1 Overview

This document

1. defines terms relating to arrow models of LF programs;
2. describes properties of such models, including properties of an addition operation over logical times;
3. describes an algorithm for producing finite canonical representations of such models;
4. describes an algorithm for abstracting parts of a program out of such models, e.g., to conceal the internals of a module; and
5. includes examples to illustrate the meaning and purpose of such models.

2 Comments about the original post of discussion #1307

A few high-level comments on the discussion post:

1. Discussion #1307 includes motivation, description, and intuition. This document was written because the discussion post does not communicate any detailed understanding or analysis.
2. The need for function-like reactors[1] was the primary motivation for the discussion post. A reasonable question in the context of any programming language is, "if I put something in (to a procedure, function, method, etc.), is it *certain* or *uncertain* that I get something out?"[2] In addition, in the context of reactors, it also makes sense to ask, "if I do get a response, *when* do I get it?" Even the question of "when" pertains to correctness of the computation, not just the timing of it; for elaboration on this point, see section 11. Given this context, it is unsurprising that the models discussed here represent functions accurately but have problems in representing certain other types of modules, such as sporadic producers (section 12) or even FSMs (section 10).
3. Another of the initial motivations was our discussions of causality interfaces (which are needed to maintain encapsulation while explaining causality loops to end users). Although it is possible to imagine that an approach similar to the one discussed here could also be used to generate causality interfaces, I have come to view causality interfaces as a separate problem.[3]

[1] The term Marten used in describing Magnition's feedback was "request-response patterns," which might mean the same as what Edward has described as function reactors.

[2] Originally, I had constructs like null-safe type systems and Haskell-style "maybe" types in mind because these specify whether a module (e.g. procedure, method) returns something meaningful. However, the lack of interface information about whether a reactor produces a response is worse than that. One does not know whether the module returns *at all*, so now we have unspecified control flow in addition to unspecified data validity. I think Marten said that some users regard our programming model as a throwback to GO TO-oriented programming, probably for this reason.

[3] Indeed, a fundamentally different concept of causality is used in this document than the one that we need to in order to construct causality interfaces. To avoid muddying the waters, I will restrict my focus to just one concept of causality.

4. It might have been a mistake to use the symbols `?` and `!` in my original post because their meaning there is unrelated to their meaning in the context of interface automata. It is a coincidence that, like in interface automata, they have opposite and complementary implications for the compatibility of different components.

3 Definitions/Interpretation

I will use the term "present" to apply to reactions as well as to ports and triggers. A reaction is present at a tag[4] $g \in \mathbb{N} \times \mathbb{Z}$ if it is triggered at any point during the execution of g. Ports and triggers are present at a tag g if their `is_present` field is set to be `true` at any point during the execution of g.

A (logical) time difference is[5] an element of $\mathbb{N} \times \mathbb{Z}$. Although I use the word "difference" to distinguish these conceptually from absolute logical times, time differences and logical times are mathematically identical because an (absolute) logical time is a difference from time zero; therefore I will use similar notation for both.[6]

A (logical) time smear Δg is a set of (logical) time differences.

Addition[7] of time differences can be defined as follows:[8]

$$(t_0, n_0) + (t_1, n_1) = \begin{cases} (t_0, n_0 + n_1), & t_1 = 0 \\ (t_0 + t_1, n_1), & \text{otherwise} \end{cases}$$

[4]It will be apparent later why it might be useful to allow microsteps to be integers instead of just natural numbers.

[5]This definition will be extended in section 11, but it is good enough to get started.

[6]The extension proposed in 11 is in this sense a generalization of the concept of logical times.

[7]It is intuitive to use the term "addition" even though we are not talking about an abelian group nor even a group; I hope this abuse of language will be forgiven.

[8]This addition is (obviously) intended to model how events are scheduled in actual LF programs. I recently found out that the scheduling rules are more complicated than this in the C target, but the C++ target does not have this problem, and the situation might change depending on how we handle issue #1464.

Such addition is not commutative,[9] but it is associative.[10]

Note also that the operation of left-addition by some fixed element of $\mathbb{N} \times \mathbb{Z}$ is injective,[11] hence left-invertible, whereas right-addition is not injective in general.[12] The range of left-addition by (t, n) is $\{(t', n') : t' \geq t, n' \in \mathbb{Z}\}$.

A consequence of this non-injectivity of right-addition is that we cannot have equivalence classes modulo some closed-under-addition subset.[13] Perhaps the closest approximation to equivalence modulo $\lambda > (0, 0)$ is "equivalence in λ-reachability from g_0," where we can declare two time differences g_1, g_2 equivalent if there exists a time difference $(0, 0) \leq h < \lambda$

[9]For the rules of "commuting" summands with each other, see section 8. It works, but summands do not pass through each other unaltered.

[10]Proof of associativity:

$$((t_0, n_0) + (t_1, n_1)) + (t_2, n_2) = \left(\begin{cases} (t_0, n_0 + n_1), & t_1 = 0 \\ (t_0 + t_1, n_1), & \text{otherwise} \end{cases} \right) + (t_2, n_2) \tag{1}$$

$$= \begin{cases} (t_0, (n_0 + n_1) + n_2), & t_1 = 0 \wedge t_2 = 0 \\ (t_0 + t_2, n_2), & t_1 = 0 \wedge t_2 \neq 0 \\ ((t_0 + t_1), n_1 + n_2), & t_1 \neq 0 \wedge t_2 = 0 \\ ((t_0 + t_1) + t_2, n_2), & t_1 \neq 0 \wedge t_2 \neq 0 \end{cases} \tag{2}$$

$$= \begin{cases} (t_0, n_0 + (n_1 + n_2)), & t_1 = 0 \wedge t_2 = 0 \\ (t_0 + (t_1 + t_2), n_2), & t_1 = 0 \wedge t_2 \neq 0 \\ (t_0 + t_1, (n_1 + n_2)), & t_1 \neq 0 \wedge t_2 = 0 \\ (t_0 + (t_1 + t_2), n_2), & t_1 \neq 0 \wedge t_2 \neq 0 \end{cases} \tag{3}$$

$$= (t_0, n_0) + \left(\begin{cases} (t_1, n_1 + n_2), & t_2 = 0 \\ (t_1 + t_2, n_2), & \text{otherwise} \end{cases} \right) \tag{4}$$

$$= (t_0, n_0) + ((t_1, n_1) + (t_0, n_0)) \tag{5}$$

Note that I made use of the fact that time differences were elements of $\mathbb{N} \times \mathbb{Z}$ and not $\mathbb{Z} \times \mathbb{Z}$ when I assumed on line 4 that $t_1 + t_2 = 0 \iff t_1 = 0 \wedge t_2 = 0$.

[11] Proof: suppose $g + g_0 = g + g_1$. Let $\text{first}(g)$ denote the first element of g and $\text{second}(g)$ denote the second element of g. Then either the $\text{first}(g_0) = \text{first}(g_1)$ or not. In the latter case, we already have a contradiction because our addition acts like regular integer addition in the first slot, and integer addition is injective. Now, consider two sub-cases of the first case: Either $\text{first}(g_0) = \text{first}(g_1) = 0$, or $\text{first}(g_0) = \text{first}(g_1) = t \neq 0$. In the latter case, we must have that $g_0 = g_1$ because when the right summand is zero in the first slot, addition acts like regular addition in the second slot. In the former case, we also must have that $g_0 = g_1$ because $\text{second}(g_0) = \text{second}(g + g_0) = \text{second}(g + g_1) = \text{second}(g_1)$.

[12]This non-injectivity will cause some awkwardness, but nothing insurmountable, in section 8. The left/right asymmetry, and in particular the non-invertibility of right-addition, has an advantage that may justify the complication. This advantage will be explored in section 11.2. The basic idea is that we may need to drop events to enforce certain guarantees in the presence of turing-complete scheduling logic, and that one controlled, predictable way to drop events is to make right-addition by a time smear non-injective.

[13]As a naive attempt, you say that a is congruent to b modulo H if $a^{-1}b \in H$. Then you don't have symmetry necessarily because there is no guarantee that $b^{-1}a \in H$, even if you require H to be closed under left-inversion, since $b^{-1}a$ is *not* necessarily the inverse of $a^{-1}b$ (since a^{-1} is not the right-inverse of a, or equivalently, a is not the left-inverse of a^{-1}). Suppose you then try to *force* symmetry by simply declaring the relation symmetric. Then you do not have transitivity: $a^{-1}b$ might be in H, and $c^{-1}b$ might be in H, but together those will not imply that $a^{-1}c \in H$ nor that $c^{-1}a \in H$. For example, both c and a might have a different microstep from all elements of H, while b has the same microstep as some element of H, and all non-identity elements of H might have nonzero first element. (If this explanation was unclear, it should make sense in the context of section 11.)

such that there exist integers j, k such that $g_1 = g_0 + j\lambda + h$ and $g_2 = g_0 + k\lambda + h$.[14]

Despite these unpleasant properties, every time difference is uniquely expressible as the sum of an element of $T = \{(t, 0) : t \in \mathbb{N}\}$ with an element of $N := \{(0, n) : n \in \mathbb{Z}\}$. T and N are isomorphic to the natural numbers and integers respectively, which both do have nice properties such as commutative addition, injective right addition, and existence of at least one common divisor for each pair of elements.

The sum of a logical time or logical time difference g and a logical time smear Δg is another time smear:

$g + \Delta g = \{g + g' : g' \in \Delta g\}$

$\Delta g + g = \{g' + g : g' \in \Delta g\}$

The sum[15] of a logical time smear Δg_1 and a logical time smear Δg_2 is another logical time smear:

$\Delta g_1 + \Delta g_2 = \{g'_1 + g'_2 : g'_1 \in \Delta g_1 \wedge g'_2 \in \Delta g_2\}$

This generalization of the addition from elements of $\mathbb{N} \times \mathbb{Z}$ to elements of $2^{\mathbb{N}\times\mathbb{Z}}$ preserves associativity but not necessarily invertibility.

Multiplication of a time smear or time difference g by a natural number n is the n-fold addition of g with itself.[16]

Suppose b is an object (a port, reaction, or trigger), and we are interested in the question of whether b is present at some tag g.

