Safe, Modular Packet Pipeline Programming

DEVON LOEHR, Princeton University, US
DAVID WALKER, Princeton University, US

The P4 language and programmable switch hardware, like the Intel Tofino, have made it possible for network engineers to write new programs that customize operation of computer networks, thereby improving performance, fault-tolerance, energy use, and security. Unfortunately, *possible* does not mean *easy*—there are many implicit constraints that programmers must obey if they wish their programs to compile to specialized networking hardware. In particular, all computations on the same switch must access data structures in a consistent order, or it will not be possible to lay that data out along the switch's packet-processing pipeline. In this paper, we define Lucid 2.0, a new language and type system that guarantees programs access data in a consistent order and hence are *pipeline-safe*. Lucid 2.0 builds on top of the original Lucid language, which is also pipeline-safe, but lacks the features needed for modular construction of data structure libraries. Hence, Lucid 2.0 adds (1) polymorphism and ordering constraints for code reuse; (2) abstract, hierarchical pipeline locations and data types to support information hiding; (3) compile-time constructors, vectors and loops to allow for construction of flexible data structures; and (4) type inference to lessen the burden of program annotations. We develop the meta-theory of Lucid 2.0, prove soundness, and show how to encode constraint checking as an SMT problem. We demonstrate the utility of Lucid 2.0 by developing a suite of useful networking libraries and applications that exploit our new language features, including Bloom filters, sketches, cuckoo hash tables, distributed firewalls, DNS reflection defenses, network address translators (NATs) and a probabilistic traffic monitoring service.

CCS Concepts: • **Theory of computation** → **Type structures**; • **Software and its engineering** → *Formal language definitions.*

Additional Key Words and Phrases: Network programming languages, P4, PISA, type and effect systems

ACM Reference Format:
Devon Loehr and David Walker. 2022. Safe, Modular Packet Pipeline Programming. *Proc. ACM Program. Lang.* 6, POPL, Article 38 (January 2022), 28 pages. https://doi.org/10.1145/3498699

1 INTRODUCTION

As industrial networks have grown in size and scale over the last couple of decades, there has been an inexorable push towards making them more programmable. Doing so allows networks to be customized to particular tasks or operating environments, and can deliver better response times, decreased energy usage, superior fault tolerance, or improved security.

P4 (Bosshart et al. [2014]) is one of the outcomes of this push towards programmability. The P4 language allows programmers to not only modify the stateless forwarding behavior of networks (à la NetKAT (Anderson et al. [2014]) or Frenetic (Foster et al. [2011])), but to write stateful networking applications that run inside the packet-processing pipelines of networking hardware like the Intel Tofino (Bosshart et al. [2013]). A plethora of prior work has shown that running applications in these pipelines can yield tremendous performance benefits: in an environment where nanoseconds

Authors' addresses: Devon Loehr, Princeton University, US, dloehr@princeton.edu; David Walker, Princeton University, US, dpw@cs.princeton.edu.

2475-1421/2022/1-ART38
https://doi.org/10.1145/3498699

matter, adaptive, P4-based services such as load balancers (Alizadeh et al. [2014]; Hsu et al. [2020]; Katta et al. [2016]), automatic rerouters (Hsu et al. [2020]), and DDoS defenses (Liu et al. [2021]) can react orders of magnitude faster than systems using network controllers hosted on servers. Indeed, recent work has demonstrated latency reductions of up to 98% in 5G mobile cores (Shah et al. [2020]), and speedups of over 300X in stateful firewalls (Sonchack et al. [2021]), after moving applications into hardware pipelines.

However, while P4 makes it possible to write these applications, it does not make it *easy*: syntactically correct P4 programs regularly fail to compile, because the hardware imposes a collection of implicit constraints on programs. To achieve both programmability and guaranteed high throughput, switches like the Tofino have adopted the *Protocol-Independent Switch Architecture* (PISA), which is structured as a linear pipeline of reconfigurable packet-processing stages. Packets flow forward through the stages, with each stage having its own independent memory for storing persistent information. Since stage X cannot access the memory of stage Y, all computations implemented on a switch must access data structures in the same order. If one computation accesses D_1 and then later D_2, and another accesses D_2 then D_1, there is no way to allocate D_1 and D_2 to stages and compile the computations to hardware.

In this paper, we define Lucid 2.0 (or simply Lucid2), an extension of the original Lucid language [Sonchack et al. 2021] (henceforth Lucid1) for programming packet-processing pipelines. Lucid1 defined a distributed, event-driven programming model for programmable switches, showed how to develop a number of useful network applications, and provided an optimizing compiler targeting a subset of P4 that can be compiled to the Tofino. Lucid1 also defined a type system that ensured data is used in a consistent order. However, the Lucid1 type system was inflexible and did not support modular programming idioms: it was impossible to implement data structure libraries, define abstract types and enforce information hiding, or enable most forms of code reuse. Lucid2 ameliorates these deficiencies by allowing users to implement, use, and reuse rich, high-level libraries for common networking data structures such as (cuckoo) hash tables, sketches, caches, and Bloom filters, while ensuring they and their uses in client code are *pipeline-safe*. In other words, Lucid2 guarantees that all computations touch data in a consistent order, and hence can be laid out along a pipeline.

To achieve these results, Lucid2 introduces a series of new language and type system features that together make it possible for users to write modular programs:

- **Polymorphism** allows safe reuse of functions on data at many pipeline locations, and **ordering constraints** guarantee these functions are safe to call.
- **Hierarchical locations**, which represent abstract pipeline stages, make it possible to define compound data structures inside modules with abstract types, while hiding the structure of the data from client code.
- Despite the fact that PISA architectures do not support dynamically allocated memory, **compile-time constructors, vectors and loops** make it possible to write functions that allocate data structures of variable size and operate over them.
- **Type inference** largely hides static locations and effects from programmers, while a reduction from our algebra of hierarchical locations to the SMT theory of arrays allows us to **automate constraint satisfaction and validity checks**. Only in module interfaces and at declarations of mutually recursive event handlers, where constraints act as loop invariants, do programmers need to explicitly add annotations.

We illustrate the utility of these new features by reimplementing a variety of applications that had previously been implemented in Lucid1. The Lucid1 implementations were each monolithic and non-modular, with no reuse of libraries across different programs. In contrast, in Lucid2 we began by creating a collection of generic, reuseable libraries for common networking data structures including cuckoo hash tables, Bloom filters, count-min sketches, and maps. Many of the libraries include variations with extra features, like the ability to time out and delete stale entries. We used these libraries to construct several useful stand-alone applications, including a distributed firewall, a DNS reflection defense, a NAT, and a probabilistic traffic monitoring service—each of these applications saw significant benefits in terms of modularity and clarity from being able to reuse data structures. Only three Lucid1 benchmarks (chain replication of a single array, the RIP routing protocol, and an automatic rerouting application) were simple enough, or perhaps unusual enough, that they failed to benefit significantly from modularization.

We also formalize Lucid2's semantics and prove sound its type system. In the latter case, the key challenge arises in analyzing the correctness of loops: in order to ensure pipeline safety, the type system must show that all data accesses during the $i + 1^{th}$ iteration of a loop occur later in the pipeline than accesses during the i^{th} iteration of the loop, for all i. To achieve this property, we show that checking the safety of a finite number of loop iterations—three, to be precise—implies the safety of an arbitrary number of loop iterations.

Finally, although Lucid2 is built on top of Lucid1, which compiles to the Intel Tofino, there are other architectures that use reconfigurable pipelines—pipelined parallelism is fundamental for achieving the high throughputs necessary in modern switches. For instance, the Broadcom Trident-4 (Kalkunte [2019]) and the Pensando Capri (Baldi [2020]) are both alternative architectures for packet-processing, and others have been proposed (Jeyakumar et al. [2014]; Sivaraman et al. [2016]). Reconfigurable pipelines have also been used in other domains, such as signal processing (Ebeling et al. [1996]). Lucid2 and its type system lay a new foundation for this important paradigm.

In summary, Lucid2 is the first language to enable safe, *modular* programming for pipelined architectures. In the remainder of the paper, §2 provides more background on PISA architectures and describes Lucid2 and its features by example. §3 formalizes the core features of Lucid2, including its operational semantics and type system. §4 develops the meta-theory of Lucid2 and sketches a proof of soundness. §5 describes our implementation and some of the additional challenges there, including our solution to the constraint solving problem. We also describe the libraries and applications we have built to date. We discuss related work in §6, and conclude in §7.

2 KEY IDEAS

This section presents several of the key ideas underlying the design of Lucid2 and its type system. §2.1 provides background on the mechanics of the PISA architectures Lucid2 is designed to program. §2.1, §2.2 and §2.3 also introduce the basic imperative programming model used by Lucid2. The ideas in these sections are not new; they are borrowed from Lucid1 (Sonchack et al. [2021]). §2.4 through §2.7 describe new ideas introduced in this paper: polymorphism and constraints; records and hierarchical locations; compile-time constructors, vectors, and loops; and type-and-effect inference.

2.1 Packet Processing Pipelines

Programmability, high and guaranteed line rate, and feasible hardware implementation are the primary design goals of modern switch chips like the Intel Tofino. We can characterize these chips, generally, as instances of the *Protocol-Independent Switch Architecture* (PISA) (Bosshart et al. [2013]). In such an architecture, when packets arrive at a switch, they are parsed, key *header fields* (source IP, destination IP, *etc.*) are extracted, and the data in these fields is passed to the switch's *packet-processing pipeline.*

The pipeline itself consists of several *stages.* At a high level of abstraction, each stage has two main components: (1) some of stateful memory, which persists across packets, and (2) a match-action

```
global int g1 = 1; // Global mutable integers persist
global int g2 = 7; // across invocations of handlers

handle simple() {
 int x = !g1;       // Read g1's current value; store in local x
 int y = x + x;
 g2 := y;           // Read y; store in g2
}
```

Fig. 1. A simple Lucid program. The body of simple is executed whenever the switch receives a "simple" event, which may be tied to reception of a packet.

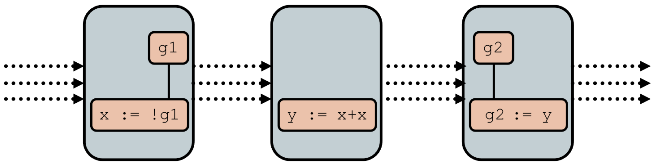

Fig. 2. A 3-stage pipeline that executes the code in Figure 1. Packets enter one-by-one from the left and travel left-to-right through the stages. Stage 1 contains the persistent state g1 as well as code, executed by an ALU, that reads that state. Stage 2 uses only temporaries x and y, which flow from one stage to the next, but whose values do not persist from one packet to the next. Stage 3 contains state g2 and an action to store into g2.

```
global int g1 = 1;
global int g2 = 2;

handle simple() {
 int x = !g1;
 int y = x + x;
 g2 := y;
}

// badly() accesses g1 and g2 in a different order from simple()
handle badly() {
 int x = !g2;
 int y = x + x;
 g1 := y;
}
```

Fig. 3. An uncompilable program. Persistent mutable references g1 and g2 cannot be allocated to pipeline stages because the two handlers access them in opposite orders, generating unsatisfiable ordering constraints.

table, containing a number of rules that each match some set of packets, and, when they match, execute some *action*. Actions can involve reading or writing local variables and/or stateful data, and performing simple arithmetic or other operations such as computing a hash. However, while header fields of packets and local variables are propagated from stage to stage, stateful memory can only be accessed in the stage that contains it. Even then, stateful data can be "accessed" only once per packet[1], because packets are forwarded to the next stage immediately upon completion of the prior stage's actions. Although several aspects of the pipeline (such as the the amount of memory in each stage or the possible actions) vary by architecture, they all share this basic form.

