Type-disciplined, Scalable, Practical Composition of Networked Services

Azer Bestavros

Joint work with
Adam Bradley, Yarom Gabay, Assaf Kfoury, Likai Liu, and Ibrahim Matta
(and maybe some of you ☺)

NRG, BU CS Department
March 13, 2006

Alternative titles/subtitles...

- Everything you wanted to know about iBench and were afraid/embarrassed to ask!
- How to (and why sometimes we have to) judge a book by its cover?
- What do you get when you cross network calculus with a type inference engine?
- What computer science problems are worthy of solving in networking?
- Could type theory research be of value to anything other than programming systems?
My agenda is not

☐ To give a 60-minute refresher course in formal type theory and inference upon which most of the formalism depends
  - OK, you’ll get some of that...

☐ To give a 60-minute crash course in the Network Calculus upon which we mostly demonstrate our approach here
  - OK, you’ll get some of that too...

My agenda is ...

☐ To note the difficulty of analyzing large, complex networks using most existing approaches to compositional analysis

☐ To propose a type-theory-inspired framework within which to explore ways of scaling up compositional analysis
Compositional analysis

- It’s good to know that a network agent (e.g., router, HTTP proxy, ...) doesn’t crash, when it’s disconnected and idle

- It’s better to know it won’t crash when connected to (composed with) another agent

- It’s even better to know it won’t crash when composed with a whole bunch of other agents in some arbitrary configuration!

A few potent examples

- Network protocols
  - What happens when TCP is “modulated” by another control schemes?

- Compatibility questions
  - Can a stream bridge two networks with similar QoS “goals” but different “mechanisms”? How about three networks?
  - Will upgrading to HTTP 1.1 break my system? (regression is hell!)

- Data plane interactions
  - Could I substitute a lossless compressor with a lossy one?

- Control plane interactions
  - Does AS1’s BGP policy compose safely with AS2’s? Does AS1+AS2 compose with “the world”?
  - Does my firewall security rules compose safely with my network monitoring infrastructure?
Compositional analysis

- A few potent examples
  - Network protocols
    - What happens when TCP is "modulated" by another control schemes?
  - Compatibility questions
    - Can a stream bridge two networks with similar QoS "goals" but different "mechanisms"? How about three networks?
    - Will upgrading to HTTP 1.1 break my system? (regression is hell!)
  - Data plane interactions
    - Can I substitute a lossless compressor with a lossy one?
  - Control plane interactions
    - Does AS1's BGP policy compose safely with AS2's? Does AS1+AS2 compose with "the world"?
    - Does my firewall security rules compose safely with my network monitoring infrastructure?

Composition:
The system Z that results from having X interact with Y

Analysis:
Formally derive safety properties of a system W

Analyzing a composition:
Derive properties of Z by analyzing the composition of X and Y

Composing the analysis:
Derive properties of Z by composing the analysis of X and the analysis of Y
Compositional analysis

- Does compositional analysis scale relative to
  - Representation size?  is it manageable?
  - Representation legibility?  is a PhD required?
  - Computational tractability?  is it feasible?

- Examples of analysis “tools”:
  - Queuing theory: What’s the biggest queuing network you were able to analyze?
  - Scheduling theory: What do you get when you use an EDF scheduler in a VM with a RR VMM dispatcher?
  - Finite state models: Welcome to the poster child of the state-space explosion!

Component model composition

\[ [A] \bullet [B] \Rightarrow [A \otimes B] \]
\[ [A \otimes B] \bullet [C \otimes D] \Rightarrow [A \otimes B \otimes C \otimes D] \]
\[ [A \otimes B \otimes C \otimes D] \bullet [E \otimes F \otimes G \otimes H] \Rightarrow [A \otimes B \otimes C \otimes D \otimes E \otimes F \otimes G \otimes H] \]

... how about an Internet-scale application ??? ...

