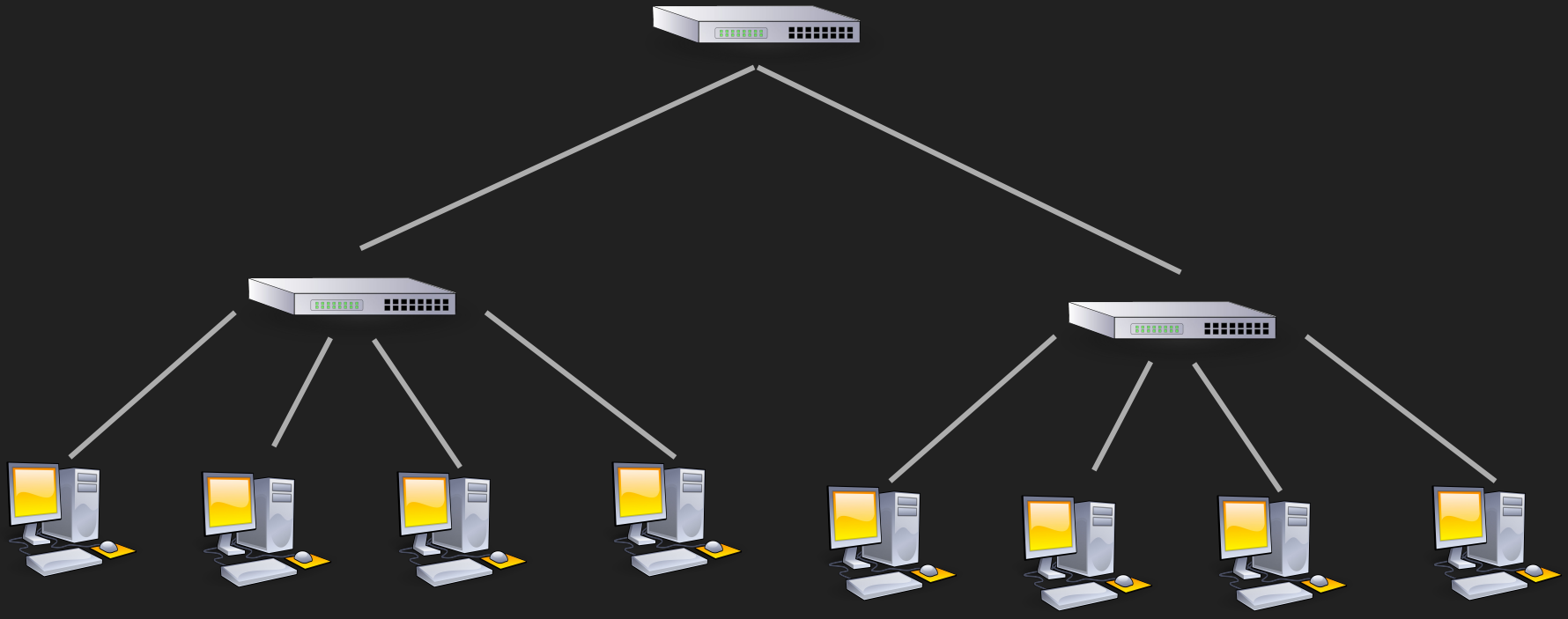
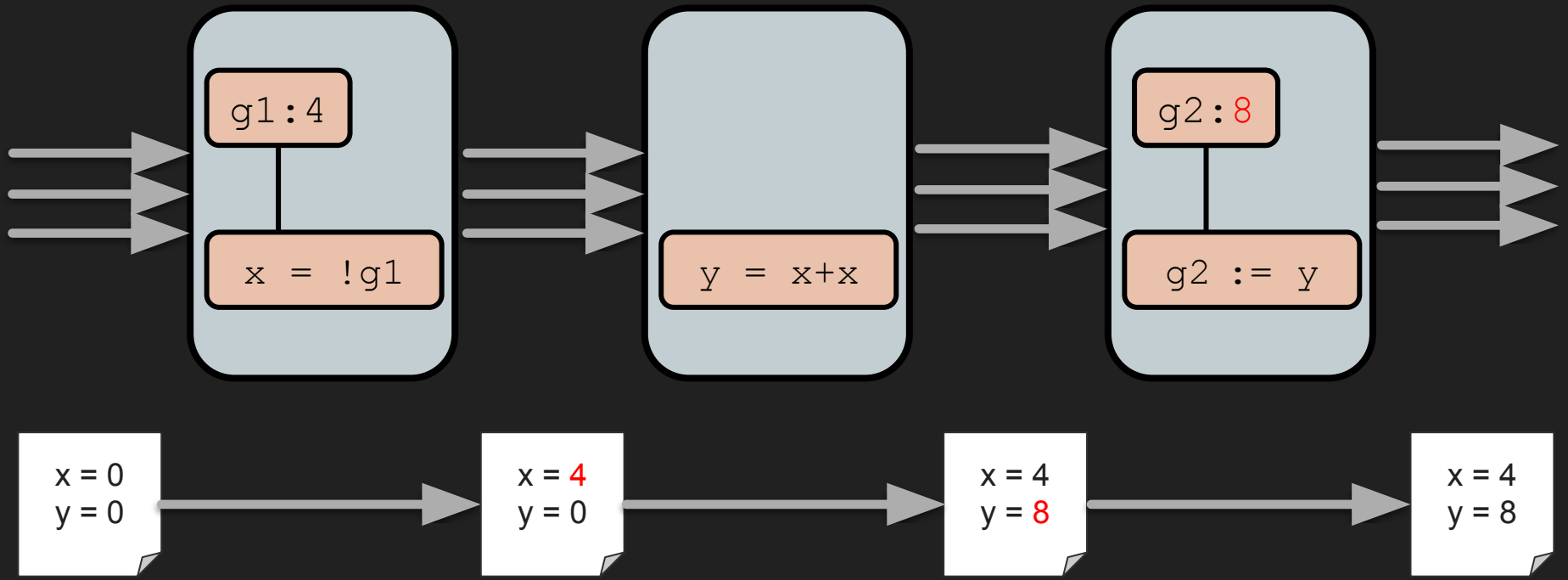
Safe, Modular Packet Pipeline Programming