1. If `a -> Δg! b`, then the presence of a at some time g_0 is a sufficient condition for the presence of b at some time that is in the set $g_0 + \Delta g$. I think of arrows that are followed by `!` as "certainly" arrows because they can guarantee the presence of b within some set of logical times.
2. Suppose b is any object[17] other than `startup`.[18] The following predicate is a necessary condition for the presence of b at tag g: There exists an object a, a tag g_0, and a time smear Δg such that a is present at tag g_0 and (`a -> Δg ? b` or `a -> Δg ! b`) and $g \in g_0 + \Delta g$. I think of arrows that are followed by `!` or `?` as "maybe" arrows because they they are used to determine whether b *might* be present at time g.

[14] We have symmetry here by construction. By the injectivity of left-addition and the fact that $h < \lambda$, if h exists then h is unique for each g_1, g_2 given g_0, λ, so we have transitivity. Reflexivity is obvious.

[15] Due to the relationship between this addition and the Cartesian product, and due to the worst-case multiplicative effect it might have on the cardinalities of finite smears, it is tempting to call it a multiplication. I will persist in calling it an addition to emphasize its role in translating events across time.

[16] Obviously even time differences that are invertible do not form a group – let alone an abelian group – because they may not be invertible on both the left and right. Even if invertibility is used to permit multiplication by negative integers, we therefore cannot have a module.

[17] Physical actions can also become present spontaneously from the perspective of the program, but that can be modeled by drawing a "maybe" arrow with time smear $\mathbb{N} \times \mathbb{N}$ from `startup` to any physical action. Therefore, `startup` is the only exception that need be made.

[18] I will use the (obvious) assumption that `startup` is present at tag g iff $g = (0, 0)$.

A set of "certainly" and "maybe" arrows is said to be invalid if it gives any false guarantees[19] about when some object will or will not be present.

The following are consequences of the definitions of "certainly" and "maybe" arrows:

1. "certainly" arrows are a subset of "maybe" arrows.
2. A "maybe" arrow from a to b with a time smear Δg has the same meaning as a set of "maybe" arrows from a to b, the union of whose time smears is equal to Δg. In particular, it can be replaced with a set of arrows, each corresponding to a time smear of cardinality one.[20]
3. Addition of a "maybe" arrow to a valid set of arrows can never make the arrows invalid. However, the new arrow can make the set of arrows less informative in the sense that it removes counterfactual causality information.[21]
4. Weakening of a "certainly" arrow to just a general "maybe" arrow in a set of arrows can never make the set of arrows invalid.
5. Weakening of the time smear Δg of an arrow to a superset of Δg can never make the set of arrows invalid.
6. A consequence of the previous two items is that any LF program has at least one valid arrow model; a trivial way to find one is to connect every object to every other object using `-> (*, *) ?` arrows.[22] This model is the top of the type lattice corresponding to the given set of objects.[23] However, this does not mean that the

[19]A set of arrows gives a false guarantee whenever the semantics of the corresponding LF program admits an execution trace that includes a counterexample to the sufficient and/or necessary condition semantics of the set of arrows, regardless of whether such an execution trace is realized physically. Note that validity is a property of the set of arrows, and not of any individual arrow. This is because unlike the "sufficient condition" semantics of "certainly" arrows, the "necessary condition" semantics of "maybe" arrows involves all arrows that have a given object as their target, as opposed to just one arrow.

[20]This raises the question of why it is necessary to permit time smears to be sets of pairs of numbers instead of just individual pairs. The obvious reason is that this rule does not work for "certainly" arrows. Another reason is that the discussion presented in section 5 requires that a finite number of arrows can represent the possibility that a reaction can schedule an action at any of an infinite number of times in the future.

[21]Proof: Suppose A is the set of all objects that have "maybe" arrows to an object b and we are interested in whether b might occur at time g. If no objects in A (or any superset thereof) are present at tags g_0 such that g is in the set g_0 plus the time smear of a "maybe" arrow, then b will not be present. Suppose we add some extra "maybe" arrows going to b; then $A' \supsetneq A$ is the new set of all objects that have "maybe" arrows going to b. Given the new set of "maybe" arrows, it is no longer sufficient for all objects in A to be absent (at the relevant tags) in order to conclude that b will be absent; instead, we need all elements of A' to be absent. But before we added the extra arrows, we already had enough information to conclude that b would be absent whenever $A' \supset A$ (or any superset $A'' \supset A' \supset A$ of A') was absent. The counterfactual causality information that we had before adding the "maybe" arrows is a strict superset of the counterfactual causality information that we had after adding the "maybe" arrows.

[22]Proof: Suppose a program has some valid arrow model; I want to show that this implies that the top is a valid arrow model. Apply property 4 to weaken all arrows to "maybe" arrows. Then apply property 5 to weaken all the time smears to $\mathbb{N} \times \mathbb{N}$, which will be a superset of all time smears arising from the explicit arrows of the program. Then there are finitely many distinct arrows, and they are the ones described for the top.

[23]The arrow models possible for the set of objects do indeed form a lattice, by the way, if you consider arrow models obtained from each other using property 2 to be equivalent. The join of any two elements is the arrow model with the intersection of their "certainly" arrows and the union of their "maybe"

arrows can describe the program in a way that is informative; a more informative way of finding an arrow model is discussed in section 5.

7. When you compose two arrows associated with time smears Δg_1 and Δg_2, respectively, the resulting arrow has time smear $\Delta g_1 + \Delta g_2$. The associativity of arrow composition follows from the associativity of time smear addition.

8. Composition of a "certainly" arrow from a to b with a "certainly" arrow from b to c yields a "certainly" arrow from a to c (cuz hypothetical syllogism).

The following is *not* a strictly necessary consequence of the definitions, and it pertains to the addition of extra "maybe" arrows and therefore the loss of (potentially important!) counterfactual causality information.

9. Composition of a "maybe" arrow from a to b with a "maybe" arrow from b to c yields a "maybe" arrow from a to c. That the resulting arrow is not necessarily a "certainly" arrow when the two component arrows are not both "certainly" arrows is clear; furthermore, from our intuition about the relationship between the arrows and some concept of causality, it seems like composition should be possible. However, because this rule removes counterfactual causality information, a more thorough justification for it is warranted. Its usefulness only becomes evident when we want to abstract away certain objects, as discussed in sections 6 and 8.

A valid arrow model of a program is defined to be a set of objects together with a valid set of arrows that is closed under composition.

4 Example

Suppose we have a reaction that is triggered by a timer, and by nothing else:

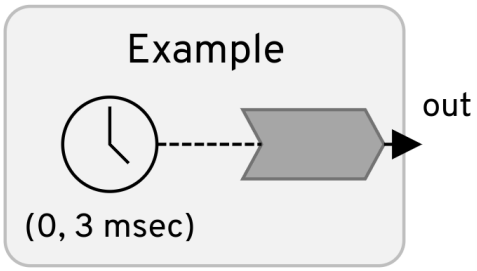

Figure 1: A simple periodic producer.

The following finite set of arrows arrows is explicit in the structure of the program:[24]

arrows, and the meet of any two elements is the arrow model with the union of their "certainly" arrows and the intersection of their "maybe" arrows. If the meet is valid then the two original models are both valid, and if either of the two original models is valid then the join is valid.

[24]The arrow from r to p is not a "certainly" arrow because r is a reaction. However, if the reaction body is analyzed, the arrow from r to p could potentially be strengthened to a "certainly" arrow. Shaokai has shown that control-flow analysis of reaction bodies is not out of reach. Control-flow analysis is, after all, what structured programming is designed for, and it is well-understood and frequently used in practice (e.g., by linters). Moreover, the fact that linters are often wrong does not imply that control-flow analysis cannot be used to rigorously prove trivial facts about important classes of simple programs; instead, it is a reflection of the fact that not everyone has the time to formally specify what they want the linters to prove.

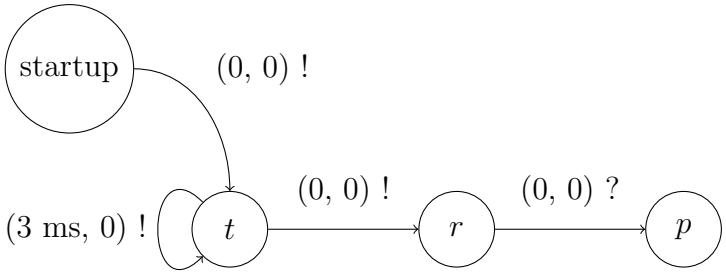

Figure 2: Arrows explicit in the program.

In addition, there are many more arrows that can be drawn by composing this original set of generating arrows. For example, the self-loop can be composed with itself, the arrow going into the timer can be composed with the self-loop, and the self-loop can be composed with arrows going out of the timer. Here are a few of the arrows that result:

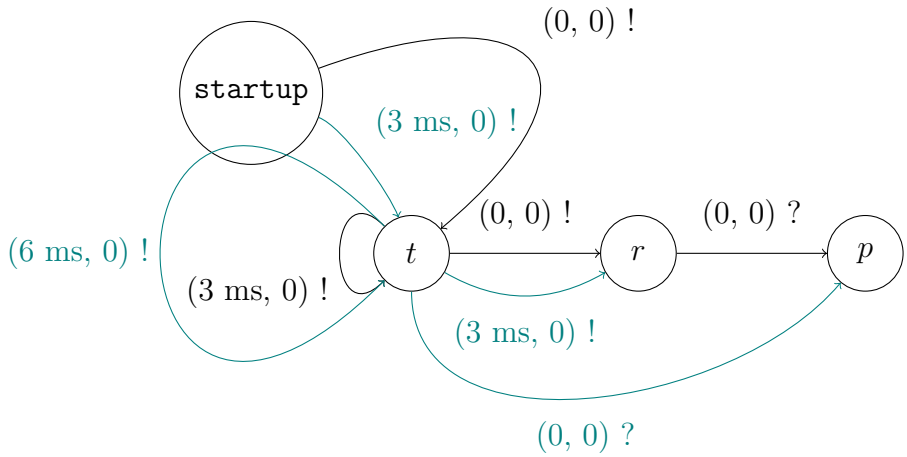

Figure 3: Arrows resulting from composition.