[1]An "access" can involve a read, a simple arithmetic computation, such as an addition, and a write back to stateful memory.

As a point of reference, the Tofino has 12 stages, each containing approximately 1MB of stateful memory which can be partitioned into at most 4 separate *register arrays*. Each packet has approximately 512 bytes of dedicated header space in which local variables and control information are stored. These numbers are likely to grow as new hardware (such as the Tofino 2 [Intel 2020]) is released, but the PISA architecture itself is independent of them.

Once a packet has passed through the pipeline, it is forwarded through one of the switch's ports. Most of the time, such packets will travel on to other switches or host machines, but sometimes a switch will use *recirculation* to send a packet back into the pipeline from which it just came. Recirculation allows the switch to continue processing the packet, but it is an expensive operation—it cuts directly into the number of packets per second a switch can process and increases the latency of packets travelling from point A to point B. Hence, it must be used sparingly, typically only on a very few *network control packets*, which are responsible for configuration of network behavior.

Lucid2 is designed to program PISA pipelines, providing the veneer of a simple imperative language on top of the hardware. Figure 1 presents a small program that illustrates a few basic features of the language using a simplified syntax. The program declares two global variables, `g1` and `g2` (globals are mutable and their state is persistent across packets), and a user-defined event handler, triggered when the switch receives the `simple` event. Events are triggered when particular packets arrive at the switch. In this case, the `simple` handler reads from `g1` and writes to `g2`.

Compiling a program to a PISA pipeline involves deciding in which stage each global variable and computation should reside, while abiding by hardware limitations on the amount of state and number of actions that fit in a stage. Figure 2 shows one way to compile this program to a 3-stage pipeline, which we will assume can accommodate a single action per stage. Here, the compiler places `g1` in stage 1 and `g2` in stage 3. Stage 2 is used for the addition operation. The program dependencies determine the pipeline layout rather directly here: `g2 := y` must take place after `y = x+x`, which must occur after `x = !g1`, and the globals must be allocated in the same stage as the actions that refer to them.

Compiling high-level computations to hardware is not always as easy as this example suggests. Figure 3 presents a second program that accesses `g1` before `g2` in the first handler, and `g2` before `g1` in the second handler. To lay out both computations on a single pass through a PISA pipeline, we would have to place `g1` before `g2` and `g2` before `g1`, which is impossible. One solution would be to eschew a single pass and use recirculation to implement one of the two functions. However, doing so adds an enormous (often impractical) cost to packet processing. Hence, rather than introduce recirculation automatically, our goal is to detect these sorts of problems and provide programmers with useful source-level feedback for correcting the error.

2.2 Ordering Constraints

Our type system is designed to ensure the following properties.

(1) No stateful data is accessed twice in the same pipeline pass (since the packet moves to the next stage immediately after accessing the data)
(2) There is some order on global data such that for every pair of data accesses, the data accessed first appears earlier in the order

These constraints are reminiscent of those imposed by certain substructural type systems (Girard [1987]; Polakow and Pfenning [1999a]; Polokow and Pfenning [1999]; Walker [2005]). For instance, Polakow and Pfenning's ordered type systems (Polakow and Pfenning [1999a]; Polokow and Pfenning [1999]) provide programmers control over the order in which their data must be accessed. Such a system, appropriately modified for our domain, might imply many of the constraints we need, but appears more restrictive than we would like. For example, our system contains loops,

```
1   const int len = ...;
2   global array<bool> a0 = Array.create(len);
3   global array<bool> a1 = Array.create(len);
4   const int s0 = ...; // seed for first hash table
5   const int s1 = ...; // seed for second hash table
6
7   // add item to bloom filter
8   fun void add(int item) {
9       a0.(hash(s0, item)) := true;
10      a1.(hash(s1, item)) := true;
11  }
12
13  // return true if item in bloom filter
14  fun bool query(int item) {
15      bool b1 = a0.(hash(s0, item));
16      bool b2 = a1.(hash(s1, item));
17      return (b1 and b2);
18  }
```

Fig. 4. A basic Bloom filter with m = 2. Functions `add` and `query` may be called from many different handlers.

which require careful reasoning about inequalities that does not appear possible in vanilla ordered type systems. Moreover, switch hardware permits ordered data to be allocated during compile time only, which is simpler than the dynamic allocation permitted in standard ordered type systems.

2.3 A Basic Bloom Filter

For the remainder of this section, we will explain Lucid2 through the working example of a Bloom filter. A Bloom filter is a probabilistic data structure for representing a set of elements, consisting of k boolean arrays of length m, each associated with a hash function. Items are added to the Bloom filter by processing them with each of the k hash functions to produce k array indices, and then setting each index to `true` in the associated array. To check if an item appears in the data structure, one hashes that item k ways and returns `true` if and only if all the associated indices are already set to `true`. Bloom filters are useful for applications which are willing to trade occasional imprecision for reduced memory usage, and are often found in network monitoring applications.

Figure 4 shows a simple Lucid2 program that implements a Bloom filter. As Lucid2 type checks the program, it keeps track of both *raw types* and *locations* of global mutable data. For instance, in this case, `a0` is an array of booleans stored at location `0` (because it is the first declaration). We write `a0`'s *full type* as `array<bool>@0`. Since `a1` is declared immediately after `a0`, `a1`'s full type is `array<bool>@1`. Thanks to Lucid2's type inference, programmers typically need only write raw types (as shown in Figure 4) and may drop explicit location annotations.

As Lucid2 checks that a series of statements or expressions is well-formed, it keeps track of where the computation is—called the *current location*—in a virtual pipeline. Whenever a global variable is accessed, it first checks if the current location precedes the location of that global variable. If so, it updates the current location, moving it one location past whichever global variable was accessed. If not, the program fails to typecheck.

Figure 4 typechecks, but suppose a programmer accidentally permuted the two array accesses on lines 9 and 10 of the add method, resulting in the following two lines.

```
9   a1.(hash(s1, item)) := true;
10  a0.(hash(s0, item)) := true;
```

In this case, Lucid2 would generate an ordering violation at line 10, since line 10 accesses a0, which is at location 0, when that location has already been bypassed in the pipeline. The programmer would then be able to look backwards from line 10, notice that they had already accessed a1 on line 9, and determine a solution. In this case, simply swapping the offending lines would suffice.

Aside: An alternate design choice. Lucid2 demands that all program components access stateful data in the order it is declared. If all components consistently used state in some other order, our system would flag an error even though the program could be compiled. An alternate design could allow programmers to use data in any order, provided they do so consistently across their whole program, or provided the system can permute accesses without changing program semantics to arrive at a consistent order (as was the case in the prior paragraph's example).

We conjecture this other design is easily achievable and, from a technical perspective, varies little from our chosen design (we would simply find a satisfying assignment to ordering constraints rather than check that such constraints are consistent with an *a priori* ordering). However, we chose to require that programmers follow declaration order for two reasons: (1) declaration order provides useful, built-in documentation and (2) it is easier to provide targeted error messages when things go wrong. Although programmers cannot entirely avoid thinking about state ordering, Lucid2 boils the requirements down to a simple, easy-to-state guideline. When programmers violate this guideline, Lucid2 can issue a simple message of the form "Line X conflicts with the global order," which allows programmers to navigate right to the source of their problem and fix it quickly.

2.4 Polymorphism and Constraints

Unfortunately, the Bloom filter code in Figure 4 is not reusable: The add and query routines operate over particular arrays, whose locations in the pipeline are fixed. Consequently, programmers must write new Bloom filter code with separate add and query methods every time the underlying arrays or their locations are changed.

To better accommodate code reuse, a first effort might simply parameterize the add and query methods by the arrays to be used, as is done in the following code.

```
fun void add(array<bool> a0, array<bool> a1, int s0, int s1, int item)
{
   a0.(hash(s0, item)) := true;
   a1.(hash(s1, item)) := true;
}
```

However, one cannot guarantee the code above is safe. Indeed, the function is only safe when the location of a0 precedes the location of a1.

To facilitate proofs of safety, we extend our function definitions to admit location polymorphism and ordering constraints over polymorphic locations. Below, we rewrite our function with appropriate constraints, using the special keyword start to denote the location at which the function begins execution. Within the constraint clause below, we write a0 < a1 to mean that $\ell_{a0} < \ell_{a1}$, where ℓ_{a0} and ℓ_{a1} are the locations associated with a0 and a1.

```
fun void [start <= a0 /\ a0 < a1]
     add(array<bool> a0, array<bool> a1, int s0, int s1, int item)
{
   a0.(hash(s0, item)) := true;
   a1.(hash(s1, item)) := true;
}
```

Since type checking now involves reasoning about symbolic integer locations and inequality constraints, we deploy an off-the-shelf SMT solver to check satisfiability.

2.5 Records and Modules

Now our intrepid programmer has the ability to reuse their Bloom filter code when the underlying state is located at different stages in a pipeline. Still, the representation of the Bloom filter is apparent and explicitly manipulated by the client code—there is no way to reimplement the filter (*e.g.* to improve its accuracy by using three or more arrays) without modifying the client as well. Figure 5 presents a revised design that uses compound record types and data abstraction to hide the structure of the Bloom filter implementation from the client. The record type `filter` represents a Bloom filter, and the constructor `createFilter` is a special compile-time function that allocates memory to create a `filter` value.

While extending most languages with compound and abstract types is relatively straightforward, in our case, these extensions have unusual consequences for the structure of the effect system. During compilation, records must be unboxed (there is no hardware support for them), and their array fields must be placed in the pipeline, as in Figure 6. Just like top-level globals, we require that global fields of each record are stored in the order those fields are declared in the record type.

A first, naïve choice for choosing locations for the data might be to house `a0` at location ℓ (for some ℓ) and `a1` at location $\ell + 1$.[2] However, if we do so, then client code that uses a Bloom filter operation will move forward k locations, where k is the number of arrays in the filter. In other words, information about the filter's underlying implementation will be leaked to the client.

2.5.1 Hierarchical Locations. Our solution is to introduce hierarchical locations, with the structure of the hierarchy following the structure of the type declarations introduced by the programmer. In our hierarchy, if a record is allocated at location ℓ then its fields will be nested at locations "within" ℓ, written as $\ell.0$ for the first field, $\ell.1$ for the second, $\ell.2$ for the third, and so on. Intuitively, the record's location is "virtual", and the nested locations which correspond to array types are the "real" locations that will be allocated along the hardware's pipeline during compilation.

For example, when a programmer declares a record type holding a pair of arrays, like the one in Figure 5, each record `r` will be placed at some virtual location ℓ, and the arrays `r.a0` and `r.a1` will be nested underneath it at locations $\ell.0$ and $\ell.1$, respectively. Some other data structure located immediately after the record may be positioned at location $\ell + 1$. Notice how the location $\ell + 1$ reveals nothing about the structure of ℓ. The location ℓ may contain may nested sub-locations and they in turn may contain more nested sub-locations, or none at all. The client cannot tell.