Devon Loehr and David Walker



Switch Pipeline



Lucid: A Language for Control in the Data Plane

- Sonchack, Loehr, Rexford, Walker (SIGCOMM 2021)
- Provides a high-level, event-based abstraction of switches

```
1 global int g1 = 4; // Global variables persist across packets
2 global int g2 = 9;
3
4 event simple() {
5    int x = !g1;      // Read g2, store in local x
6    int y = x + x;
7     g2 := y;         // Read local y, store in g1
8 }
```

Pipeline Types: The Basics

- Key idea: annotate globals with their location in an abstract pipeline
- Track our current location in the abstract pipeline while typechecking handlers

```
1 global int g1 = 4; // Has type int@0
2 global int g2 = 9; // Has type int@1
3
4 event simple() {    // Current location: 0
5   int x = !g1;      // Current location: 1
6   int y = x + x;    // Current location: 1
7    g2 := y;         // Current location: 2
8 }
```

Pipeline Types: The Basics

- If we try to access a global after we've passed it in the pipeline, ERROR!
- Since the global order is fixed, we can point directly to the offending line

```
1 global int g1 = 4; // Has type int@0
2 global int g2 = 9; // Has type int@1
3
4 event broken() {    // Current location: 0
5   int x = !g2;      // Current location: 2
6   int y = x + x;    // Current location: 2
7    g1 := y;         // Error! Invalid access!
8 }
```

Function Types

- Function types contain both input and output types and locations
- All Lucid functions are non-recursive

```
1 // (unit, 0) -> (unit, 2)
2 fun unit copy_g1_g2() {
3    int x = !g1;
4    int y = x + x;
5     g2 := y;
6 }
7 event simple() {     // Current location: 0
8    copy_g1_g2();     // Current location: 2
9 }
```

Polymorphic Function Types

- Function types may also refer to polymorphic locations (here, a and b)

```
1 // Type (int@a * int@b, a) -> (unit, b+1)
2 fun unit copy(int@a glob1, int@b glob2) {
3   int x = !glob1;  // Current location: a
4   int y = x + x;   // Current location: a+1
5   glob2 := y;      // What if b < a?
6 }
7 event simple() {   // Current location: 0
8   copy(g1, g2);    // Current location: 2
9 }
```

First Extension: Compound, Abstract Data Types

```
module BloomFilter {
  type filter

  val add : (filter@a * int, a) -> (unit, a+1)
  val mem : (filter@a * int, a) -> (bool, a+1)
}
```

```
type filter = {
  array arr;  // Mutable, stored in stage memory
  int   seed; // Immutable, value fixed at compile-time
}
fun unit add(filter@a f, int entry) {
  int idx = hash(filter.seed, entry); // Compute an index
  set(f.arr, idx, 1);// Update arr at that index
}
```

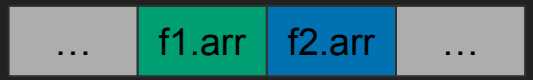
First Extension: Compound, Abstract Data Types

```
1  type filter = {
2    array arr1;  // Mutable, stored in stage memory
3    array arr2;  // Location: after arr1
4    int   seed1; // Immutable, value fixed at compile-time
5    int   seed2;
6  }
7
8  // Type (filter@a * int@b, a) -> (unit, a+2) doesn't match interface!
9  fun unit add(filter@a f, int entry) {
10   int idx1 = hash(filter.seed1, entry);
11   int idx2 = hash(filter.seed2, entry); // Current Location: a
12   set(f.arr1, idx1, 1);  // Current Location: a+1
13   set(f.arr2, idx2, 1);  // Current Location: a+2
14 }
```