In fact, composition lets us have infinitely many arrows:

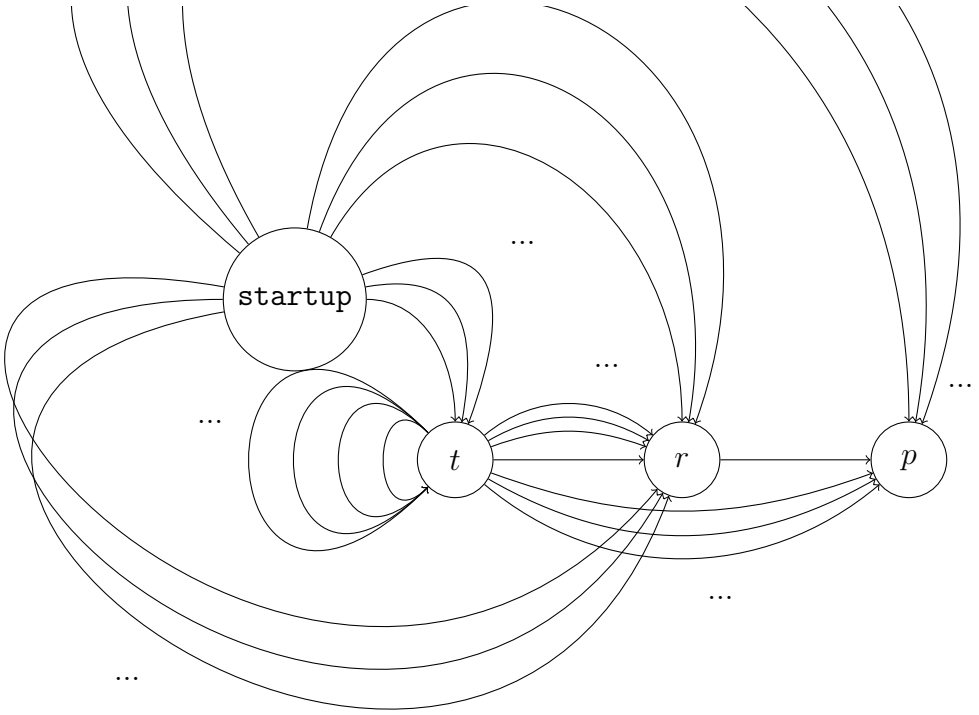

Figure 4: The closure of the original arrows.

The presence of infinite arrows extending from startup to all other objects in the program means that each object other than `startup` might be present at infinitely many moments in time relative to the start time. It is the presence of the self-loop that allows this program to have infinite arrows. In this sense, among all arrows in the original explicit generating set of figure 2, the self-loop holds the special role of encoding the program's periodicity.

So far all of the arrows added beyond those explicit in the program were created by composing the original set of arrows. We may also wish to have valid arrows that are contrary to our intuition about causality, which typically involves physical temporal precedence. These arrows can be inferred using only the original set of arrows when there is only one way for an object to be present at a particular time.[25] For example, there is only one zero-delay[26] "maybe" arrow whose target is p; this means that the presence of r at a tag g is a necessary condition for the presence of p at a tag g, which in turn means that the presence of p at a tag g is a sufficient condition for the presence of r at tag g. This fact can be represented using a "certainly" arrow. A similar line of reasoning justifies a "certainly" arrow from r to t:

[25]It is worth pointing out that such "acausal" arrows would be far more prevalent if reactions were triggered by the logical AND of the presence of their inputs instead of the logical OR.

[26]Unfortunately, it is important that this arrow is zero-delay, for that means that the corresponding arrow that we will infer in the opposite direction can also have zero delay. If the delay were strictly positive, then the acausal arrow would have a negative delay, which is not permitted by our definition of time smears nor by our notion of addition. This problem will be revisited in section 11.

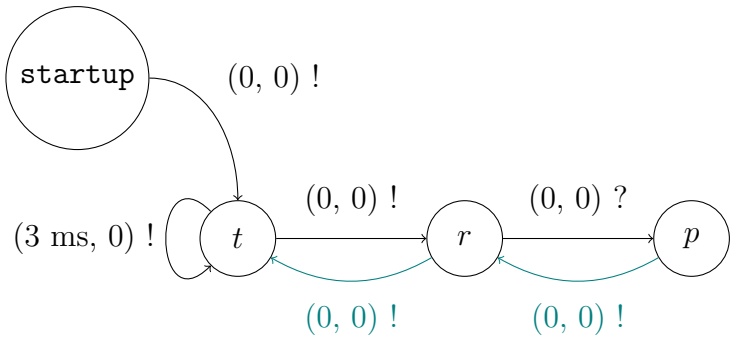

Figure 5: Arrows that are contrary to our intuition about causality.

In light of this example, a problem with such inferred arrows arises. By composing the arrow from p to r with the arrow from r to p, we see that p has a zero-delay "maybe" self-loop. According to the "necessary condition" semantics of "maybe" arrows, this gives p (and with it, r and t) free license to spontaneously be present at any time during program execution, regardless of events in the rest of the program![27] Clearly, counterfactual causality information was lost in the addition of the inferred arrow because of the inferred arrow's "maybe" semantics. Therefore, I define a new type of arrow, symbolized by `!!`, and call it an "acausal" arrow because it lacks the "necessary condition" semantics which is associated with counterfactual causality and which all "maybe" arrows have. Acausal arrows have the following properties:

10. Removal of an acausal arrow from a valid set of arrows can never make that set of arrows invalid.[28]

11. The composition of an acausal arrow with a "certainly" arrow is an acausal arrow.[29]

12. The composition of an acausal arrow with a "maybe" arrow is nothing.[30]

[27]The semantics of "maybe" arrows are violated if p is present at time g_1, but no object a was present at a time g_0 such that there exists a "maybe" arrow from a to p with time smear Δg, and $g_1 \in g_0 + \Delta g$. Now, suppose p has a zero-delay self-loop that is a "maybe" arrow. Then an appearance of p at an arbitrary tag g cannot violate the semantics of "maybe" arrows because p is itself the object a, and $g_0 = g$, and $g \in g_0 + \{(0, 0)\}$.

[28]Compare this to property 3.

[29]This completes a symmetry with the rule for composing "certainly" arrows with "maybe" arrows. In both, the rule is that the arrow resulting from the composition has the intersection of the semantics of the component arrows.

[30]If it bothers you that the two types of arrow do not compose, say that they compose to form a "nothing" arrow, which exists but has no meaning; this continues the pattern that arrows resulting from composition have the intersection of the component arrows' semantics.

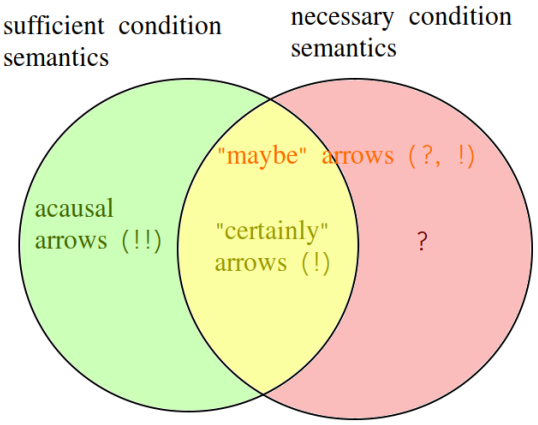

Figure 6: From left to right: `!!`, `!`, and `?` arrows.

5 Finite representability

There are two aspects of arrow models that are not necessarily finite: The set of arrows, and any given time smear which might be associated with the arrows. For these to be analyzed, they must be represented finitely, and ideally, scalably.[31]

In particular, any LF program explicitly presents some finite[32] set of arrows via its port connections and via the declared effects of its reactions.[33] Some additional arrows may optionally be inferred, and all of the (perhaps infinitely many!) remaining arrows can be obtained by composition.

Since time smears have thus far been defined as arbitrary subsets of an infinite set, it is impossible to represent arbitrary time smears finitely.[34] Therefore, the original generating set of explicit arrows that the programmer can specify must be restricted. Even in the presence of reasonable restrictions, however, it is difficult to limit the complexity of representations of compositions of time smears without using the overapproximation of property 5. A specific approach would be to require the time smears of the original, explicit set of arrows to be expressible by a minimum m, an upper bound $M \in (\mathbb{N} \times \mathbb{N}) \cup \{\infty\}$, and an increment g. The time smear would then be $\{m+ng : m+ng < M, n \in \mathbb{N}\}$, and its finite representation would be the triple (m, M, g). An overapproximation[35] of the sum of two time smears (m_0, M_0, g_0) and (m_1, M_1, g_1) would be $(m_0 + m_1, M_0 + M_1, \gcd(g_0, g_1))$ if g_0 and g_1 have any common divisors.

[31] In the sense that representations should ideally grow sublinearly wrt program size.

[32] If we permit recursive instantiation, the set of objects and the set of explicit arrows will not in general be finite. To solve this, guess a limiting arrow model m that is a fixed point with respect to adding another step of recursion and abstracting away the internals of the resulting module by the algorithm of section 8. Then, in order to analyze any program (including the recursive module!) that instantiates the recursive module, use m as the arrow model of the recursive module instead of trying to infinitely unroll it.

[33] In example 1, this was shown in figure 2.

[34] If any time smear can be represented using a finite number of symbols from a finite alphabet then the power set of the natural numbers is countable, which is not true.

[35] Bezout's identity suggests that this should be a close approximation (as measured by set difference, if elements of the set that are close to zero are not given disproportionately large weight) if $-m_0 + M_0$ and $-m_1 + M_1$ are both large in comparison to both g_0 and g_1. Note also that property 5 gives us license to use this approximation without rendering our model invalid.

If g_0 and g_1 do not have a common divisor, then sadly, even this overapproximation will not work, and we may have to use the unreduced representation of the sum smear.[36] g_0 and g_1 have a common divisor if they both have zero as their first element or if they have the same second element and neither has zero as its first element. The common divisor can be found by running the Euclidean algorithm, as usual.[37] This divisibility property will become important again near the end of section 9.

