid Synchrone in an interactive theorem prover focussed on the semantics itself, rather than proving programs [**?**]. There is ongoing work to construct a denotational semantics of Vélus for program verification [**?**]. We believe that the hybrid SMT approach of F* will allow for a better mixture of automated proofs with manual proofs. Compared to Vélus alone, the trusted computing base of Pipit is larger: we depend on all of F*, Low*'s unverified C code extraction and the Z3 SMT solver; in comparison, Vélus' C code generation is verified and does not depend on any SMT solver.

The deferred aspect of our proofs is similar to the deferred proofs of verification conditions for imperative programs, such as [**?**]. However, such verification conditions are *syntactically* deferred so that the verification condition can be proved later; in our case, the verification conditions are *semantically* deferred, so that more knowledge of the enclosing program can be exploited in the proof. In imperative programs, this sort of extra knowledge is generally provided explicitly as loop invariants, and non-looping statements have their weakest precondition computed automatically. In Lustre-style reactive languages such as ours, programs tend to be composed of many nested recursive streams, which perform a similar function to loops. Explicitly specifying an invariant for each recursive stream would be cumbersome; deferring the proof allows such invariants to be implicit.

8 Conclusion

We have presented Pipit, a verified compiler and proof system for reactive systems. Our implementation of the TTCAN driver logic shows that, by embedding pure F* functions for array operations, Pipit can express programs which are currently unsupported by other verified Lustre compilers. Pipit can also verify high-level program properties which are difficult to express and prove in existing Lustre model-checkers. Our development includes verified translations to both abstract and executable transition systems; both are shown to preserve the dynamic semantics. We also introduced a checked semantics, which describes the ~~proof obligations of a program; in future work, we intend to verify that the~~ semantics of checked properties and contracts; proof obligations generated by ~~the~~ translation to abstract transition system ~~match the checked~~ are verified to correspond to these semantics.

In the future, we intend to verify the remainder of the TTCAN driver logic. We also intend to increase the expressivity of Pipit by adding *clocks*, which are used to describe partially-defined streams [**?**]. Clocks are important for composing complex systems together and avoiding unnecessary computation; they may be useful if it becomes necessary to optimise the runtime of the TTCAN driver.

We are interested in further pursuing the intersection of model-checking with interactive theorem proving. A smart-contract called Djed [**?**] currently uses a mixture of Kind2 [**?**] and manual Isabelle/HOL proofs to show that the contract is well-behaved. In future work, we would like to further investigate whether Pipit's integration of streaming proofs with F*'s automated proof system would be able to provide similar proofs, without introducing any semantic gap between the two systems.