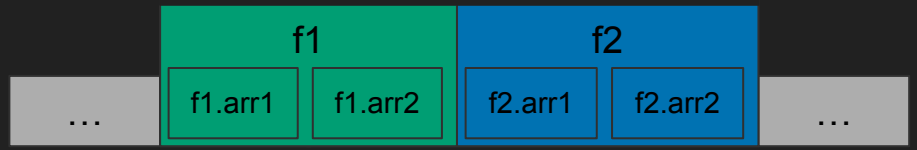
Abstracting Records

```
1 global filter f1;
2 global filter f2;
```

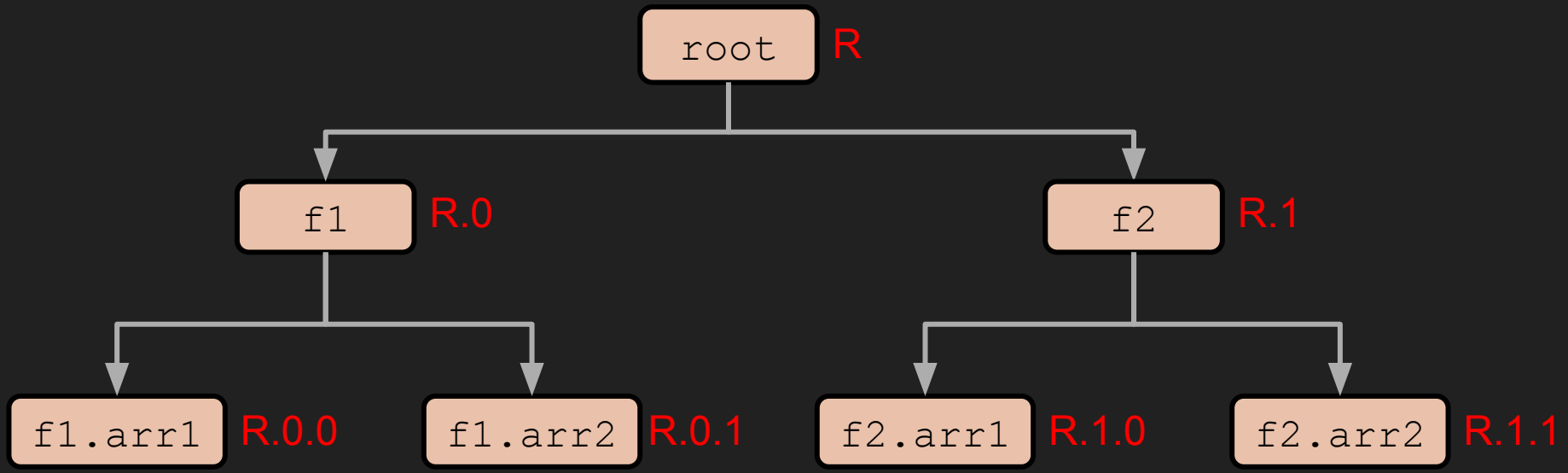
One Array



Two Arrays



Hierarchical Locations



R < R.0 < R.0.0 < R.0.1 < R.1 < R.1.0 < R.1.1

First Extension: Compound, Abstract Data Types

```
1  type filter = {
2    array arr1;  // Mutable, stored in stage memory
3    array arr2;  // Location: after arr1
4    int   seed1; // Immutable, value fixed at compile-time
5    int   seed2;
6  }
7
8   // New type: (filter@R.a * int@b, R.a) -> (unit, R.(a+1))
9  fun unit add(filter@R.a f, int entry) {
10   int idx1 = hash(filter.seed1, entry);
11   int idx2 = hash(filter.seed2, entry); //Current location: R.a
12   set(f.arr1, idx1, 1);  // Current location: R.a.1
13   set(f.arr2, idx2, 1);  // Current location: R.a.2
14 }                        // Round up to location: R.(a+1)
```

Second Extension: Vectors and Bounded Loops

- Problem: Data-plane programmers need the flexibility to make trade-offs with the size of their data structures
- Adding another array to our filter type requires re-writing all code that uses it!
- Solution: Vectors!
- Vectors are fixed-length lists of values, whose lengths are known at compile-time