6 Modularity

One way to maintain encapsulation and to limit the complexity of the arrow models of large programs is to model the program's modules with arrow models that avoid any mention of the objects that are encapsulated inside the module. In abstract terms, it is necessary to

- Extract the arrows that are explicitly encoded in the program.
- Optionally infer some other arrows, such as the acausal arrows from objects that have only one way to be present at a given time.
- Use the explicit (and optionally, inferred) arrows to generate all of the other arrows (of which there might be infinitely many![38])
- *Delete* the objects (reactions, triggers) that are encapsulated within the module, so that only global triggers (startup, shutdown) and the ports exposed by the module remain. Delete all arrows that were incident to the deleted objects.
- Encode the arrows that remain using a finite set of generating arrows that all start and end at global triggers or at ports of the reactor.

The problem with the last step in this process is that it seems difficult to provide a good algorithm for finding a finite set of arrows that generate infinitely many arrows of interest. In section 5, I mentioned that the finite generators are given in the structure of the entire program, but that fact is not so easily applicable if we wish to abstract part of the program away.

Property 6 provides an easy escape hatch, but it provides a fully uninformative model of the program.

In search of a better solution, I return to the example of section 4. We can see by inspection that the following two arrows generate the infinite set of arrows involving `startup` and p:

[36] Obviously elements of the additive closure of the restricted set of possible generating smears will still be finitely representable (and countable) – the only problem is that it will not be possible to make the length of the expression grow sublinearly with the number of generating smears in the sum using the means explored here.

[37] In the first case, time differences with zero as their first element are isomorphic to the natural numbers (and will be isomorphic to the integers, should we later decide to allow the second element to be negative), so of course the Euclidean algorithm works. Similarly, for any n, the set $\{(t, n) : t \in \mathbb{Z}_+\} \cup \{(0, 0)\}$ is isomorphic to the natural numbers, so the Euclidean algorithm works for the second case as well. The isomorphism $\mathbb{N} \to \{(t, n) : t \in \mathbb{Z}_+\} \cup \{(0, 0)\}$ maps 0 to $(0, 0)$ and it maps t to (t, n) for $t > 0$.

[38] Clearly this is not an implementable algorithm. The purpose is just to convey an idea of what is being asked.

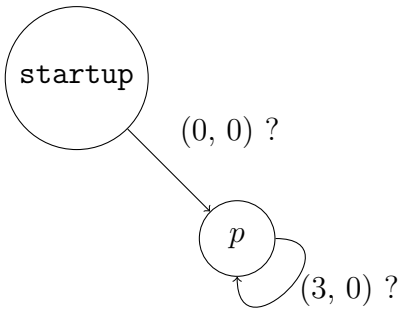

Figure 7: A reduced representation of the program of figure 1.

In this case, the self-loop at p is a composition of a "certainly" arrow from p to r, a "certainly" arrow from r to t, a "certainly" arrow from t to r, and a "maybe" arrow from r to p.

In order to obtain the infinite set that we needed, we seem to have "commuted out" the self-loop at the timer t and migrated it to the port p. This example can serve as inspiration for the algorithm of section 8. Section 8 uses another algorithm which could be useful in its own right, and which I will describe first.

7 Algorithm for canonical representation

There should be a way to compute a canonical representation for an arrow model that has a finite set of generating arrows. This might make it easier to determine whether two arrow models of a program (e.g., one that is specified by a human, and one that is machine-computed) are equivalent,[39] or more generally, whether the validity of one arrow model implies the validity of the other.

I must make two significant caveats to the claim that the representation proposed here is unique. The first is that although it is a unique representation of a given set of arrows, it is not a unique representation of the meaning of such arrows.[40] The second is that even this weak kind of uniqueness requires arrows not to form zero-delay cycles; this requirement is incompatible with the extensions proposed in section 11.[41]

The idea is to represent a set of arrows using only the minimal arrows of the set. The existence of an algorithm for finding the minimal arrows will prove that they are finite in number for any implementable program. I will also argue that they generate all arrows in the set.

Minimality of arrows is defined as follows: An arrow a->b is minimal if it cannot be represented as a sequence of existing arrows, at least one of which is a self-loop not equal to a->b, such that the same sequence of arrows with the self-loop elided is not equal to a->b. For example, in figure 3, the arrow from t to itself with time smear $\{(3\text{ms}, 0)\}$ is minimal, as is the arrow from t to p with time smear $\{(0, 0)\}$, but the arrow from t to

[39] If they imply each other's validity then they are equivalent.

[40] Two distinct sets of arrows may have the same meaning in the sense of implying each other's validity, e.g. due to property 2.

[41] In section 8, uniqueness is not used. Instead, we only use the property that the arrows in the output of this algorithm are a finite superset of the set of minimal arrows. This property can be guaranteed by modifying this algorithm to omit the pruning step.

itself with time smear $\{(6\text{ms}, 0)\}$ is not minimal, nor is the arrow from `startup` to t with time smear $\{3\text{ms}, 0\}$.

I consider an arrow that results from composing a finite set of generators[42] with each other. Such an arrow can be represented as a finite sequence of those generators, in the order in which they were composed. Such a sequence consists of loops that have no repetition except in the first and last object visited, interspersed with straight-line sequences that have no repetition at all. Because there are finitely many generators, we can find all such no-repetition loops[43] and summarize them with the single self-loop arrow that is their composition. We can also find all no-repetition sequences and summarize them with direct arrows from the object at the beginning of the sequence to the object at the end of the sequence. The resulting "summary" arrows are a superset[44] of the minimal arrows of the arrow model. This finite set of arrows arising from no-repetition sequences and self-loops can be pruned of arrows that are not minimal. Doing so preserves the property that the set generates all arrows.[45]

The following is the canonical representation of one of the valid arrow models of the example program of section 4:

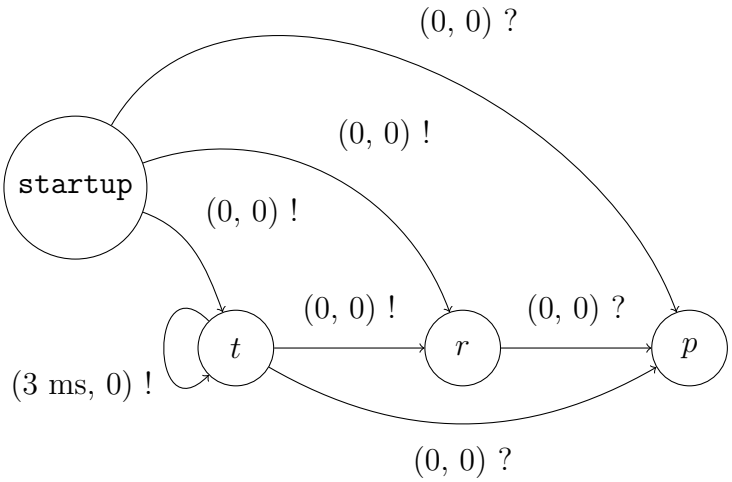

Figure 8: A canonical representation of the arrow model of figure 4.

[42]The finite generators that one would typically start with would be the arrows explicit in the program, as in figure 2.

[43]The number of such no-repetition self-loops and sequences is upper-bounded by the factorial of the number of generators, which is finite.

[44]We have obtained all arrows that can be generated without self-loops, but the minimal arrows are the arrows that cannot be generated with self-loops. The only arrows in the latter set but not the former would be those arrows which cannot be generated at all, but we are not interested in those.

[45]The arrows that generate `->` cannot be generated by `->` if there are no zero-delay cycles. (It is not too restrictive to require that "maybe" arrows never form zero-delay cycles.) Therefore, given the assumptions used in this section, the closure wrt composition of the set cannot be diminished by removing `->` when `->` can be generated from other arrows.

8 Algorithm for abstracting objects out of an arrow model

This algorithm will require some way to "commute" arrows with each other. Sadly, we *can't* commute arrows with each other: As mentioned in 3, when arrows are composed, their time smears add, and time smears do not commute in general.

The next best thing to commutation of a time difference g_0 with another time difference g_1 is existence of a time difference $C(g_0, g_1)$ such that $g_0 + g_1 = g_1 + C(g_0, g_1)$. Such $C(g_0, g_1)$ is unique if g_1 is expressible as an element of $\mathbb{N} \times \mathbb{Z}$ and exists if g_1 and g_0 are elements of $\mathbb{N} \times \mathbb{Z}$.[46] Uniqueness follows from the injectivity of left-addition of an element of $\mathbb{N} \times \mathbb{Z}$, which was proved in footnote 11.[47] Existence can be proved as follows: Suppose $g_0 = (t_0, n_0)$ and $g_1 = (t_1, n_1)$. Since our addition acts on the first element of time differences like integer addition, which is commutative, the first element of $C(g_0, g_1)$ is t_0. To determine the second element, there are four cases:

1. $t_0 \neq 0$ and $t_1 \neq 0$. Then the second element of $C(g_0, g_1)$ is n_1.
2. $t_0 \neq 0$ and $t_1 = 0$. Then the second element of $C(g_0, g_1)$ is $n_0 + n_1$.
3. $t_0 = 0$ and $t_1 \neq 0$. Then the second element of $C(g_0, g_1)$ is 0. Note that in this case $C(g_0, g_1)$ is the identity element and so $g_0 + g_1 = g_1$, regardless of n_0.
4. $t_0 = 0$ and $t_1 = 0$. Then the second element of $C(g_0, g_1)$ is n_0.

Obviously, if g_0 and g_1 are both in the subset of the time differences which is isomorphic to the natural numbers, or if they are both in one of the groups which are isomorphic to the integers, then $C(g_0, g_1) = g_0$.

As shorthand, let us define $C(\Delta g_0, \Delta g_1)$, where Δg_0 and Δg_1 are time smears, to be $\{C(g_0, g_1) : g_0 \in \Delta g_0 \wedge g_1 \in \Delta g_1\}$.