More generally, our "virtual pipeline" is now an ordered tree of locations—Figure 7 presents a picture of such a memory. The root is a virtual location; each top-level global program variable is a child of the root; and compound types such as records induce additional nested locations. We refer to specific locations using paths from the root to other nodes of the tree. For instance, the path $n_0.n_1.n_2$ is read from left-to-right, and chooses the n_{ith} child at each step from root to leaf. In the example of Figure 7, `f1` would have location 0, `f1.a0` would have location 0.0, and `f1.a1` would have location 0.1. Similarly, `f2`, `f2.a0`, and `f2.a1` have locations 1, 1.0, and 1.1, respectively.

To prevent ordering errors, the type system must reason about the order that these locations will be laid out in a physical pipeline. When comparing locations, the ordering used corresponds the pre-order traversal of the (non-root) nodes of the memory tree. For instance, here is the ordering of several locations: $0 < 1 < 1.0 < 1.4.7 < 1.5 < 1.5.3 < 2$.

The type system must also reason about, relate, and manipulate abstract, universally quantified locations. It does so via a simple algebra of locations that includes a successor function. Hence, in general, we may write $S(\ell)$ (or equivalently, $\ell + 1$) for the successor of the (possibly abstract)

[2]Immutable scalars such as the integers `s0` and `s1` need not be housed in pipeline stages so we need not give them locations.

```
module BloomFilter = {
    // An abstract record type, with definition hidden from module clients
    abstract type filter = {
        a0 : array<bool>;  // Where should this be stored?
        a1 : array<bool>;  // Depends on the location of the filter object
        int s0;            // These never change, so they don't
        int s1;            // need to be stored in the pipeline
    }

    // A compile-time function for creating global values.
    constructor createFilter(int m, int seed1, int seed2) = {
        a0 = Array.create(m);
        a1 = Array.create(m);
        s1 = seed1;
        s2 = seed2;
    }

    fun void [start <= bf] add(filter bf, int item) {
        bf.a0.(hash(bf.s0, item)) := true;
        bf.a1.(hash(bf.s1, item)) := true;
    }

    fun bool [start <= bf] query(filter bf, int item){
        bool b1 = bf.a0.(hash(bf.s0, item));
        bool b2 = bf.a1.(hash(bf.s1, item));
        return (b1 and b2);
    }
}

// Using the constructor
global filter f1 = BloomFilter.createFilter(...);
global filter f2 = BloomFilter.createFilter(...);
```

Fig. 5. An abstract, compound type for Bloom filters.

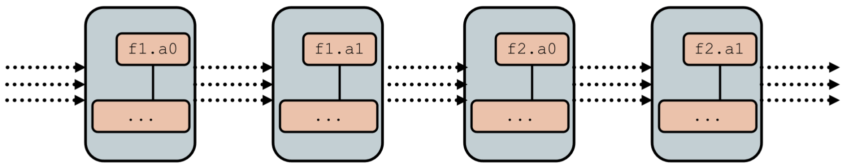

Fig. 6. Data layout for the two globals in Figure 5. Only the array values appear in the pipeline—the seeds are immutable and do not need to be store in mutable stage memory; the records themselves are unboxed and compiled away.

location ℓ. Checking satisfiability of constraints involving polymorphic variables is trickier in this setting, but is still decidable with an SMT encoding we have developed (see §5.3).

In our model, the leaf nodes of the tree are precisely the array-type variables—that is, the mutable globals that must be stored in the pipeline.[3] We can linearize our memory model and assign mutable data to physical pipeline stages in a PISA architecture simply by dropping the non-leaf nodes from the tree and assigning the leaves to stages in order.

[3]Arrays do not themselves contain other arrays or mutable references. Memory is flat. There are no pointers.

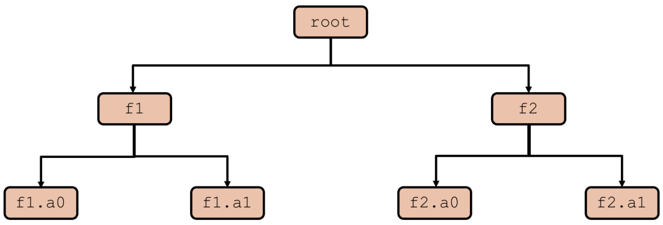

Fig. 7. An abstract representation of the memory in Figure 5. The location order is the preorder traversal of the tree. The ordering of the pipeline in Figure 6 is given by the left-to-right sequence of the leaves.

```
module BloomFilter = {
    // A filter with k arrays
    abstract type filter<k> = {
        arrs : array<bool>[k]; // Vector of k arrays of booleans
        seeds : int[k];       // Vector of k ints
    }

    // create Bloom Filter with ss -- a vector of k seeds
    constructor createFilter(int m, int[k] ss) = {
        arrs = [Array.create(m) for i < k]; // Vector comprehension
        seeds = ss;
    }

    fun void [start <= bf] add(filter<k> bf, int item) {
        for i < k { // Declares a new index i ranging from 0 to k-1 inclusive
            bf.arrs[i].(hash(bf.seeds[i], item)) := true;
        }
    }

    fun bool [start <= bf] query(filter<k> bf, int item) {
        for i < k {
            bool b = bf.arrs[i].(hash(bf.seeds[i], item));
            if (not b) { return false; }
        }
        return true;
    }
}
```

Fig. 8. A general module for Bloom filters

2.5.2 Typechecking Figure 5. With this new location structure, we have the tools we need to typecheck our modular Bloom filter. If we have a filter at location ℓ, we assign `a0` and `a1` the locations $\ell.0$ and $\ell.1$, respectively. While the functions add and `query` manipulate the sublocations $\ell.0$ and $\ell.1$, we will avoid revealing those locations to a client by "rounding our location up" at the end of the function to the successor of the parent location ℓ (namely, $\ell + 1$) rather than, say, to the successor of $\ell.1$, (namely, $\ell.(1 + 1) = \ell.2$). From the perspective of a user outside the module, the add function now simply consumes the `filter` argument, moving from location ℓ to $\ell + 1$—all information about the implementation of the `filter` type is hidden.

2.6 Vectors

Our Bloom filter implementation has come a long way, but there's one annoyance left—namely, all our work has focused on Bloom filters implemented with two arrays (*i.e.*, with $m = 2$). If an application requires a different memory-accuracy trade-off, it may want to use a Bloom filter with $m = 3$ or $m = 4$. Unfortunately, to implement such a filter at this point, one would have to write an entirely new module with a new type and functions. To address this limitation, we allow programmers to write variably-sized vectors of values, providing them the flexibility needed to write a general Bloom filter module as in Figure 8.

Since data-plane programs must ultimately run on the linear switch hardware, which does not permit looping, we allow only bounded loops of the form "`for i < k` ..." that can be unrolled during compilation. In order to avoid out-of-bounds errors, we include the length of a vector in its type, and allow indexing operations only if the index can be proved to be in bounds. Constraints generated from an index declaration $i < k$ suffice for such proofs in our application domain.

Fortunately, adapting the hierarchical locations of the previous section to accommodate vectors is simple. We can view vectors as nodes in the heap with a variable number of identical children, and when we specify a child we may do so either with a concrete integer as before, or with a loop variable (for example, $0.1.i$ where i is a loop variable). When comparing locations ℓ_1 and ℓ_2 that involve variables, we say that $\ell_1 < \ell_2$ only if that relationship holds for every instantiation of the variables in ℓ_1 and ℓ_2. So, for example, 0.i < 1, but 0.i and 0.1 are incomparable.

Loop constraints. Since all our loops are bounded, and include bounds checking, termination is guaranteed and indexing errors do not occur. However, we do need to ensure that loop bodies will not result in ordering errors when run multiple times.

To check a loop of the form `for i < k { e }` starting at location ℓ_{init}, we must ask:

(1) Can we safely execute the loop body with $i = 0$ and starting at ℓ_{init}?
(2) For all $j > 0$, can we safely execute the loop body with $i = j$, starting at the ending location of the prior iteration?

To demonstrate the necessity of (1), assume we have two globals of type `array<bool>[k]` named `arr1` and `arr2`, with locations 1 and 2, respectively, and assume the function `access` consumes its argument. Consider the following loop:

```
access(arr2[0]);
for i < k { access(arr1[i]); }
```

At the start of the loop, ℓ_{init} will be `2.1` (one step past 2.0), and on the first iteration we will access `arr1[0]`, which has location 1.0. Since 1.0 < 2.1, we run into an ordering error immediately. We can always detect violations of property (1) this way, by typechecking the loop body with $i = 0$.

Detecting violations of property (2) is trickier. If the loop bound k is an unbounded size (e.g. if the loop is inside a size-polymorphic function), then naïvely we would need to typecheck the loop body for arbitrarily many iterations, which would require a universally-quantified SMT constraint. Unfortunately, typechecking recursive event handlers requires proving an implication of constraints, and it is unclear whether such an implication will wind up being decidable when the constraints are universally quantified.

Fortunately, there is a better way, which becomes apparent after looking at several "bad" loops. Consider the following programs (in which the types of `arr1` and `arr2` vary as necessary):

```
for i < k { // Loop (a)
    access(arr1[0]);
}

```

```
for i < k { // Loop (b)
    access(arr1[i]);
    access(arr2[i]);
}
```

```
for i < k { // Loop (c)
    for j < k' {
        access(arr1[j][i]);
} }
```

At a glance, they all might seem fine. Loop (a) will begin at location 0, then access location 1 on the first loop. However, on the second loop, it will try to access location 1 again, causing an error. Loop (b), on the other hand, will first access locations 1.0 and 2.0, both of which are in order. However, on the second iteration, it will try to "go back" and access location 1.1, which is less than 2.0. Finally, loop (c) will execute the outer loop once, ending at location 1.k'.1, but on the second iteration it will try to access location 1.0.1, which is less than 1.k'.1 (if k' > 0).

The common thread in all these examples is that despite the loops having several different forms, each of the errors occurred very quickly (within a few iterations of the outermost loop). This is not a coincidence; we have proved that, given certain minor restrictions, *every* "bad" loop will fail in at most three iterations. In other words, if the loop doesn't violate ordering constraints in the first three iterations, it will not do so in any future iteration.

This insight allows us to reduce property (2) from a universal statement to a finite one. Rather than having to reason about every iteration of the loop simultaneously, it suffices to only check the first three. This is a significant victory, and our type system leverages it to turn a potentially undecidable problem into an obviously-decidable one.

2.7 Location Inference

We have now extended our language and type system to handle a fully general Bloom filter module, which is parametric in both m and k. However, this did not come entirely without cost – it is only through location inference that we have avoided leaving cumbersome location annotations throughout the program. Inference is crucial for real programs, since it allows the programmer to think at a high level – rather than reasoning about the low-level details of the effect system, they can maintain a high-level abstraction that "global variables must be used in declaration order".

To support inference, the location grammar we use is carefully designed to have a minimal set of simple constructors: zero (0) and successor (S(ℓ)) constructors to represent integers, and constant/variable projection operators for record and vector entres ($\ell.0$ and $\ell.i$). This choice means that standard unification algorithms (Milner [1978]) can be directly applied to infer both types and locations. Moreover, we can infer constraints for each expression and function, and for the program as a whole, by collecting them as we walk through the program.

In this way, we have almost entirely eliminated locations from the surface syntax of Lucid2. The exceptions are in module interfaces, where we do not have function bodies available to run inference, and in mutually recursive event handlers (see §5.2). Through location inference, Lucid2 programmers are provided with the easy, high level abstraction of "use global variables in the order they are declared", and are not forced to learn a new system before they can continue writing code.

3 LANGUAGE AND TYPE SYSTEM

In this section, we present the formal definition of Lucid2, an extension of an idealized subset of Lucid1 designed to illustrate and prove correct the central elements of our type system.