Second Extension: Vectors and Bounded Loops

```
1 type filter<n> = {
2   array[n] arrs;  // Vector of n arrays, stored in memory
3   int[n]   seeds; // Vector of n ints
4 }
```

```
1 // Functions may accept filters of any size
2 fun unit add(filter<n> f, int entry) {
3   for (i < n) {
4       int idx = hash(filter.seeds[i], entry);
5       set(f.arrs[i], idx, 1);
6   }
7 }
```

Typechecking loops

- Tricky part: catching errors across iterations of a loop

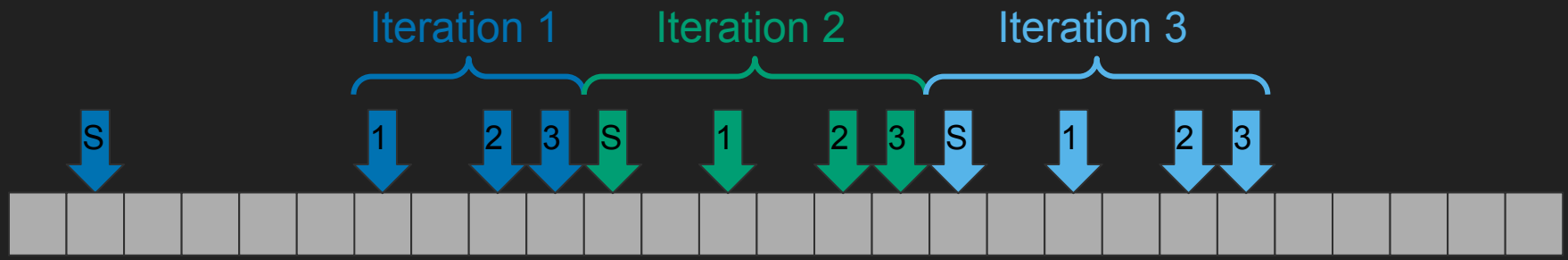
```
1 for (i < n) {
2   access(arr[0]);
3 }
4
```

```
1 for (i < n) {
2   access(arr1[i]);
3   access(arr2[i]);
4 }
```

```
1 for (i < n) {
2   for (j < m) {
3   access(arr[j][i]);
4 }}
```

Loop Unrolling Lemma

```
1 ...                          // Some accesses beforehand
2 for (i < n) {
3   access(vec[i].foo);        // Access 1
4   ...
5   access(vec[i].bar);        // Access 2
6   ...
7   access(vec[i].baz);        // Access 3
8 }
```



Related Work

- Other data-plane languages such as Domino, Chipmunk and Lyra offer high-level abstractions for programming control planes, but give little useful feedback should compilation fail.

- Pipeline types are reminiscent of substructural type systems, which enforce similar ordering constraints. However, such type systems are cumbersome in practice, and do not have an obvious way of handling vectors and loops.

Conclusion

- In SIGCOMM ‘21, we introduced Lucid’s event-driven programming model and demonstrated its applicability to a variety of networking applications

- We’ve now shown (POPL ‘22) that Lucid’s system of pipeline types is capable of handling features such as abstract datatypes and parametric vectors, which are critical for writing modular programs

- We hope that the ideas of pipeline types can find traction in other data-plane languages, and even in pipelined settings outside of networking, such as signal processing

Questions?

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

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

Polymorphic Function Types

- Limitation: the following function cannot be typed in the rudimentary system!

```
1 // Type (int@a * int@b, a) -> (unit, b+1) is too general!
2 fun unit copy(int@a glob1, int@b glob2) {
3   int x = !glob1;  // Current location: a
4   int y = x + x;   // Current location: a+1
5   glob2 := y;      // What if b < a?
6 }
7 event simple() {   // Current location: 0
8   copy(g1, g2);    // Current location: 2
9 }
```

First extension: Constraints and Polymorphism

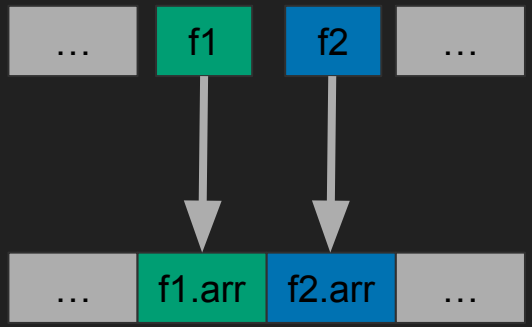
- Simple constraints allows functions to be polymorphic in location
- In practice, most constraints can be inferred

```
1 // [a < b] => (int@a * int@b, a) -> (unit, b+1)
2 fun unit copy(int@a glob1, int@b glob2) [a < b] {
3   int x = !glob1;
4   int y = x + x;
5    glob2 := y;
6
7 event simple() {    // Current location: 0
8   copy(g1, g2);     // Current location: 2
9 }
```

Abstracting Records

```
1 global filter f1;
2 global filter f2;
```

One Array



Two Arrays