8.1 Algorithm

Start with the representation computed in section 7. Then greedily remove the objects that need to be abstracted away, one at a time. When you remove an object from the arrow model, you also remove its incident arrows, so care is required to ensure the object is "safe to remove" in the sense that removing it cannot result in an invalid arrow model.[48]

An object a that has no self-loops is already safe to remove. Suppose some implicit arrow $b_1 \text{->}_0 b_n$ is generated by a composition $b_1 \text{->}_0 b_n = b_1 \text{->}_1 b_2 \text{->}_2 \cdots \text{->}_{i-1} b_i \text{->}_i a \text{->}_{i+1} b_{i+2} \text{->}_{i+2} \cdots \text{->}_{n-1} b_n$. Then the canonical representation will include the arrow $b_i \tilde{\text{->}} b_{i+2} := b_i \text{->}_i a \text{->}_{i+1} b_{i+2}$ (i.e., the composition of the arrow $b_i \text{->}_i a$ entering a and the arrow $a \text{->} b_{i+2}$

[46] These requirements are problematic in the context of section 11. To make them compatible, we would need additional commutation rules, the development and application of which is not in the scope of this document.

[47] In the notation of 11, by left-adding the additive left-inverse of g_1 to both sides, we must conclude that $C(g_0, g_1) = -g_1 + (g_0 + g_1)$.

[48] In practice, the way to ensure that removing an object a will not result in an invalid arrow model is to assume that the starting arrow model is valid, and then to ensure that after a and its incident arrows are removed from the arrow model of the program, the remaining explicit arrows will still generate all of the arrows involving objects not removed from the model. Properties 3 and 4 apply.

leaving a) as one of its finite set of explicitly represented generators (if $b_i \tilde{\to} b_{i+2}$ is minimal), or it will include arrows not involving a that generate $b_i \tilde{\to} b_{i+2}$ – in particular, one or more self-loops at b_i and an arrow from b_i to b_{i+2}, or an arrow from b_i to b_{i+2} and one or more self-loops at b_{i+2}.[49] Therefore, even after removing a, the implicit arrow $b_1 \to_0 b_n$ can still be obtained as $\to_0 = b_1 \to_1 b_2 \to_2 \cdots \to_{i-1} b_i \tilde{\to} b_{i+2} \to_{i+2} \cdots b_n$.

In the preceding argument, I used the facts that $b_i \neq a$ and $b_{i+2} \neq a$; if these were not true, then the inductive step would not necessarily have reduced the number of occurrences of a in the composition.[50] Therefore, an object a that does have a self-loop may not be safe to remove. However, it can be made safe to remove. The idea is to "commute out" the self-loop.

Suppose a has a self-loop with time smear Δg_a. For each object b and time smear Δg such that there exists an arrow from a to b with smear Δg, add a "maybe" self-loop from b to b with time smear $C(\Delta g_a, \Delta g)$ if $C(\Delta g_a, \Delta g)$ is not the identity time smear $\{(0,0)\}$, and do nothing otherwise.

I must prove that once this is done, a is safe to remove. Suppose some implicit arrow $b_1 \to_0 b_n$ is generated by a composition

$$\to_0 = b_1 \to_1 b_2 \to_2 \cdots \to_{i-1} b_i \to_i a \to_{i+1} \cdots \to_{j-2} a \to_{j-1} b_j \to_{j+1} \cdots \to_{n-1} b_n$$

If we added an arrow $b_j \to_C b_j$ with the time smear $C(\Delta g_a, \Delta g)$, then $a \to_{j-2} a \to_{j-1} b_j$ can be safely replaced with[51] $a \to_{j-1} b_j \to_C b_j$. If we did not add such an arrow, then $a \to_{j-2} a \to_{j-1} b_j$ can be safely replaced with[52] $a \to_{j-1} b_j$.

In either case, the composition of arrows can be replaced with another composition of arrows in which the length of the sequence of self-loops starting and ending at a is reduced by one. Eventually, we reach the base case where no self-loops at a remain. This case has already been discussed, and so we can conclude that after possibly adding some "maybe" self-loops to all objects downstream of a, a can be made safe to remove.

9 Relationship to some of Shaokai's comments during and after the 10/24 meeting

I start by re-phrasing some of the general ideas that I think he mentioned, just to see if I understood.

For a given program there are many ways to define a time evolution operator. For example, it is possible to imagine a time evolution operator that evolves the system

[49] The point here is that the self-loops must be at b_i or b_{i+2} and not at a since a has no self-loops.

[50] For example, if $b_i \to_i a \to_{i+1} b_{i+2}$ were replaced with an arrow from b_i to b_{i+2} followed by a self-loop at b_{i+2}, and $b_{i+2} = a$, then we would have gone from two visits of a to two visits of a.

[51] The two are not necessarily equal because there may have been information loss, i.e., the resulting arrow will be a "maybe" arrow even though the previous arrow might have been a "certainly" arrow. However, this operation will not result in an invalid model.

[52] There is a subtlety here, which is that $a \to_{j-2} a$ might have been a "maybe" arrow while $a \to_{j-1} b_j$ might have been a "certainly" arrow; in that case, it is not obvious from property 4 that this transformation is safe. However, the entire composition of arrows is in that case strictly weaker in its guarantees than another composition which does not include the "maybe" self-loop at a, and this other composition must also exist and be valid. It is this other composition which guarantees that our transformation will not create an invalid model.

forward by a fixed amount of logical time, physical time, or number of tags executed. If I understood Shaokai, the time evolution operator M that we are interested in is of the third type.[53] Let us define M_k to be the time evolution operator that evolves the system forward by k tag executions, regardless of the logical or physical time spacings between the executed tags. The eigenvalues of M_k are a superset of the eigenvalues of M_j if $j|k$ because in that case M_k is a power of M_j.

If I understood Shaokai, in a program with n objects, a state is defined to be a vector of n logical times corresponding to the times at which each object is *scheduled* to next be present. For the purpose of this section, I will prefer to use the times when each object *actually will* be present,[54] and I will assume that the state somehow contains enough additional information for the state evolution operator to be a function (which maps its inputs to unique outputs).[55]

Another point is that Shaokai was talking about physical times, if I understood him, whereas this document discusses logical times. However, both seem relevant to the problem of pre-computing static schedules.

An important concept that Shaokai discussed is that of an eigenspace of the time evolution operator of a system. If I understand, we are interested in an eigenspaces because they are a special kind of invariant subspace[56] of the state space, and invariant subspaces allow us to prune the state space to a simpler one in which the time evolution operator still makes sense.[57] More generally, even in programs that do not reach eigenstates, the most interesting of all invariant subspaces is the smallest[58] one that is ever reached in a program's execution, for that is the simplest one that we have any reason to optimize for.[59]

The following are true about arrow models of LF programs and invariant subspaces:

1. According to any valid definition of the state space, the state space V of an LF program is trivially invariant wrt any time evolution operator. In the context of arrow models, let us define the state space[60] as the set of all n-length lists of logical times[61] v satisfying the following properties:

   (a) For all i, the ith entry v_i of v is either ∞ or an element of the time smear of

[53] If we hold an ambitious goal of pinning tag executions to precise physical times, where physical times are measured in clock cycles on a PRET machine, then the time evolution operator that he discussed could also be of the second type.

[54] Because the arrow model of a program only describes what is present, not how or why, it is ill-suited for use with the first definition, so for convenience I will use the second. I hope the resulting observations will be at least somewhat relevant.

[55] The existence of a such a definition of state is another way to describe our claim of determinism.

[56] A subspace W of the state space V is invariant wrt time evolution if no element of $V \setminus W$ is reachable from W.

[57] "Makes sense" in the sense that when the state evolution operator is restricted to the invariant subspace, it does not map anything outside of its domain.

[58] Wrt the partial ordering imposed by set containment.

[59] Furthermore, as Shaokai explained, it is usually sufficient to optimize only for this subspace since it includes all states that will be visited for infinitely many tag executions.

[60] Note that this definition of the state space might already be a much smaller space than other definitions of the state space that do not use as much of the structure of the program; conversely, an even smaller space might also be valid for some programs.

[61] This is not a vector space, so I avoid the terminology of "vectors."

an arrow extending from `startup` to the ith object

(b) There exists no "certainly" arrow from `startup` to the ith object with a time smear all of whose elements are greater than or equal to $\min_j v_j$ but less than v_i

2. The set $W_{\max,\,g}$ of elements of V with all entries greater than some fixed time g is trivially an invariant subspace of all time evolution operators corresponding to timesteps greater than or equal to zero.

3. The set $W_{\min,\,g}$ of the elements of W_g whose ith entries are either ∞ or the sole element of the time smear of a "certainly" arrow extending from `startup` to the ith object is not an invariant subspace of any time evolution operator, in general; however,

4. If $W_{\min,\,g} = W_{\max,\,g} =: W_g$, and the totally ordered[62] sequence[63] of all elements of W_g is n-periodic with respect to the equivalence imposed by "λ-reachability from some time g_0" in the terminology of 3, then W_g is an eigenspace of M_n.

Perhaps the most practical way to use an arrow model to determine the worst-case[64] schedule that should be optimized for is to, for each object b in the program, run the algorithm of section 8 to abstract away all objects other than b and `startup`. Then there are a few possibilities, each covering a broader set of cases than the previous:

1. If b has no self-loop in the resulting arrow model, or if b has a self-loop with zero time and the program is expected to continue for nonzero time, then b is not recurrent and can be ignored.

2. If[65] all self-loops of b in the resulting arrow model have some GCD d, then for each minimal arrow from `startup` to b with time smear[66] Δg, plan for the possibility that b occurs periodically with period d starting at all times in Δg. Then it should be possible to use any established techniques[67] for scheduling a set of periodic tasks with known periods and start times.

3. In any case, the first elements of all explicit self-loops of b will certainly have some GCD d, and the second elements of all explicit self-loops will also have a GCD

[62]States in this context are ordered according to their least entry (where the least entry is an element of $\mathbb{N} \times \mathbb{N}$, which we order lexicographically). In this context, no two states have the same least element, so all states are comparable. The reason why no two states have the same least element is that property 2 in the definition of the state space ensures that when $W_{\min,\,g} = W_{\max,\,g}$, a state with least element g is the unique vector that lists the first time when each object is present at or after time g.