Lucid2's type system (see Figure 9 for the syntax) contains a collection of compile-time integers, which we refer to as *sizes*. These sizes are used for describing vector lengths, and may appear in locations. They include constants n (a natural number) as well as two different sorts of identifiers, b and κ. We refer to b as a *bounded size*—our type system ensures that such identifiers will always appear with a constraint $b < k$. Such constraints make vector bounds checking straightforward. We refer to identifiers κ as *unbounded sizes*.

Lucid2's type system also includes *locations*, which describe where in a pipeline a piece of persistent memory is stored. The metavariable z ranges over concrete locations whereas ℓ ranges over symbolic locations. The first location in a pipeline is 0. The location $S(z)$ follows the location z. Locations may also be hierarchical. Hence, if z is a location then $z.0$ is the first location within z and $S(z.0)$ is the next location within z. Symbolic locations can be location variables α or hierarchical locations such as $\ell.b$ where b is an index into ℓ.

$\langle \iota$ *(indices)* $\rangle ::= n \mid b$

$\langle k$ *(sizes)* $\rangle ::= \iota \mid \kappa$

$\langle z$ *(concrete locations)* $\rangle ::= 0 \mid \mathrm{S}(z) \mid z.0$

$\langle \ell$ *(locations)* $\rangle ::= 0 \mid \alpha \mid \mathrm{S}(\ell) \mid \ell.0 \mid \ell.b$

$\langle C$ *(constraints)* $\rangle ::= \texttt{true} \mid \ell \le \ell \mid C \wedge C$

$\langle T$ *(base types)* $\rangle ::= \texttt{Bool} \mid \texttt{Unit}$

$\langle t$ *(raw types)* $\rangle ::= T \mid \texttt{addr}(T) \mid (t, t) \mid \texttt{vector}(t, k) \mid \forall \overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau, \ell) \rightarrow (\tau, \ell)$

$\langle \tau$ *(types)* $\rangle ::= \mathrm{t}\langle \ell \rangle$

$\langle v$ *(values)* $\rangle ::= () \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}]\ (x : \tau, \ell) \rightarrow e \mid \texttt{addr}(z) \mid (v, v) \mid \texttt{vector}(v, \ldots, v)$

$\langle e$ *(expressions)* $\rangle ::= v \mid x \mid (e, e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{vector}(e, \ldots, e) \mid e[\iota] \mid [e\ \texttt{for}\ b < k] \mid !e \mid e := e \mid$
$\quad \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{for}\ b < k\ \texttt{do}\ e \mid e[\overline{k}, \overline{\ell}]\ e$

Fig. 9. Formal Lucid2 Syntax

Constraints C are conjunctions of inequalities $\ell_1 \le \ell_2$, which describe the order that locations must appear in memory. There will be more on constraints, locations and operations over them in the following subsection.

Lucid2 contains `Bool` and `Unit` base types as well as *raw types* that include mutable references (`addr(T)`), vectors with elements of type t and length k ($\texttt{vector}(t, k)$), and pairs (t_1, t_2). There are no references to references (the hardware only admits "flat" data structures); this is why we distinguish "raw types" and "base types." Vectors will be unrolled and their associated contents allocated to stages at compile time; their length k is a compile-time computed value. Types proper (τ) are pairs of a raw type and the virtual pipeline stage that stores the value of that raw type, written $t\langle \ell \rangle$. For simplicity and uniformity in the system, base types like `Bool` and `Unit` are associated with a location even though it is not necessary to do so (the stage of a base type winds up playing no role in the system)—only persistent mutable data need be allocated to stage memory.

In general, functions have a type of the form $\forall \overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau_1, \ell_1) \rightarrow (\tau_2, \ell_2)$. These functions are non-recursive, call-by-value functions and will be fully inlined at compile time (the hardware does not have mechanisms for implementing a general purpose function call). They are polymorphic in the sizes (κ) that parameterize vectors, and in locations (α). Function preconditions C are a collection of inequality constraints that must be satisfied prior to calling the function. Functions take an argument with type τ_1 and start at location ℓ_1 in the pipeline, returning a result with type τ_2 and completing at location ℓ_2 in the pipeline. Our implementation contains type-polymorphic functions as well; they are not hard to formalize, but for simplicity we elide them here.

There are values (v) for each type. Notice that function values do not specify required function constraints C—they will be inferred during typechecking. Expressions contain many standard forms. We often use $e1; e2$ as an abbreviation for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ when x does not appear free in e_2. Components of a pair are projected using the `fst` and `snd` operators. Vector projection is written $e[\iota]$. The expression $!e$ reads from the address e and $e_1 := e_2$ writes the value of e_2 to the address e_1. A vector comprehension $[e\ \texttt{for}\ b < k]$ generates a vector of length k with i^{th} component $e[i/b]$. The construction $\texttt{for}\ b < k\ \texttt{do}\ e$ iterates k times over the body, replacing b with i in the i^{th} iteration. Finally $e_1[\overline{k}, \overline{\ell}]e_2$ calls function e_1 with size vector $\overline{k}$, location vector $\overline{\ell}$ and value e_2 as arguments.

We define capture-avoiding substitution in the usual way, and, for instance, use the notation $e[\ell/\alpha]$ for the expression e with all free occurrences of α replaced with ℓ. We substitute vectors of terms ($\overline{\ell}$) for vectors of variables ($\overline{\alpha}$) using the notation $e[\overline{\ell}/\overline{\alpha}]$. Analogous notation is used to denote other sorts of substitutions. We also treat expressions as equivalent if they differ only in the names of bound variables, which we refer to as "alpha-renaming".

3.1 Locations

Location Representations. Locations (ℓ) denote (hierarchical) pipeline stages. We have defined the syntax of location expressions (see Figure 9) via an algebra that involves a successor function $S(\ell)$, which denotes the location after ℓ. However, an expression like $S(S(S(0.0).k))$ is challenging to understand, and sometimes inconvenient technically (though other times it is quite convenient, especially for unification-based type inference, which is why we chose it). There is an isomorphic notation as a (non-empty) list of symbolic natural numbers. Such lists have the following form:

$$\langle L\ (\textit{list location})\rangle ::= \iota + n \mid \alpha + n \mid L.(\iota + n)$$

The following function f converts the standard representation of locations ℓ into a list-based representation L.

$$f(0) = 0 \qquad f(\alpha) = \alpha \qquad f(\ell.\iota) = f(\ell).\iota \qquad f(S(\ell)) = \begin{cases} L.(\iota + n + 1) & \text{if } f(\ell) = L.(\iota + n) \\ f(\ell) + 1 & \text{otherwise} \end{cases}$$

For example, if we apply f to $S(S(S(0.0).i))$ we get the list $0.1.(i+2)$. We use standard list syntax to refer to elements; in our previous example, the head would be 0 and the tail would be $1.(i+2)$. The function f is bijective, so either location syntax contains the same information. In a slight abuse of notation, from this point forward, we will implicitly convert locations back and forth between representations, using whichever is most convenient at the time. We will use the metavariable ℓ to range over effects regardless of the representation.

Location Ordering. When location ℓ_1 occurs earlier in a pipeline than ℓ_2, we write $\ell_1 < \ell_2$. In general, $\ell_1 < \ell_2$ is defined (using the list-based representation of locations) as follows: $\ell_1 < \ell_2$ iff:

(1) ℓ_1 is an empty list and ℓ_2 is a non-empty list[4], or
(2) $\texttt{hd}\ \ell_1 < \texttt{hd}\ \ell_2$, or
(3) $\texttt{hd}\ \ell_1 = \texttt{hd}\ \ell_2$ and $\texttt{tl}\ \ell_1 < \texttt{tl}\ \ell_2$

If either list contains variables (αs, κs, or bs), we say $\ell_1 < \ell_2$ if and only if that relationship holds for all possible instantiations of the variables. That is, we would have $0.0 < 0.(i+1)$, but 0.1 and $0.i$ would be incomparable.

Location Rounding. When processing symbolic locations, we sometimes wish to jump forward to a location guaranteed to come after the symbolic location. For example, given the location $0.0.b$, we may want to jump to 0.1, which is "ahead" of (i.e. greater than) $0.0.b$, for all b. We call this operation *rounding*, and write it $\texttt{round}(\ell, b)$.

We define `round` in terms of another function `drop`, which simply drops all entries after the first instance of b it encounters. Below, and elsewhere, we use the notation $b \notin \ell$ to indicate that ℓ does not contain any instances of b.

$$\texttt{round}(\ell, b) = \begin{cases} \ell & b \notin \ell \\ S(\texttt{drop}(\ell, b)) & \text{otherwise} \end{cases}$$

[4]Although the output of f will never be empty, we may generate an empty list while checking inequality by use of the `tl` operator.

where $\texttt{drop}(\ell, b) = \ell$ if $b \notin \ell$, and otherwise

- $\texttt{drop}(S(\ell), b) = \texttt{drop}(\ell, b)$
- $\texttt{drop}(\ell.0, b) = \texttt{drop}(\ell, b)$
- $\texttt{drop}(\ell.b, b) = \texttt{drop}(\ell, b)$
- $\texttt{drop}(\ell.b', b) = \texttt{drop}(\ell, b)$

Location Well-formedness. The predicate $\texttt{nri}(\ell, b)$ is true when ℓ contains no more than one instance of b. The predicate $\texttt{nri}(\ell)$ is true when ℓ contains no more than one instance of any single b. Finally, $\texttt{nri}(C)$ is true when all locations ℓ appearing in C satisfy $\texttt{nri}(\ell)$.

Constraints. We write $C \Rightarrow C'$ to mean that C implies C', and we write $\vDash C$ when C is *valid—i.e.*, for all well-typed substitutions of values for variables, C is satisfied.

3.2 Pipeline Semantics

Our operational model captures execution of expressions on an abstract pipelined processor. In this model, computations must be organized so that they access memory locations in order, possibly skipping over some of the locations they do not need to access. Immediately after a computation accesses a location, the state of the machine is advanced—each location is accessed at most once. In a real PISA architecture, such as Intel's Tofino chip, a single atomic action may involve several operations, such as a read, a conditional test and a write to the same state that was read from, but successive atomic actions may not touch the same state. Augmenting our machine model with additional primitives to model such compound operations is straightforward. The abstraction we present here, with its simplified atomic actions, captures the essence of such computations.

More formally, the states of our abstract machine are triples (M, z, e), where M is a pipelined memory, z is our current location in the memory, and e is the expression to execute. A pipelined memory is a partial mapping from concrete locations to values.

Figure 10 presents selected rules from the small-step operational semantics of these machines as a relation with the form $(M, z, e) \longrightarrow (M', z', e')$. The complete semantics appears in appendix A of the auxiliary archive.

The most interesting rules are Deref-2 and Update-3. Given that the current location is z and the computation requests a read from address z_e, Deref-2 states that the machine skips forward to z_e (which must be higher in the ordering than z), reads the value in memory at that location, and then advances the current location to $S(z_e)$. Update-3 is similar— the machine skips forward from z to z_e, writes to z_e and then moves forward to the successor location $S(z_e)$.

There are a number of ways such stateful computations can "go wrong." The location z_e might not exist. If it does, it might not be higher in the ordering than the current location z (*i.e.*, we might have already passed it in the pipeline). Our language type system will have to present such scenarios from arising.

Readers will also want to examine the operational rules for vectors and loops. In particular, at run time, a loop bounded by n may be unrolled to n copies of its body. A key goal of the type system will be to prove such an unrolling is safe—that execution of n copies of the loop body in sequence will not cause an ordering error.

3.3 Type Checking

The central goal of the type system is to ensure that the stages of the pipeline are accessed in order, though there are auxiliary goals as well, such as ensuring that vectors are not indexed out of bounds and that operations are applied to arguments of appropriate type.

$$\textsc{Deref-1}\;\frac{M, z, e \longrightarrow M', z', e'}{M, z, !e \longrightarrow M', z', !e'} \qquad \textsc{Deref-2}\;\frac{z \le z_e}{M, z, !\mathtt{addr}(z_e) \longrightarrow M, S(z_e), M[z_e]}$$

$$\textsc{Update-1}\;\frac{M, z, e_1 \longrightarrow M', z', e_1'}{M, z, e_1 := e_2 \longrightarrow M', z', e_1' := e_2} \qquad \textsc{Update-2}\;\frac{M, z, e \longrightarrow M', z', e'}{M, z, v := e \longrightarrow M', z', v := e'}$$

$$\textsc{Update-3}\;\frac{z \le z_e}{M, z, \mathtt{addr}(z_e) := v \longrightarrow M[z_e := v], S(z_e), ()}$$

$$\textsc{Vector}\;\frac{M, z, e_0 \longrightarrow M', z', e_0'}{M, z, \mathtt{vector}(v_0, \dots, v_n, e_0, \dots, e_m) \longrightarrow M', z', \mathtt{vector}(v_0, \dots, v_n, e_0', \dots, e_m)}$$

$$\textsc{Index-1}\;\frac{M, z, e \longrightarrow M', z', e'}{M, z, e[n] \longrightarrow M', z', e'[n]} \qquad \textsc{Index-2}\;\frac{n \le m}{M, z, \mathtt{vector}(v_0, \dots, v_m)[n] \longrightarrow M, z, v_n}$$

$$\textsc{Loop}\;\frac{}{M, z, \mathtt{for}\ b < n\ \mathtt{do}\ e \longrightarrow M, z, e[0/b]; \ldots; e[n-1/b]; ()}$$

$$\textsc{Comp}\;\frac{}{M, z, [e\ \mathtt{for}\ b < n] \longrightarrow M, z, \mathtt{vector}(e[0/b], \dots, e[n-1/b])}$$

$$\textsc{App-1}\;\frac{M, z, e_1 \longrightarrow M', z', e_1'}{M, z, e_1\ [\overline{k}, \overline{\ell}]\ e_2 \longrightarrow M', z', e_1'\ [\overline{k}, \overline{\ell}]\ e_2} \qquad \textsc{App-2}\;\frac{M, z, e_2 \longrightarrow M', z', e_2'}{M, z, v_1\ [\overline{k}, \overline{\ell}]\ e_2 \longrightarrow M', z', v_1\ [\overline{k}, \overline{\ell}]\ e_2'}$$

$$\textsc{App-3}\;\frac{v_1 = \mathtt{fun}\ \ [\overline{\kappa}, \overline{\alpha}]\ (x : \tau, \ell) \longrightarrow e_{body}}{M, z, v_1\ [\overline{k}, \overline{\ell}]\ v_2 \longrightarrow M, z, e_{body}[v_2/id][\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]}$$

Fig. 10. Pipeline Semantics

3.3.1 Typing Environments. The typing environment, $\Omega = (\mathbb{G}, \Delta, \mathbb{K}, \Gamma)$, consists of:

- $\mathbb{G}$, the global persistent state, a partial map from concrete locations z to base types;
- Δ, a set of location and unbounded size variables (αs and κs) that are currently in scope;
- $\mathbb{K}$, a mapping from bounded sizes b to their upper bound, a size (with $\mathbb{K}$ written as a sequence of inequalities $b_1 < k_1, \dots, b_n < k_n$);
- Γ, a mapping from value identifiers to types;

We often refer to part of the environment using dot notation (*e.g.*, $\Omega.\mathbb{G}$). We use the notation $\Omega.(\ldots)$ to denote Ω with one of its fields replaced by the body of the parentheses, e.g. $\Omega.(\Delta \cup \Delta')$ replaces Δ with $\Delta \cup \Delta'$. We use the metavariable Σ to range over environments in which all but the first entry are empty; that is, Σ is an environment with the form $(\mathbb{G}, \emptyset, \emptyset, \emptyset)$.

3.3.2 Well-Formedness. The locations, sizes and types manipulated by the type checker must be well-formed, that is, any free variables must be declared in the type checking environment. We write $\Delta, \mathbb{K} \vdash k$ and $\Delta, \mathbb{K} \vdash \ell$ when the free variables of k and ℓ are contained in Δ and the domain of $\mathbb{K}$. We say $\mathbb{K}$ is well-formed with respect to Δ, written $\Delta \vdash \mathbb{K}$ under the following conditions.

$$\frac{}{\Delta \vdash \emptyset} \qquad \frac{\Delta \vdash \mathbb{K} \qquad b \notin \mathsf{Dom}(\mathbb{K}) \qquad \Delta, \mathbb{K} \vdash k}{\Delta \vdash \mathbb{K}, b < k}$$

We use similar notation (*e.g.*, $\Delta, \mathbb{K} \vdash t$, $\Delta, \mathbb{K} \vdash \tau$, and $\Delta, \mathbb{K} \vdash \Gamma$) to describe well-formedness of other objects. Likewise, we write $\Omega \vdash k$ when $\Omega.\Delta, \Omega.\mathbb{K} \vdash k$ and again similarly for other objects. The formal definition is standard; a complete set of well-formedness rules appears in appendix B in the auxiliary archive.

We impose additional well-formedness conditions on function types. The conditions represent useful properties of the type system, which we wish to ensure are respected by any type annotations in the program. The conditions are not strictly necessary — allowing programs with ill-formed type annotations would not violate soundness — but enforcing the conditions allows us to prove properties of the system modularly.

Definition 3.1 (Well-Formed Types). If $t = \texttt{fun } \forall\overline{\kappa}, \overline{\alpha}.C_f \Rightarrow (\tau_{in}, \ell_{in}) \longrightarrow (\tau_{out}, \ell_{out})$, in order to show $\Omega \vdash t$ we additionally require that

- (monotonicity) C_f implies the constraint $\ell_{in} \leq \ell_{out}$; that is $C_f \Rightarrow \ell_{in} \leq \ell_{out}$, and
- (well-constrained) For every atomic constraint $x \leq y$ in C_f, $C_f \Rightarrow \ell_{in} \leq x \leq y \leq \ell_{out}$.

We impose an additional well-formedness condition on $\mathbb{G}$ as well. Intuitively, $\mathbb{G}$ represents the locations in memory where values are stored; that is, $\mathbb{G}$ should contain entries for each leaf node in the heap. For example, a $\mathbb{G}$ representing the heap in figure 7 would have four entries: 0.0, 0.1, 1.0, and 1.1. Our well-formedness condition requires that no entry in $\mathbb{G}$ is a parent or child of another entry. If $\mathbb{G}$ did contain two entries, one a parent of the other, then intuitively the data in those two entries would "overlap." Such constructions do not conform to our mental model of how heaps should be structured and do not arise in practice, though admitting such artificial structures would not actually compromise the soundness of the system.

Definition 3.2 (Well-Formed Globals). A global map $\mathbb{G}$ is well-formed, written $\vdash \mathbb{G}$, if for any two concrete locations z_1, z_2 where z_1 is a strict prefix of z_2, at most one of $\mathbb{G}[z_1], \mathbb{G}[z_2]$ exists.

3.3.3 Constructing Global Maps. In the rest of this paper, we assume that global maps $\mathbb{G}$ are simply handed to us. However, when checking real programs, we must construct the maps ourselves. Fortunately, we can do so easily by processing global declarations one-by-one at the beginning of the program. For example, to construct the map for a program that begins with

```
global int g1 = ...;
global (int, bool) g2 = ...;
global int[4] g3 = ...;
```

we would add entries for the locations 0, 1.0, 1.1, 2.0, 2.1, 2.2, and 2.3. Notice that this map adheres to our well-formedness condition.

3.3.4 Expression Typing. The typing judgement for expressions has the form $\Omega, \ell_{in} \vdash e : \tau, \ell_{out}, C$. Here, τ is the type of expression e, ℓ_{in} denotes our place in the pipeline prior to execution of e, while ℓ_{out} denotes our place in the pipeline after execution of e. C contains any ordering constraints required for e to be safe to execute. Figures 11 and 12 present the typing rules.

Unit
$$\frac{\Omega \vdash \ell'}{\Omega, \ell \vdash () : \texttt{Unit}\langle \ell' \rangle, \ell, \texttt{true}}$$

True
$$\frac{\Omega \vdash \ell'}{\Omega, \ell \vdash \texttt{true} : \texttt{Bool}\langle \ell' \rangle, \ell, \texttt{true}}$$

False
$$\frac{\Omega \vdash \ell'}{\Omega, \ell \vdash \texttt{false} : \texttt{Bool}\langle \ell' \rangle, \ell, \texttt{true}}$$

Addr
$$\frac{\Omega.\mathbb{G}[z] = T}{\Omega, \ell \vdash \texttt{addr}(z) : \texttt{addr}(T)\langle z \rangle, \ell, \texttt{true}}$$

Var
$$\frac{\Omega.\Gamma[id] = \tau}{\Omega, \ell \vdash id : \tau, \ell, \texttt{true}}$$

Pair
$$\frac{\Omega, \ell_0 \vdash e_1 : t_1\langle \ell.0 \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : t_2\langle \ell.1 \rangle, \ell_2, C_2}{\Omega, \ell_0 \vdash (e_1, e_2) : (t_1, t_2)\langle \ell \rangle, \ell_2, C_1 \wedge C_2}$$

Fst
$$\frac{\Omega, \ell_0 \vdash e : (t_1, t_2)\langle \ell \rangle, \ell_1, C_1}{\Omega, \ell_0 \vdash \texttt{fst}\ e : t_1\langle \ell.0 \rangle, \ell_1, C_1}$$

Snd
$$\frac{\Omega, \ell_0 \vdash e : (t_1, t_2)\langle \ell \rangle, \ell_1, C_1}{\Omega, \ell_0 \vdash \texttt{snd}\ e : t_2\langle \ell.1 \rangle, \ell_1, C_1}$$

Let
$$\frac{\Omega, \ell_0 \vdash e_1 : \tau_1, \ell_1, C_1 \qquad \Omega.(\Gamma[id := \tau_1]), \ell_1 \vdash e_2 : \tau_2, \ell_2, C_2}{\Omega, \ell_0 \vdash \texttt{let}\ id = e_1\ \texttt{in}\ e_2 : \tau_2, \ell_2, C_1 \wedge C_2}$$

If-left
$$\frac{\Omega, \ell_0 \vdash e_1 : \texttt{Bool}\langle \ell \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : \tau, \ell_2, C_2 \qquad \Omega, \ell_1 \vdash e_3 : \tau, \ell_3, C_3 \qquad \ell_2 \leq \ell_3}{\Omega, \ell_0 \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, \ell_3, C_1 \wedge C_2 \wedge C_3}$$

If-right
$$\frac{\Omega, \ell_0 \vdash e_1 : \texttt{Bool}\langle \ell \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : \tau, \ell_2, C_2 \qquad \Omega, \ell_1 \vdash e_3 : \tau, \ell_3, C_3 \qquad \ell_3 \leq \ell_2}{\Omega, \ell_0 \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, \ell_2, C_1 \wedge C_2 \wedge C_3}$$