[63]The term "sequence" here implies bijection with the natural numbers, which we do indeed have for all programs that can execute on an existing computer.

[64]The "worst case" here is when all objects that might be present are present. My use of this phrase is based on the implicit assumption that a decrease in the number of present objects will not cause deadline misses. Although this may be a poor assumption for some popular dynamic scheduling algorithms, I consider it to be a reasonable assumption when the schedule is computed offline.

[65]Although it might seem like this is a special case given the infinite number of possible microsteps, this case could be quite common. There might be only a few paths from b back to itself; furthermore, if those paths all have a common suffix that includes a time smear whose time differences all have nonzero first element, then the possible microsteps of all the resulting arrows will all be the same.

[66]If Δg is infinite, then it will be difficult in general to create a static schedule involving the object b, so assume that Δg contains only one or a few elements.

[67]I know little about such techniques, but I assume they exist.

e. Consider execution of b at a *tag* to be a task while execution of b at all tags corresponding to a given *time* may in general be a "supertask" (i.e., a task that consists of many tasks).[68] Then we have a supertask which we must pessimistically assume to have period $(d, 0)$, and in the schedule for executing the supertask, we may consider execution of b to have period $(0, e)$. Of course, this case can also be divided into subcases, e.g. where the "supertask" actually is known to be finite with a known bound because the first element of all elements of all smears of the self-loops of b is nonzero.

This strategy of abstracting away all objects other than the one of interest and `startup` also yields a straightforward way of determining whether any two objects can co-occur. Specifically, if we know that b_0 might be present at times $\{(t_0, n_0) : t_0 \in X_0 \pmod{d_0} \wedge n_0 \in Y_0 \pmod{e_0}\}$ and b_1 might be present at times $\{(t_1, n_1) : t_1 \in X_1 \pmod{d_1} \wedge n_1 \in Y_1 \pmod{e_1}\}$, then b_0 might be present at the same time as b_1 if $X_0 \cap X_1 \neq \emptyset \pmod{\gcd(d_0, d_1)}$ and $Y_0 \cap Y_1 \neq \emptyset \pmod{\gcd(e_0, e_1)}$. We can then build a graph where two objects are connected if they might be present at the same time. This could be useful for limiting the state space. For example, suppose we wish to have a static schedule for the execution of a tag when various sets of actions and triggers are present, like the schedules that the quasi-static scheduler uses currently. Then we need only provide a schedule that can handle the presence of subsets of each strongly connected component in this graph since the set of present actions and triggers will always be a subset of some strongly connected component.[69]

Furthermore, because this discretizes the times at which an object b can be present, it permits the inference on finite bounds on the number of events involving b in the event queue **if** there is an upper bound L on the maximum time into the future when b can be scheduled **and** a finite[70] multiple of the minimum spacing imposed by the discretization[71] is greater than L.[72]

10 Edward's comments on discussion #1307

Edward had a couple of comments on the discussion post.

[68] Apparently some people use "supertask" as a technical term such that it must have countably infinite component tasks. It is true that LF programs will in general admit the possibility that a countably infinite number of tasks *might* be scheduled at any given time, which is consistent with the assumption used here that we pessimistically consider there to be a task whenever something *might* need doing. We are able to execute with bounded lag precisely when there exists a last task that takes nonzero time, i.e., when all tasks following that last one do not actually need doing. This is the only way for a standard computer, which does work in discrete chunks (e.g., instructions), to complete supertasks.

[69] If this were implemented, then the strategy would probably be to check the strongly connected components from smallest to largest, in order of containment, until the set of all present objects intersected with the complement of the strongly connected component is empty, as can be determined by a fast bitwise AND operation.

[70] If $L \geq (1, 0)$ and the minimum spacing is strictly less than $(1, 0)$, then this condition will not hold.

[71] To be explicit: If all microsteps when b is present are equivalent modulo e then the minimum spacing is at least e, and if all times when b can be present are equal to some time g plus a multiple of some time difference λ, then the minimum spacing is at least λ.

[72] As with the case of so-called "supertasks" which actually have a known bound on tasks that actually need doing, we can refine these rules to bound the number of events involving b in the event queue even when b has nonzero self-loops whose second elements are not equal.

One was that it is related to this paper, which discusses a behavioral type system. This behavioral type system explicitly models state, and it is similar in flavor to Hoare's CSP formalism in the sense that concurrent processes interlock with each other (except through transitions instead of states). This discussion post is different from that because it avoids explicitly modeling state and is therefore too simple to represent the kinds of handshaking that a behavioral type system could model using FSMs. In particular, it provides no way to detect illegal states – states in which one component cannot accept some input which it might receive from another.

Given the communities from which LF takes inspiration, it might seem strange to argue that the way of modeling LF programs described here is closer to the LF core language than one based on concurrent finite-state machines. I perceive LF to be a purely declarative domain-specific language for describing correlations and timing relationships between activity in different components. According to such a perspective on LF, the core language of LF is the connection statement, which intrinsically has little relation to any concept of state.[73] Furthermore, there is practical heuristic reason to desire similarity between the model and the "core language." If the model is similar to the code, then concise code is likely to be accompanied by a simple model. For example, a model based on composition of finite state machines can become very complex for a parent reactor whose child reactors can legally be in any combination of states because they are not tightly coupled. This can happen even if the code for the parent reactor itself looks like it should expose a simple interface. In contrast, a model that only uses arrows between ports that are declared in the code as inputs and outputs of the parent reactor might, according to this heuristic, have a good chance[74] of being simple and easy to infer from the "core language."

Another comment from Edward was that type systems can be prohibitively cumbersome, that type inference might be required, and that the need for type inference can heavily restrict the kinds of type systems that are possible. In particular, he pointed out that language designers are sometimes constrained to base their type systems on Hindley-Milner in order for type inference to be practical; this is one paper that discusses such issues. Perhaps I should not have described this idea as a type system. The idea discussed here does not really require type inference because when the programmer does not bother with an arrow model of the program, the compiler need not bother with one either.[75] The program will still work because no arrow model of the program is required to produce a valid executable. Furthermore, when an explicit reactor definition is given, sections 7 and 8 provide ways to compute either a complete or a simplified arrow model without programmer intervention.

[73] This perspective is mostly arbitrary, but because some people might find it strange, I will attempt to justify it. Connection statements can be annotated with after delays, which means they can represent "certainly" arrows with time smears of cardinality one. They can also represent "maybe" arrows if their source is a reaction. Timers can be implemented using after delays. Logical actions cannot – they are necessary in order to represent arrows with time smears with cardinality greater than one. However, arrows implemented using logical actions can be thought of as a generalization of those which are implemented by connection statements with "after" delays, so I regard the non-implementability of logical actions using connection statements alone to be more an accident of syntax than a fundamental consequence of design intent.

[74] The purpose of the preceding sections has been to make this claim precise.

[75] This is not a particularly unusual property even for systems that we commonly refer to as "type systems" – for example, Java uses type erasure with generics, and static types used in TypeScript also are not required because TypeScript compiles to dynamically typed JavaScript anyway.

11 Functional correctness

I previously asserted that timing information encoded by the arrows would be useful for determining functional correctness. I here endeavor to justify that claim.

First, it is useful to revisit the idea of acausal arrows. If the meaning of such arrows is to be generalized beyond zero-delay arrows, it is necessary to extend the definition of a time smear such that an arrow can point backward in time.[76]

To do this, give any time difference g an additive left-inverse[77] $-g$. Instead of our previous definition of time differences as $\mathbb{N} \times \mathbb{Z}$, we should define time differences as the closure under addition[78] of elements of $\mathbb{N} \times \mathbb{Z}$ and their left-inverses; for example, time differences will include finite-length expressions such as $(0, 1) + -(1, 2)$.

The introduction of left-inverses is consistent with the associativity of addition. One way to explain this is to identify time differences with functions[79] that act on elements of $\mathbb{N} \times \mathbb{Z}$ and to identify left-addition with left-composition.[80] Then associativity is obvious because function composition is associative, and the existence of left-inverses is also immediate from the injectivity of left-addition by a time smear.

[76]The way I do this might seem strange to some people. To illustrate why it is necessary, here is a seemingly simpler approach that does *not* work: Define time smears as subsets of $\mathbb{Z} \times \mathbb{Z}$ instead of subsets of $\mathbb{N} \times \mathbb{Z}$. Suppose that, in the initial set of arrows explicit in the program, some object b is the target of only one "maybe" arrow with source a and time smear $\{(t, n)\}$, where t might be strictly positive. You can add an acausal arrow with time smear $\{(-t, z) : z \in \mathbb{Z}\}$ to reflect the fact that tags with all possible microsteps are in the preimage of the function that adds $\{(t, n)\}$. So far so good. Then, compose the acausal arrow on the right with an arrow from a to b' with time smear $\{t, n\}$ where $k > 0$ to obtain an arrow from b to b' with time smear $(0, n)$. Oops! If all the arrows involved were "certainly" arrows or acausal arrows, then the new arrow from b to b' means that b' is guaranteed to be present exactly n microsteps after b, but in fact b' is guaranteed to be simultaneous with b. Recall also that the proof of associativity used the fact that time differences were elements of $\mathbb{N} \times \mathbb{Z}$ instead of elements of $\mathbb{Z} \times \mathbb{Z}$.

[77]This means $-g + g = (0, 0)$, but we do not necessarily have that $g + -g = (0, 0)$. Our extension of addition will preserve the fact that $(0, 0)$ is the unique additive identity.

[78]Note that we are *not* using the closure under left-inversion. There is no guarantee that all left-inverses have left-inverses; instead, there is only a guarantee that they have right-inverses.

[79]There is a potential source of confusion here wrt left vs. right: Left-addition corresponds to left-composition, which is intuitive. However, we like to think of right-addition as the operation that we perform as we follow arrows in the direction in which they point, which makes right-addition together with Polish function notation more intuitive. We are not interested in right-addition because it does not have the same nice properties.