Abs
$$\frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \\ \Delta' = \Omega.\Delta \cup \overline{\kappa} \cup \overline{\alpha} \qquad \Delta', \mathbb{K} \vdash \tau_{in}, \ell_{in} \qquad (\mathbb{G}, \Delta', \mathbb{K}, \Gamma[id := \tau_{in}]), \ell_{in} \vdash e : \tau_{out}, \ell_{out}, C \\ t_f = \forall \overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau_{in}, \ell_{in}) \longrightarrow (\tau_{out}, \ell_{out}) \qquad \Omega \vdash \ell' \qquad \Omega \vdash t_f\end{array}}{\Omega, \ell \vdash \texttt{fun}\ \ [\overline{\kappa}, \overline{\alpha}](id : \tau_{in}, \ell_{in}) \longrightarrow e : t_f\langle \ell' \rangle, \ell, \texttt{true}}$$

App
$$\frac{\begin{array}{c}\Omega \vdash \overline{k}, \overline{\ell} \qquad \Omega, \ell_0 \vdash e_1 : t_f\langle \ell' \rangle, \ell_1, C_1 \\ t_f = \forall \overline{\kappa}, \overline{\alpha}.C_f \Rightarrow (\tau_{in}, \ell_{in}) \longrightarrow (\tau_{out}, \ell_{out}) \qquad \Omega, \ell_1 \vdash e_2 : \tau_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_2, C_2\end{array}}{\Omega, \ell_0 \vdash e_1\ [\overline{k}, \overline{\ell}]\ e_2 : \tau_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], C_1 \wedge C_2 \wedge C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \wedge \ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]}$$

Fig. 11. Expression Typing: Values, Conditionals, Functions

Part 1: Values, Functions, and Conditionals. Figure 11 presents the rules for values, variables, pairs, functions, let and if statements. Notice that the beginning and ending locations for values are always the same—they have no effect on the state of the pipeline. For uniformity, base types (`Unit` and `Bool`), are associated with a location ℓ'. However, these locations are artificial—only mutable globals need be assigned a stage for storage—and hence the location assigned may be arbitrary. On the other hand, the global stored at address `addr`(z) (see rule Addr) is given a type that includes its location. Values may appear anywhere and hence never directly give rise to any ordering constraints (the generated constraints C are always simply `true`).

Pairs, let expressions and if statements all involve execution of multiple expressions, and may see the current pipeline location advance from ℓ_0 to ℓ_1 to ℓ_2, *etc.*, as subexpressions are executed. The

Deref
$$\frac{\Omega, \ell_0 \vdash e : \texttt{addr}(T)\langle \ell_2 \rangle, \ell_1, C \qquad \Omega \vdash \ell'}{\Omega, \ell_0 \vdash !e : T\langle \ell' \rangle, S(\ell_2), C \wedge \ell_1 \leq \ell_2}$$

Update
$$\frac{\Omega, \ell_0 \vdash e_1 : \texttt{addr}(T)\langle \ell_3 \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : T\langle \ell \rangle, \ell_2, C_2 \qquad \Omega \vdash \ell'}{\Omega, \ell_0 \vdash e_1 := e_2 : \texttt{Unit}\langle \ell' \rangle, S(\ell_3), C_1 \wedge C_2 \wedge \ell_2 \leq \ell_3}$$

Vector
$$\frac{\Omega, \ell_0 \vdash e_1 : t\langle \ell_v.0 \rangle, \ell_1, C_1 \qquad \cdots \qquad \Omega, \ell_{n-1} \vdash e_n : t\langle \ell_v.(n-1) \rangle, \ell_n, C_n}{\Omega, \ell_0 \vdash \texttt{vector}(e_1, \ldots, e_n) : \texttt{vector}(t, n)\langle \ell_v \rangle, \ell_n, C_1 \wedge \cdots \wedge C_n}$$

Index-const
$$\frac{\Omega, \ell_0 \vdash e : \texttt{vector}(t, n')\langle \ell \rangle, \ell_1, C \qquad n < n'}{\Omega, \ell_0 \vdash e[n] : t\langle \ell.n \rangle, \ell_1, C}$$

Index-var
$$\frac{\Omega, \ell_0 \vdash e : \texttt{vector}(t, k)\langle \ell \rangle, \ell_1, C \qquad \Omega.\mathbb{K}[b] = k}{\Omega, \ell_0 \vdash e[b] : t\langle \ell.b \rangle, \ell_1, C}$$

Loop
$$\frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \qquad \alpha_{start} \notin \Delta \qquad \Omega \vdash k \qquad \mathbb{G}, \Delta, (\mathbb{K}, b < k), \Gamma, \alpha_{start} \vdash e : \tau, \ell_{end}, C \\ \texttt{nri}(C, b) \qquad C_0 = C[\ell_{init}/\alpha_{start}][0/b] \qquad \ell_1 = \ell_{end}[\ell_{init}/\alpha_{start}][0/b] \\ C_1 = C[\ell_1/\alpha_{start}][1/b] \qquad \ell_2 = \ell_{end}[\ell_{init}/\alpha_{start}][1/b] \qquad C_2 = C[\ell_2/\alpha_{start}][2/b]\end{array}}{\Omega, \ell_{init} \vdash \texttt{for}\ b < k\ \texttt{do}\ e : \texttt{Unit}\langle \ell \rangle, \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b), C_0 \wedge C_1 \wedge C_2}$$

Comp
$$\frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \qquad \alpha_{start} \notin \Delta \qquad \Omega \vdash k \qquad \mathbb{G}, \Delta, (\mathbb{K}, b < k), \Gamma, \alpha_{start} \vdash e : t\langle \ell_v.b \rangle, \ell_{end}, C \\ \texttt{nri}(C, b) \qquad C_0 = C[\ell_{init}/\alpha_{start}][0/b] \qquad \ell_1 = \ell_{end}[\ell_{init}/\alpha_{start}][0/b] \\ C_1 = C[\ell_1/\alpha_{start}][1/b] \qquad \ell_2 = \ell_{end}[\ell_{init}/\alpha_{start}][1/b] \qquad C_2 = C[\ell_2/\alpha_{start}][2/b]\end{array}}{\Omega, \ell_{init} \vdash [e\ \texttt{for}\ b < k] : \texttt{vector}(t, k)\langle \ell_v \rangle, \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b), C_0 \wedge C_1 \wedge C_2}$$

Fig. 12. Expression Typing: State, Vectors, Loops

resulting location of an if-statement is the greater of the two locations of its branches (locations will be bypassed if one branch uses a location and another does not).

Functions abstract over polymorphic location and size variables and capture the constraints a caller must satisfy to call them. Rules Abs and App are relatively standard, although the last constraint of the App rule allows locations to be skipped to match the function's input location.

Part 2: State, Vectors, and Loops. Figure 12 presents rules for checking state, vectors and loops.

In the Deref rule, the current location has advanced to ℓ_1 just prior to derefence. Hence, one must prove the address accessed (ℓ_2) appears later than ℓ_1 in the pipeline (the constraint added in the conclusion of the rule). After execution of the expression, the current location will be the successor of ℓ_2. Because the value returned from the read has a base type, the location ℓ' associated with it is irrelevant and may be chosen arbitrarily. The Update rule follows a similar pattern.

When checking indexing operations, the key is to ensure indices are in bounds. Fortunately, patterns for using vectors in Lucid2 programs are limited, so simple bounds checking rules suffice. The rule Index-const allows constants to be used to index vectors of known length and checks that the index n is less than the vector length n'. In rule Index-var, variables b may index vectors only when the bound on b (given by $\mathbb{K}$) is equal to the length of the vector. This latter rule allows simple loops to iterate over vectors one location at a time, the common case in our suite of applications.

Notice that these rules do not affect the final location, because vectors are not themselves global values.

The most interesting rules are the rules for loops (Loop) and comprehensions (Comp). The Loop rule analyzes the loop body e, as if it starts from some arbitrary location α_{start} and with respect to a loop index variable b. Doing so generates a collection of constraints C that is parametric in α_{start} and b. Three instances of C are then created, C_0, C_1, and C_2, representing the constraints that would be generated on the 0^{th}, 1^{st}, and 2^{nd} iterations of the loop. The premise $\texttt{nri}(C, b)$ requires that all locations ℓ appearing in C contain at most one occurrence of b (for example, the location $0.b.1.b$ would be disallowed; see §3.4 for a more detailed explanation). So long as it is satisfied, it suffices to only check C_0, C_1 and C_2. If they are consistent, then the loop is safe to execute—there will be no ordering violations regardless of the number of iterations of the loop at run time. We sketch the proof of this property in §4; a full proof can be found in the auxiliary archive.

To determine the current location after execution of the loop, we take the effect at the end of the loop body, $\ell_{end}[\ell_{init}/\alpha_{start}]$, and we "round up" past b. For instance, if we were just iterating over locations 0.0.0, 0.0.1, 0.0.2, ... *etc.*, which are all captured parametrically as $0.0.b$, then this rounding operation advances us past all such indices to location 0.1 by "rounding up," or chopping off everything after b and moving to the successor location.

The Comp rule governs type checking of vector comprehensions. It too is an iterative construct and hence inherits much of the complexity of the Loop rule.

3.4 Limitations

Like most type systems, Lucid2 is incomplete: there exist programs that execute without error, but which fail to type check. One example of incompleteness arises while checking if-statements. Expressions like the following one will not type check when the relation between locations of `x` and `y` is unknown.

```
if ... then !x else !y
```

We considered adding a "max" operator to serve as a join for our algebra of locations ($\texttt{max}(\ell_1, \ell_2)$ being the larger of the two locations), but doing so appeared to complicate type inference, and did not appear worth the effort at the moment: in practice, we have not yet developed any applications that would benefit from such an extension.

One other source of incompleteness arises in the Loop and Comp rules, where the premise $\texttt{nri}(C, b)$ rules out programs that use the same index variable twice, as in the expression `g[i][i]`. The following program fragment demonstrates why this is necessary:

```
1  for i < 10 {
2    !g[i][i]; // Double indexing -- eventually we'll try to access g[6][6]
3    !g[i][5]; // Single indexing -- eventually we'll try to access g[6][5]
4  }
```

This program would succeed for the first five iterations, but fail on the sixth. That is, it is *not* sufficient to check only the first three iterations of this loop. The $\texttt{nri}(C, b)$ premise serves to weed out these examples. This restriction does rule out some legitimate programs – e.g. the above example with line 3 commented out. However, while there are applications that iterate through elements of a vector, we have not seen any that iterate along a diagonal like this. So again, this limitation does not appear to have any practical impact.

4 PROPERTIES OF LUCID 2.0

In this section, we discuss selected properties of Lucid 2.0, primarily those involving locations, and finish with a statement of soundness. Proofs of each property are available in appendices C and D in the auxiliary archive.

Value Lemma. The following lemma states that values are inert; they do not have an effect on the world or generate constraints. They can appear anywhere in the pipeline.

Lemma 4.1 (Value Lemma). *If* $\Omega, \ell \vdash v : \tau, \ell', C$, *then*

- *(V-1)* $\ell = \ell'$ *and* $C = \texttt{true}$.
- *(V-2) For all* ℓ, *we have* $\Omega, \ell \vdash v : \tau, \ell, C$.

Location Weakening. Intuitively, the following lemma states that if we can typecheck an expression from a given location, we can also typecheck it from any earlier location. This is exactly as we would expect, since starting execution from an earlier location in the pipeline gives us access to all the same data as before.

Lemma 4.2 (Location Weakening). *Assume* $\vdash \Omega$ *and* $\Omega, \ell_{start}, \vdash e : \tau, \ell_{end}, C$ *where* $\models C$. *Then for all* $\ell'_{start} \leq \ell_{start}$, *then there is some* $\ell'_{end} \leq \ell_{end}$ *such that* $\Omega, \ell'_{start}, \vdash e : \tau, \ell'_{end}, C'$, *where* $\models C'$. *Furthermore, either* $\ell'_{end} = \ell_{end}$ *or* $\ell'_{end} = \ell'_{start}$.

Monotonicity. When the constraints generated from an expression hold, computations are guaranteed to move forward in the pipeline. The monotonicity property establishes this fact.

Lemma 4.3 (Monotonicity). *If* $\vdash \Omega$, *and* $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, *then* $C \implies \ell_{start} \leq \ell_{end}$.

Bounded Constraints. The following lemma is the first step in proving properties of loops. It allows us to connect the starting and ending location of a typing judgement with the constraints generated by that judgement.

Lemma 4.4 (Bounded Constraints). *If* $\vdash \Omega$, *and* $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, *then for each constraint* $x \leq y \in C$ *we have* $C \implies \ell_{start} \leq x \leq y \leq \ell_{end}$.

Loop Unrolling. If a loop survives three iterations, it will survive arbitrarily many more; the following lemma is key to proving this fact. Since it is such an important property, we provide a high-level proof sketch as well as the statement of the lemma.

Lemma 4.5 (Loop Unrolling). *Assume* $\vdash \Omega$ *and* $\Omega, \alpha_{start} \vdash e : \tau, \ell_{end}, C$. *For all locations* ℓ_{init} *and bounded sizes i, define* $\ell_0 = \ell_{init}$, $C_0 = C[\ell_0/\alpha_{start}][0/i]$ *and for* $j > 0$ *define* $\ell_j = \ell_{end}[\ell_{j-1}/\alpha_{start}][(j-1)/i]$ *and* $C_j = C[\ell_j/\alpha_{start}][j/i]$. *Finally, assume* nri*(C, i). Then if M is a model of* $C_0 \wedge C_1 \wedge C_2$, *M is also a model of* $\forall j \geq 0.C_j$.

We prove this lemma by fixing a model M, then showing that for each constraint $x \leq y \in C$, $x[j/i] \leq y[j/i]$ for all $j > 0$. To do so, we use the fact that the initial location of loop iteration $j+1$ is the same as the final location of iteration j. Together with the Bounded Constraints lemma, this lets us conclude that $x[j/i] \leq y[j/i] \leq x[j+1/i] \leq y[j+1/i]$, so long as we know that the left- and right-most inequalities hold separately. We know they do when $j = 1$, since M satisfies C_1 and C_2, and so we use the fact that $y[1/i]$ is "sandwiched" between $x[1/i]$ and $x[2/i]$ (and similarly for

$x[2/i]$) to perform a case analysis on the structure of x and y that shows the inequality will always hold regardless of j.

An astute reader might wonder why we chose to use C_1 and C_2 rather than C_0 and C_1. This stems from the fact that the initial location of the loop iteration may appear in constraints, and may not always have the same form between iterations; if it does not, the sandwiching technique fails. While the initial location of each iteration after the first follows a set pattern, the initial location of the first iteration is determined by the code *before* the loop, and hence may differ from the following iterations. Thus we can relate the initial locations of iterations 1 and 2, but not of iterations 0 and 1. This may be a limitation of our proof technique, as in practice, we know of no loops that succeed for two iterations but fail on the third. However, it is not a costly limitation—our type checker can analyze any of our benchmarks in under two seconds (see §5.4).

Memory Typing. Execution through the pipeline will proceed without error provided the state associated with the pipeline has the expected structure. The following definition describes the required relation between memories M and global specificiations $\mathbb{G}$. When the $\mathbb{G}$ in question is clear from context, we may omit it and simply say "M is well-formed."

Definition 4.6. M is well-formed with respect to $\mathbb{G}$, written $M \sim \mathbb{G}$, when it satisfies the following properties.

- $M[z]$ exists if and only if $\mathbb{G}[z]$ exists, and
- if $M[z] = v$ and $\mathbb{G}[z] = T$ then for all Ω, ℓ, ℓ', we have $\Omega, \ell \vdash v : T\langle \ell' \rangle, \ell, \texttt{true}$

Soundness. The prior lemmas constitute the scaffolding on which we can prove a soundness theorem based on progress and preservation.

Theorem 4.7 (Progress). *Let $\Sigma, z \vdash e : \tau, z', C$ where $\vDash C$. Let $M \sim \Sigma.\mathbb{G}$. Then either e is a value or there are some M', z'', e' such that $M, z, e \longrightarrow M', z'', e'$.*

Theorem 4.8 (Preservation). *Let $\Sigma, z_{start} \vdash e : \tau, z_{end}, C$ and $M, z_{start}, e, \longrightarrow M', z_{step}, e'$, where $\vDash C$ and $M \sim \Sigma.\mathbb{G}$. Then $M' \sim \Sigma.\mathbb{G}$, and $\Sigma, z_{step} \vdash e' : \tau, z'_{end}, C'$, where $\vDash C'$ and $z'_{end} \leq z_{end}$.*

5 IMPLEMENTATION AND EVALUATION

We implemented Lucid2 in OCaml as an extension to Lucid1. Our implementation consists of (1) language extensions for polymorphism, constraints, records, abstract types, first order modules, vectors, and loops; (2) an extended type checker that implements the rules in §3; (3) type, location and constraint inference, and (4) compile-time transformations that eliminate each language extension, reducing the extended language back to Lucid1 for the rest of the Lucid1 system to compile to the Intel Tofino. The implementation contains a number of practically important, but theoretically straightforward extensions to the idealized language defined in the prior section, including, for example, mutable arrays rather than single-cell references (*i.e.*, the `addr` type), the adoption of an imperative C-like syntax, and the creation of a simple module system with abstract types, events and event handlers. We discus a few issues that arose in the implementation below.

5.1 Type Inference and Constraint Checking

We have implemented type, effect, and size inference using an analogue of Algorithm J (Milner [1978]). The structure of locations (in particular, the unary representation of numbers) was carefully chosen to be amenable to unification-based type inference. A key aspect of type inference involves checking satisfiability of constraints. Satisfiability queries are implemented by transforming effects into their list-based representation from §3.1 and then encoding constraints in a decidable fragment

$$\texttt{select}_\ell(i) = \begin{cases} 0 & i = 0 \\ B_b + 2 & i = 1 \\ 1 & i = 2 \\ -1 & \text{otherwise} \end{cases} \qquad \texttt{select}_\ell(i) = \begin{cases} \texttt{select}(A_\alpha, i) & 0 \leq i < L_\alpha - 1 \\ \texttt{select}(A_\alpha, i) + n & i = L_\alpha - 1 \\ \texttt{select}_{\ell'}(i) & L_\alpha \leq i < L_\alpha + \texttt{len}(\ell') \\ -1 & \text{otherwise} \end{cases}$$

(a) (b)

Fig. 13. select functions for ℓ when (a) $\ell = 0.(b+2).1$ and (b) hd $\ell = (\alpha + n)$ and tl $\ell = \ell'$

of the theory of arrays, which we check using Z3 (de Moura and Bjørner [2008]). We describe the encoding in §5.3. Although we run a large number of queries per program (once per function call), each one is typically small enough that we get good performance nonetheless (see §5.4).

5.2 Events and Handlers

Our formal language omits recursion, and our implementation is similar, since the switch hardware cannot implement unbounded recursion in a single pass through a pipeline. However, recursive programs can be implemented via the *packet recirculation* mechanism available on the Tofino chip, which directs packets exiting the chip back to the beginning of the pipeline. Recirculation is made available to programmers via events and event handlers, and hence, event handlers are effectively mutually recursive with one another. Rather than attempting to infer constraints for handlers, we opted to require user-supplied constraint annotations when events are declared. We check that the constraints hold whenever a new event of the given type is generated, and assume the constraints in the body of the event handler when it receives such an event. For instance, we might declare an event foo as follows.

```
event [x <= y] foo(array<bool> x, array<bool> y);
```

Doing so mandates the system prove `x <= y` when an event is generated, and allows a foo-handler to assume `x <= y`. In other words, these events are a form of dependent pair.

These constraints place some annotation burden on the programmer, but the burden is minimal and the explicit annotations serve as useful documentation. In practice, many events do not require constraint annotations at all—they are typically only required when an event takes multiple global variables as parameters, which is rare. In most cases, we can typecheck the body without any assumptions about the order of the parameters.

5.3 SMT Encoding

We encode locations using Z3's Array sort, using a strategy inspired by (Bradley et al. [2006]). Z3 Arrays are essentially infinite integer lists; we embed our (finite) lists into these by setting all other entries to -1.

Specifically, we encode each location ℓ as a function $\texttt{select}_\ell$ such that $\texttt{select}_\ell(i)$ is the ith element of ℓ. For concrete locations, and those which contain only bounded variables, the encoding is straightforward. For each bounded variable b, we introduce a new Int-Sort SMT variable B_b, constrain it to be nonnegative, and return it from the select function as necessary. For example, if $\ell = 0.(b+2).1$, we would add a new variable B_b, a new constraint $B_b \geq 0$, and define $\texttt{select}_\ell$ as in Figure 13 (a). This is easily represented in SMT as a nested if-then-else expression.

The tricky part is encoding locations that begin with a location variable $\alpha + n$. Since α represents a location, we have to encode it as an Array-sort variable. In fact, we create two new variables: A_α and L_α, where A_α represents α itself and L_α is an Int-sort variable representing the length of A_α.

Table 1. Modules implemented in Lucid2. All make heavy use of polymorphism, records, and vectors. When one module builds on other modules, we indicate the additional lines of code (LoC) with a +.

Module	Description	LoC	Typing time (sec)
Bloom Filter	Probabilistic set of elements.	53	0.26
+Aging	Entries time out	+74	+0.44
Hash table	Deterministic set of elements	25	0.10
+Cuckoo hashing	Contains multiple stages to deal with collisions	+45	+0.22
Hash table w/ timeout	Deterministic set of elements, plus the time each was last touched	65	0.38
+Cuckoo hashing	Contains multiple stages, and clears timed-out entries automatically	+81	+0.31
Bidirectional Map	Stores lists of integers in an array, mapping each to/from its index	39	1.1
Count-min sketch	Probabilistically counts the number of times an element is accessed	70	0.45
+Aging	Entries time out	+83	+0.71

We then encode our `select` function as follows. First, we define the Z3 expression $\texttt{len}(\ell)$ to be the length of ℓ if ℓ does not begin with an α, and define $\texttt{len}(\ell) = L_\alpha + \texttt{len}(\texttt{tl}\ \ell)$ otherwise. Now assume $\textsf{hd}\ \ell = (\alpha + n)$ and $\texttt{tl}\ \ell = \ell'$. Since αs can only appear at the beginning of a location, we can encode $\texttt{select}_{\ell'}$ as in the earlier paragraph. Using `select` to denote Z3's built-in Array indexing operation, we define $\texttt{select}_\ell$ as in Figure 13 (b). We also add constraints that the result of selecting from A_α is always nonnegative, since our location lists never contain negative entries.