[80]This map is injective (its left-inverse is the operation of applying the resulting function to $(0, 0)$), hence bijective onto its range, giving us license to identify elements of its domain with those of its range. Furthermore, this map respects left-addition. It is not hard to check that $(t_0, n_0) + (t_1, n_1)$ is the function that maps (t_2, n_2) to $(t_0 + t_1 + t_2, n_2)$ if t_2 is nonzero and to either $(t_0 + t_1 + t_2, n_1 + n_2)$ or $(t_0 + t_1 + t_2, n_0 + n_1 + n_2)$, depending on whether t_1 is nonzero, regardless of whether we add $(t_0, n_0) + (t_1, n_1)$ before considering them as functions or consider them as functions and then compose them.

Here is an example of the resulting arithmetic:

$$
\begin{aligned}
&(((0,1) + -(3,3)) + (1,2)) + (1,2) \\
&= (0,1) + -(3,3) + ((1,2) + (1,2)) && \text{Associativity} \\
&= (0,1) + -(3,3) + (2,2) && \text{“Regular” addition} \\
&= (0,1) + -((2,2) + (1,3)) + (2,2) && (2,2) + (1,3) = (3,3) \\
&= (0,1) + -(1,3) + -(2,2) + (2,2) && -(1,3) + -(2,2) + (2,2) + (1,3) = 0 \\
&= (0,1) + -(1,3) && -(1,3) + -(2,2) + (2,2) + (1,3) = 0 \\
&= (0,1) + -(1,3) && \text{Cannot reduce } (0,1) + -(1,3)
\end{aligned}
$$

The ordering structure of time differences can be extended. Left-inverses are compared according to the reverse of the ordering of the time differences of which they are inverses.[81] Time differences of the form $g + -h$, where g, h are pairs, cannot be represented as pairs or their additive inverses in general, so comparisons involving such time differences must be defined separately. If $h > g$, then $h = g + f$ for some left-invertible f. Therefore $-h = -f + -g$, in which case $g + -h = g + -f + -g$. Otherwise, if $h < g$, then $g = h + f$ for some f, and $-g = -f + -h$, and $g + -h = g + f + -g$. Now, let us declare that a time difference f (positive or negative) conjugated by $g \in \mathbb{N} \times \mathbb{Z}$[82] to obtain $g + f + -g$ is less than everything greater than f and greater than everything less than f.[83] This exhaustively determines the ordering of time differences because time differences of the form $g + -h$ are the only ones that cannot be simplified to either an element of $\mathbb{N} \times \mathbb{Z}$ or an inverse thereof. [84]

11.1 Example: Logical simultaneity

Suppose we implement a fork-join pattern:

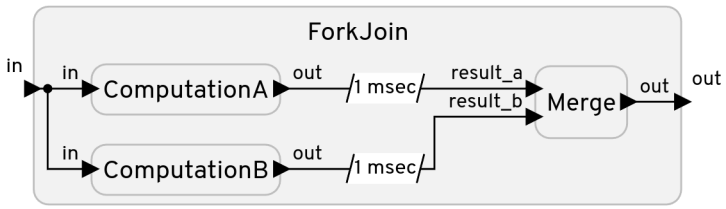

Figure 9: A fork-join pattern.

[81] Note that this creates no contradiction in the case of left-inverses of time differences of the form $(0, z)$, since it agrees with the ordering of the integers.

[82] If you conjugate the other way as $-g + f + g$ (or equivalently, if g is a left-inverse of an element of $\mathbb{N} \times \mathbb{Z}$), we have a contradiction, so we must forbid that. Suppose f is in $\mathbb{N} \times \mathbb{N}$. Then $f + g \geq g$, so $f + g = g + h$ for some h. This h could be strictly greater than or strictly less than f, but $-g + f + g = h$.

[83] This implies that $g + -g$ is either equal to zero (if $g = 0$) or incomparable to zero.

[84] Indeed, expressions of the form $-g + h$, where g and h are in $\mathbb{N} \times \mathbb{Z}$, can be simplified. If $h < g$, then $g = h + f$, and $-g = -f + -h$, so $-g + h = -f$. If $h \geq g$, then $h = g + f$ for some f, so $-g + h = f$. Since the ordering of $\mathbb{N} \times \mathbb{Z}$ is total, these two cases are exhaustive, and so we are done. Now let us consider an expression $g_1 + g_2 + \cdots + g_n$ in the additive closure of the elements of $\mathbb{N} \times \mathbb{Z}$ and their left-inverses where $n \geq 3$. Then either $g_{n-1} + g_n$ is reducible, or $g_{n-1} \in \mathbb{N} \times \mathbb{Z}$ and g_n is a left-inverse of an element of $\mathbb{N} \times \mathbb{Z}$. In the former case, we have already made the inductive step, and in the latter case, $g_{n-2} + g_{n-1}$ can certainly be reduced. This proves that all elements of the additive closure of the elements of $\mathbb{N} \times \mathbb{Z}$ and their left-inverses are equal to a sum of at most two elements of $\mathbb{N} \times \mathbb{Z}$ and their left-inverses. Furthermore, the only such expressions of length 2 that cannot be simplified to one of length 1 are those of the form $g_1 + -g_2$.

The `Merge` reactor can be simplified and optimized if we know whether `result_a` and `result_b` will be present simultaneously. The advantage is not merely in optimizing out a branch that depends on an `is_present` field. If `result_a` is a large data structure, and the `Merge` reactor is not guaranteed that `result_b` will be simultaneous with `result_a`, then it might have to perform a deep copy[85] of `result_a` to save while it waits for `result_b`. Such a precaution would be a performance red flag and an unnecessary responsibility for the programmer, even if the copy routine is never executed in practice.

The problem is that it is not obvious given the information provided so far that `result_a` and `result_b` will be logically simultaneous. For example, `ComputationA` might have an internal delay of several milliseconds, whereas `ComputationB` might be logically instantaneous. To obtain more information about the two modules, suppose we run the algorithm of section 8 on each of `ComputationA` and `ComputationB` and obtain the following results:

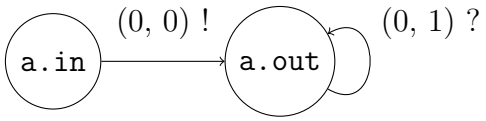

Figure 10: Arrow model of `ComputationA`.

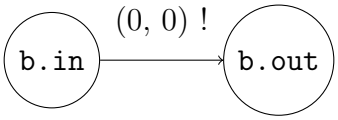

Figure 11: Arrow model of `ComputationB`.

Then, because `ComputationA` and `ComputationB` only involve microstep delays, it is guaranteed that `result_a` and `result_b` will be simultaneously visible to `Merge` at the time when `ForkJoin.in` was present plus $(1\text{msec}, 0)$. Note that it does not matter that `ComputationA` could have produced its final output after any number of microsteps, nor that `a.out` could have been present infinitely many times.

To prove this formally, it suffices to show that there exists a valid arrow model of `ForkJoin` with zero-delay arrows with "sufficient condition" semantics[86] going between `result_a` and `result_b` – this is the precondition that the author of the `Merge` reactor would have to write in order to justify his simple implementation with minimal runtime checks. And indeed, we have

`result_b -> {`$-(1\text{msec}, 0)$`}!! ComputationB.out -> {`$(0,0)$`}!! ComputationB.in -> {`$(0,0)$`}!! ForkJoin.in -> {`$(0,0)$`}! ComputationA.in -> {`$(0,0)$`}! ComputationA.out -> {`$(1\text{msec}, 0)$`}! result_a = result_b -> {`$(0,0)$`}!! result_a`

because $\{-(1\text{msec}, 0)\}+\{(0,0)\}+\{(0,0)\}+\{(0,0)\}+\{(0,0)\}+\{(1\text{msec}, 0)\} = \{-(1\text{msec}, 0)\}+$

[85] It is possible that we could guarantee that ports will always hold the data that they had when they were most recently present. However, if one thinks of an `is_present` field as a software analogue of a valid bit, it seems non-obvious that such semantics will be optimal for the implementations of LF that we might wish to explore. Furthermore, it is still possible that `result_a` will be present multiple times before `result_b` is next present. Should all the different values of `result_a` be cached, or should all but the latest be discarded? Such questions do not even arise if one knows that `result_a` and `result_b` are always logically simultaneous.

[86] This means "certainly" arrows or acausal arrows; see figure 6.

$\{(1\text{msec}, 0)\} = \{(0, 0)\}$.

A similar calculation can be done to draw a zero-delay "certainly" arrow from `result_a` to `result_b`. A slight hiccup might be forseen in inferring the zero-delay "certainly" arrow of time smear $-\{(0, k) : k \in \mathbb{N}\}$ from `a.out` to `a.in`, but this should be no problem because there must be a first microstep at which `result_a` is present. Furthermore, the addition still works out: $-\{(1\text{msec}, 0)\} + -\{(0, k) : k \in \mathbb{N}\} + \{(0, 0)\} + \{(0, 0)\} + \{(0, 0)\} + \{(1\text{msec}, 0)\} = -\{(1\text{msec}, 0)\} + (-\{(0, k) : k \in \mathbb{N}\} + \{(1\text{msec}, 0)\}) = -\{(1\text{msec}, 0)\} + \{(1\text{msec}, 0)\} = \{(0, 0)\}$.

Note that a similar advantage would be realized by simpler means in the case where both `ComputationA` and `ComputationB` have fixed, finite microstep delays due to several stages of internal pipelining.[87]

11.2 Example: Sequencing of events

Suppose we wish to implement an "anytime" algorithm without compromising determinism:[88]

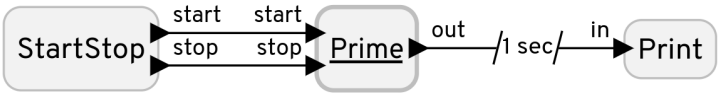

Figure 12: An anytime computation.

After k iterations of the anytime computation, the port `Prime.stop` becomes present, the computation stops, and the best result that has been found so far is used 1 second later in the `Print` reactor. Note that k might be determined at runtime. How do we know that this program will not livelock? That is, how do we know that k is finite?

This cannot be inferred given the information provided so far. For example, if `stop` is scheduled to be present a millisecond after `start` is present, the program will livelock because even though `Prime` is scheduled to stop, it has an infinite[89] number of microsteps to process before it reaches its stop time. However, if we run the algorithm[90] of section 8 on `StartStop` with the following result, then we can conclude that the program will not livelock:

[87] I am not sure how common pipelining will be in idiomatic LF programs, but it is possible to imagine that it will be pervasive, and that microstep delays will appear everywhere in attempts to increase parallelism and to improve performance by giving reactions multiple zero-delay triggers/sources as rarely as possible, so that chain ID is rarely needed.

[88] Our current version of AnytimePrime is intentionally nondeterministic, and the reason for its nondeterminism can be characterized as "fundamental" in the sense that the specification of a piece of machinery is more likely to specify a limited amount of physical time to allocate to a task than a limited amount of logical time. However, a number of logical timesteps can be used as a proxy for physical time, so the use of logical time instead of physical time in an "anytime" computation seems like a reasonable tradeoff: You can have deterministic computations or (mostly) deterministic physical timing, but not both.

[89] I deliberately neglect to model the fact that we are working with the numbers modulo 2^{32} instead of $\mathbb{N}$.

[90] Unfortunately, it might be difficult to actually infer the arrow model shown below from an LF program, depending on how it is written. If stop is produced after several microsteps by some static pipeline that always produces a result, then it is easy to infer that `stop` will certainly be present, but if `stop` is scheduled to be present conditionally after a large number of microsteps, then typically the programmer would have to assert e.g. with an annotation that `stop` will indeed become present.

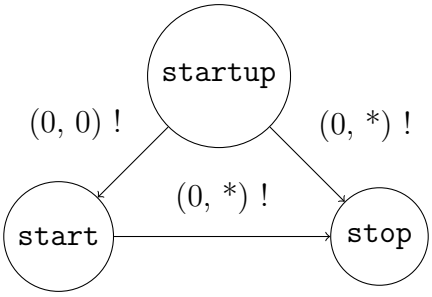

Figure 13: Arrow model of `StartStop`.

To prove that `AnytimePrime` will not livelock, it suffices to show that there exists a valid arrow model of the program with a "certainly" arrow from `AnytimePrime.start` to `AnytimePrime.stop` with a time smear in $2^{\mathbb{N}\times\mathbb{N}}$, all of whose elements have zero as their first element. This is the precondition that the author of the `AnytimePrime` reactor would have to write in order to protect his users.

Indeed, we can infer an acausal arrow from `AnytimePrime.start` to `StartStop.start`, an arrow from `StartStop.start` to `StartStop.stop`, and an arrow from `StartStop .stop` to `AnytimePrime.stop`, with time smears $-\{(0,0)\} = \{(0,0)\}$, $\{(0,n) : n \in \mathbb{N}\}$, and $\{(0,0)\}$ respectively. The sum of these three time smears is just $\{(0,n) : n \in \mathbb{N}\}$, which does indeed have zero as its first entry; furthermore, the composition of the three arrows is indeed an acausal arrow (and not a "nothing" arrow) since they all are either "certainly" arrows or acausal arrows.

It is worth comparing the requirements that can be specified this way with those that can be specified with interface automata. An interface automaton can require that inputs are present in one of a set of input sequences, whereas by requiring input ports to have acausal arrows between them, we require that when an input appears at a given port, the input sequence is required to contain one of a set of (possibly finite) subsequences.

12 Problems resulting from information loss

The following two diagrams show how information is lost about the rates of sporadic events even in the presence of min spacings. The elliptical path, which starts at the bottom left and trails off to infinity at the top of the image, is a logical timeline; the lines secant to the ellipse represent possible sequences of events, or equivalently, possible paths through time.

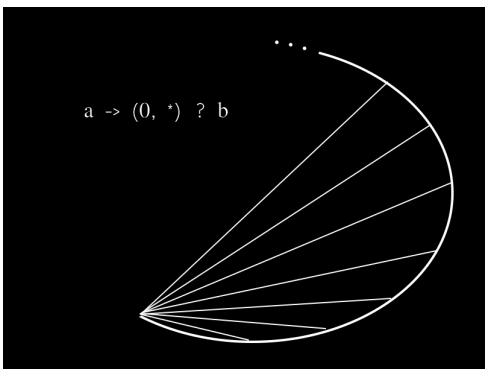

Figure 14: A naive interpretation of what a->$(0,*)?b$ "should" mean.

The above arrow characterization of the program, $a\text{->}(0,*)?b$, might encode the knowledge that a can be followed by b once, any number of microsteps later – this is the idea illustrated by the image, where the starting point of each secant line is an occurrence of a and the ending point is a possible later ocurrence of b. However, given the semantics of the arrow as it is defined in the preceding sections, the arrow might encode the possibility that b could happen repeatedly, at any number of times within the same moment. The first possibility is commonplace and safe; the second possiblity is a potential Zeno condition.

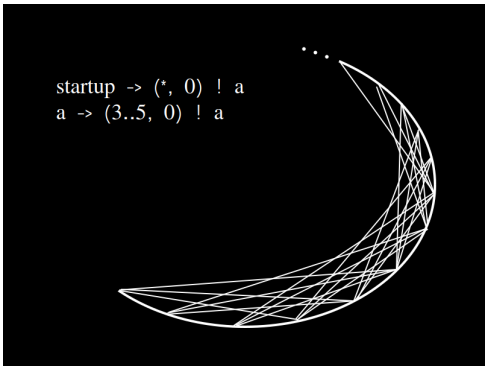

Figure 15: A naive interpretation of what $a\text{->}(3..5,0)!a$ "should" mean.

One might guess that the intent of the arrows in this second image is to encode that, for each occurrence of a, a can occur once more, 3-5 time units later – this is the idea illustrated by the image. However, given the semantics of the arrow as defined in the preceding sections, the first arrow might encode the possibility that a could happen any number of times with any frequency; furthermore, even if the first arrow were associated with a fixed time smear, it would still be possible for a to be present with any frequency at any time, after an initial startup period; this is because the interval $[3,5]$ includes integers that are coprime to each other. The one arrow with the time smear $\{(3,0),(4,0),(5,0)\}$ is equivalent[91] to three arrows with time differences that are coprime in T, the part of $\mathbb{N}\times\mathbb{N}$ isomorphic to $\mathbb{N}$. Such arrows can, through composition, eventually generate arrows in T from a to itself (and therefore from startup to a) corresponding to all natural numbers greater than or equal to seven.[92]

I am not sure how to solve this.

One way would be to allow objects to be events (presence of reactions, actions, and ports) instead of the reactions, actions, and ports themselves. This would make the resulting arrow model look more like the two images presented here, both of which convey the frequency information correctly. However, this would result in infinitely many objects, which might make analysis more difficult.

Another way would be to give the arrows a different meaning so that they somehow indicate a one-to-one relationship (i.e., the presence of an object at one time cannot imply the presence of another object at multiple other times). I have not thought much about this idea because it seems complicated.

[91] See property 2

[92] $7 = 3+4$; $8 = 3+5$; $9 = 4+5$; and numbers greater than 9 differ from 7, 8, or 9 by some positive integer multiple of 3. Bezout's identity tells us that something like this will happen whenever we have time differences in T (or in a right-translation of T by an element of N) or in N (defined in 3) that are coprime to each other in those respective sets.

Alternatively, we may simply accept that time smears with coprime time differences are an antipattern that should be avoided. The ability to schedule events in coprime intervals may, in general, result in chaos for typical programs, whereas the existence of well-defined, non-coprime intervals at which events can be present can enhance predictability. This intuition can be made precise in specific examples.

One metric of the size of a program's complexity is the number of events in one of its hyperperiods. If reaction r is scheduled with period p and with period q, and the GCD of p and q is d hyperperiod will be $\frac{pq}{d}$. If $d = 1$, corresponding to 1 nanosecond, and p and q are on the order of milliseconds, then the hyperperiod will be on the order of 10^6 times greater than either p or q. If p and q were required to be of millisecond granularity (i.e., $d \geq 10^6$), then the hyperperiod might be only a few times greater than p or q.

A second example is the prevention of overutilization. An event producer may guarantee minimum spacings between its events so that downstream event consumers have enough time to finish their work before receiving another event; this works fine. When there are two event producers, however, whose outgoing event streams are merged and both consumed by a single consumer, the property of min spacing is not preserved. Instead, only the rate limit is preserved, which is sufficient for the weak guarantee of bounded lag but insufficient for the stronger guarantee that all events can be serviced by event consumers immediately. By contrast, the discretization approach which discourages the possibility of mutually coprime time differences will in many practical situations preserve the property that minimum spacings are greater than one even when event streams are merged.

A third example which came up in Erling's work is that if two events need to occur at the same logical time on a system with a single processor, one of them will be delayed. To guarantee that this never happens, it suffices to require that the periods of both events have a common divisor[93] d and that the times at which the two events occur are never equal[94] modulo d.

A fourth example is when it is desirable for events at certain objects to be processed in parallel when they are on a single federate with multiple processors.[95] This is much more likely to happen when some arrows from startup to each of two objects involve time differences that are equivalent in λ-reachability from g_0 for some g_0 and some λ that is large relative to the actual periods of the two events.

Such examples have led me to guess that we can reasonably dismiss as pathological the programs that exhibit the information loss described here.

[93]The microstep of the periods should be zero if the two events must both be timed precisely, so the set we are working with is isomorphic to $\mathbb{N}$.

[94]One would likely also wish the times modulo d to differ by at least some minimal time difference, so that the execution of one tag does not delay the execution of the next.

[95]Even when there is a single processor, we can imagine strategies when performance improvements could result from knowing that many events might happen at a single time. Erling has recently found that we incur unnecessary overhead, e.g. in the Big benchmark, when many simultaneous events are given their own place on the event queue. We could instead merge them into a single event with some information attached to it regarding what is actually scheduled.

13 Summary

An arrow model of an LF program can be used to check simple relationships between objects that are relevant to functional correctness. It might be useful for describing timing behavior as well.

A valid arrow model exists for any LF program. Furthermore, it is possible to make such models modular and scalable in the sense that their complexity does not necessarily increase as modules are composed. The cost of such generality and simplicity is the use of extra "maybe" arrows as an overapproximation.