5.3.1 Encoding Constraints. Given our location encoding, we encode the constraint $\ell_1 < \ell_2$ as

$$\exists i < \texttt{len}(\ell_1).\ \big(\texttt{select}_{\ell_1}(i) < \texttt{select}_{\ell_2}(i) \land \forall j < i. \texttt{select}_{\ell_1}(j) = \texttt{select}_{\ell_2}(j)\big)$$

Because the existential quantifier appears at the beginning of the constraint, we may remove it via Skolemization, resulting in a query that contains only universal quantifiers. We have found this encoding works quickly without any modifications, but it is possible to remove the universal quantifiers as well, using techniques from (Bradley et al. [2006]). This shows that the problem is decidable, and empirically has been within the bounds of Z3's capabilities.

5.3.2 Encoding Implication. When typechecking recursive handlers, we need to check whether the user-supplied constraints imply the constraints of the body. This is difficult because, naïvely, the constraint $C_1 \Rightarrow C_2$ is equivalent to $\sim C_1 \lor C_2$, and introducing negation runs the risk of quantifier alternation rendering our encoding undecidable. Fortunately, there is a simple fix: the negation of the constraint $\ell_1 \leq \ell_2$ is the (positive) constraint $S(\ell_2) \leq \ell_1$. By negating our inequalities before encoding into SMT, we can encode $\sim C_1 \lor C_2$ solely in terms of positive atoms.

5.4 Programming Experience

To demonstrate the usefulness of Lucid2, we reimplemented the example applications presented in the Lucid1 paper (Sonchack et al. [2021]). To do so, we first implemented several widely-used networking data structures as stand-alone modules (listed in Figure 1), each needed by one or more applications. All of these modules utilize polymorphism, records, vectors and abstract types to provide a flexible, reusable, and abstract interface.

We found that on the whole, the example applications benefited substantially from Lucid2's extensions. Most programs used conventional data structures, which, when programming in Lucid1, had to be inlined into a monolithic application, leading to lengthy and obscure code. Once those data structures were defined as independent, reuseable modules in Lucid2, the code became clearer. In all but one case, the code became much shorter as well; the exception was the Simple NAT

Table 2. Applications implemented in Lucid2. Lines of code (LoC) is for the application alone, not including comments or the LoC for the modules on which it depends (see Figure 1 for the latter).

Application	Description	Modules Used	Lucid1 LoC	Lucid2 LoC	Typing time (sec)
Stateful Firewall	Blocks unsolicited packets.	Cuckoo Hash w/ Aging	189	37	.68
Closed-loop DNS Defense	Identify/counter DNS reflection attacks	Bloom Filter w/ Aging Cuckoo Hash w/ Aging	215	52	1.8
*Flow [Sonchack et al. 2018]	Collects packets by flow for analysis.	Vectors only	149	104	0.03
Distributed Prob. Firewall	Synchronize a firewall across multiple switches	Bloom Filter	66	39	0.28
+Aging	Entries in the firewall time out	Bloom Filter /w Aging	119	40	0.75
Simple NAT	Performs network address translation	Bidirectional Map	41	62	1.5
Historical Prob. Queries	Allows queries of frequency for traffic flows	Count-min sketch w/ Aging	93	26	1.2

application, in which the boilerplate of defining a NAT-specific module was significant compared to the original program size. We found that typechecking times were low, with even the longest example taking under 2 seconds.

A list of these programs appears in Figure 2. Lucid1 also reported three other applications (simple chain replication, single-destination RIP, and automatic rerouting), but they were either very simple or highly specialized for their particular task. We do not report on them here because they saw fewer benefits from Lucid2's new features.

6 RELATED WORK

Over the past decade, researchers have developed a number of languages for network programming. For example, Frenetic (Foster et al. [2011]) was designed to program OpenFlow controllers: Frenetic computations sat on a software server and generated lists of packet-processing rules to be sent to switches. These lists of packet-processing rules were described using their own domain-specific sublanguage. Over time, that sublanguage evolved and developed in work on NetCore (Schlesinger et al. [2014]), Pyretic (Reich et al. [2013]), and NetKAT (Anderson et al. [2014]). Other languages, like FlowLog (Nelson et al. [2014]) and Maple (Voellmy et al. [2013]) used other kinds of programming paradigms to control these OpenFlow systems at a high level of abstraction. A key distinction between earlier work based around OpenFlow, and later work based around P4, is that P4 switches contain persistent, mutable and programmable state. NetKAT (for example) is stateless and cannot describe or implement the stateful applications developed in this paper. The pipeline compilation and safety issues described in this paper do not arise in these more limited systems.

More recently, there have been a number of efforts to make programming P4 switches easier. For example, Domino (Sivaraman et al. [2016]), Chipmunk (Gao et al. [2020a]), Lyra (Gao et al. [2020b]), and P4All (Hogan et al. [2020]) allow programmers to use high-level, imperative, C-like languages to describe switch computations. They then deploy program synthesis techniques to allocate those computations to stages in the pipelines of one or more hardware devices. However, these tools provide little or no feedback when they fail to lay out computations along a pipeline. We view Lucid2's contributions to this space as complementary to, and synergistic with, these other efforts—one can certainly imagine future systems in which programmers are constrained by

Lucid2's type system, ensuring computations can be compiled, and use synthesis techniques to spread computation across one or more devices. Indeed, Lucid2's vectors and loops were inspired by related unsafe features in P4All (Hogan et al. [2020]). By incorporating appropriate elements of Lucid2's type system, P4All could deliver safe vectors, loops, and synthesis in the future.

Outside of the domain of networking, type-and-effect systems have been used to control memory access since the 80s (Gifford and Lucassen [1986]) and grew to prominence in the 90s with the work of Tofte, Talpin, Birkedal and others on region inference (Tofte and Birkedal [1998]; Tofte and Talpin [1997]). These systems protected against use-after-free errors, but did not constrain access order along a pipeline as Lucid2 does. Later, researchers developed type systems for specifying more general "resource usage protocols" (DeLine and Fahndrich [1999]; Igarashi and Kobayashi [2005]). Such systems can specify constraints on the order in which resources are used, but the protocols involved have a different character (often characterized by regular languages rather than numeric, ordered, hierarchical locations), use different technical machinery, and were targeted at different applications.

An alternative to type-and-effect systems are those type systems based on linear (Girard [1987]) or ordered logic (Polakow and Pfenning [1999b]). As mentioned earlier, ordered type systems generate similar kinds of constraints, effectively constraining the order in which data is accessed, but they have not been applied to packet processing pipelines. Moreover, to be effective they would likely need to be enriched with a variety of new features such as hierarchical locations, ordering constraints and new rules for managing vectors and loops.

7 CONCLUSION

Lucid2 is the first language to allow safe, modular programming techniques for programs which run inside packet processing pipelines. Its hierarchical, virtual pipelines, polymorphism, constraints, vectors, loops and modules make it possible to create libraries of useful data structures. Its type inference and automated constraint solving relieve programmers from unnecessary annotation burdens. Its semantics are well-defined and its metatheory is sound.

We demonstrate the utility of Lucid2 by developing a library of generic networking data structures, and using them to reimplement an existing set of applications. Most programs saw significant improvements in clarity as a result, and the library can be used for yet more applications in the future. While Lucid2 was motivated by the constraints of PISA architectures in general, and the Intel Tofino in particular, pipelined parallelism is a widely-used technique for improving the throughput of data-processing applications. Lucid2's design and type system may provide a guide for future researchers looking to deploy these ideas in the context of other network devices (Baldi [2020]; Kalkunte [2019]), other network programming languages (Gao et al. [2020b,a]; Hogan et al. [2020]; Sivaraman et al. [2016]), or other domains entirely, such as signal processing (Ebeling et al. [1996])

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. FMitF-1837030 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1837030) and Grant No. CNS-1703493 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1703493). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, The Vinh Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*. 503–514. https://doi.org/10.1145/2740070.2626316

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 113–126. https://doi.org/10.1145/2578855.2535862

Mario Baldi. 2020. Pensando Announces P4-programmable Platform and Joins P4 Community. https://opennetworking.org/news-and-events/blog/pensando-announces-p4-programmable-platform-and-joins-p4-community/.

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*. 99–110. https://doi.org/10.1145/2486001.2486011

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 427–442. https://doi.org/10.1007/11609773_28

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.5555/1792734.1792766

Rob DeLine and Manuel Fahndrich. 1999. Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi and Applications*. https://doi.org/10.1016/S1571-0661(04)80088-4

Carl Ebeling, Darren C. Cronquist, and Paul Franklin. 1996. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*. Springer-Verlag, Berlin, Heidelberg, 126–135. https://doi.org/10.5555/647923.741212

Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *ACM International Conference on Functional Programming*. 279–291. https://doi.org/10.1145/2034574.2034812

Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020b. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*. 435–450. https://doi.org/10.1145/3387514.3405879

Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020a. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*. 44–61. https://doi.org/10.1145/3387514.3405852

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. 2020. Elastic Switch Programming with P4All. In *ACM SIGCOMM HotNets Networks*. 168–174. https://doi.org/10.1145/3422604.3425933

Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A programmable system for performance-aware routing. In *USENIX Symposium on Networked Systems Design and Implementation*. 701–721.

Atsushi Igarashi and Naoki Kobayashi. 2005. Resource Usage Analysis. *ACM Trans. Program. Lang. Syst.* 27, 2 (March 2005), 264–313. https://doi.org/10.1145/1057387.1057390

Intel. 2020. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 3–14. https://doi.org/10.1145/2740070.2626292

Mohan Kalkunte. 2019. Broadcom's new Trident 4 and Jericho 2 switch devices offer programmability at scale. https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale.

Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *ACM SIGCOMM Symposium on SDN Research*. 1–12. https://doi.org/10.1145/2890955.2890968

Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium*.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *USENIX Networked Systems Design and Implementation*. 519–531. https://doi.org/10.5555/2616448.2616496

Jeff Polakow and Frank Pfenning. 1999a. Natural Deduction for Intuitionistic Non-communicative Linear Logic. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer, 295–309. https://doi.org/10.1016/S1571-0661(04)80088-4

Jeff Polakow and Frank Pfenning. 1999b. Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi and Applications*. https://doi.org/10.1016/S1571-0661(04)80088-4

J. Polokow and Frank Pfenning. 1999. Relating Natural Deduction and Sequent Calculus for Intuitionistic Non-Commutative Linear Logic. In *Fifteenth Conference on Mathematical Foundations of Progamming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999 (Electronic Notes in Theoretical Computer Science, Vol. 20)*, Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov (Eds.). Elsevier, 449–466. https://doi.org/10.1016/S1571-0661(04)80088-4

Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. 2013. Modular sdn programming with pyretic. *Technical Reprot of USENIX* (2013), 30.

Cole Schlesinger, Michael Greenberg, and David Walker. 2014. Concurrent NetCore: From Policies to Pipelines. In *ACM International Conference on Functional Programming*. 11–24. https://doi.org/10.1145/2692915.2628157

Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *ACM Symposium on SDN Research*. 83–95. https://doi.org/10.1145/3373360.3380839

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*. 15–28. https://doi.org/10.1145/2934872.2934900

John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/3452296.3472903

John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *USENIX Annual Technical Conference*. 823–835. https://doi.org/10.5555/3277355.3277435

Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 724–767. https://doi.org/10.1145/291891.291894

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM*. 87–98. https://doi.org/10.1145/2534169.2486030

David Walker. 2005. *Advanced Topics in Types and Programming Languages.* The MIT Press, Chapter Substructural Type Systems, 3–44.