Compositional models that preserve functional details of constituent components are not the way to go
Models vs. Components

Claim:
To scale up analysis, the representation of a composition should be of about the same size as that of its constituent components

Approach:
Make $|A \otimes B| \sim |A| \sim |B| \sim 1$
- Represent a component by its I/O signature – i.e., invariants on its interfaces
- No need to retain details that are not explicitly exposed through I/O – loss of expressiveness
- Compositions effectively "hide" interactions between components not exposed at the interface

Soundness vs. Completeness

- Sacrificing expressiveness for scalability is done so as to preserve soundness ...
  - Any theorem that we prove about a composition (e.g., property x holds or not) will be correct

- ... but may compromise completeness
  - There may be some correct theorems that we will not be able to prove – the fact we cannot prove a theorem does not mean it is not correct

- The question is how much of a gap there is between theorems we can versus cannot prove
The programming analogy...

- We are able to reason about (and hence scale) compositions of large software artifacts by hiding internals and only thinking about interfaces between modules:
  - All we care about in a library function with which we compose our code is the “signature” of that function, a.k.a. its “type specification”

- Specifying the type of an object is sufficient to use it, and to reason about what you get when you compose it with other objects
  - We want something similar for network components

Types

\[
\text{fix} \quad \lambda^T \sigma
\]

Another way of saying models, definitions, specifications, constraints, invariants, etc...
Types as constraints

- Types establish constraints on the set of acceptable inputs and promised outputs.

- The details encoded in a type/constraint represent a tradeoff between:
  - Expressiveness: what are you able to prove?
  - Feasibility: can you prove it?
  - Scalability: for what size problem?

TRAFFIC

**Typed Representation and Analysis of Flows**

For Interoperability Checks
TRAFFIC for network gadgets

- Network gadgets consume/produce inputs/outputs over multiple dimensions:
  - E.g., data plane versus control plane
  - E.g., dimensions in a grid setting, N-S & E-W

- Without loss of generality, assume network gadgets have two dimensions
  - Forward dimension (a.k.a., data flow)
  - Backward dimension (a.k.a., control flow)

TRAFFIC: Types

- Each socket has a type
  - \( r \in \text{FwSocketType} \)
  - \( s \in \text{BwSocketType} \)
  - \( t \in \text{SocketType} \) \( ::= r \mid s \)

- Sockets in a given direction make a bundle of some type
  - \( \rho \in \text{FwType} \) \( ::= r \{\rho_1 : \rho_2\} \)
  - \( \sigma \in \text{BwType} \) \( ::= s \{\sigma_1 : \sigma_2\} \)
  - \( r \in \text{Type} \) \( ::= \rho \mid \sigma \)

- Two forward and two backward bundles make up a (gadget) flow
  - \( t \in \text{FlowType} \) \( ::= [\rho_1, \rho_2; \sigma_1, \sigma_2] \)
TRAFFIC: Types

Examples of constraints and relationships we want to encode using types

- A video source is variable-bit-rate with a steady-state rate of \( r \) Mbps and a burst magnitude of no more than \( b \) Mb.
- A steady-state AIMD source with loss rate \( p \), a RTT of \( T \), and an MTU \( M \) will send at a rate of at most \( 1.3 \times M / (\sqrt{p} \times T) \).
- A video client is willing to tolerate up to \( p \% \) steady-state loss rate and a steady-state delay of less than \( T \) seconds.
- A wireless hop drops at least \( p \% \) and at most \( p \% \) packets, plus a single burst of at most \( b \) packets at steady-state.
- A traffic shaper adds less than \( T \) secs of delay, smoothing all transmissions to a steady state rate of \( r \) Mbps.

TRAFFIC: Type relationships

Definition: A type \( t_1 \) is a subtype of \( t_2 \) (denoted \( t_1 <: t_2 \)) if \( t_1 \) is more “constrained” than \( t_2 \)

\[
\begin{align*}
\{t_1 <: t_2\} & \iff \Delta \vdash t_1 <: t_2 \\
\vdash \tau \in \text{Type} & \iff \Delta \vdash \tau <: \tau \\
\vdash \tau \in \text{FlowType} & \iff \Delta \vdash \tau <: \tau \\
\frac{\Delta \vdash \rho <: \rho'}{\Delta \vdash \rho <: \rho'} \\
\frac{\Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\frac{\Delta \vdash \rho <: \rho, \Delta \vdash \rho <: \rho', \Delta \vdash \rho' <: \rho' \prime}{\Delta \vdash \rho <: \rho' \prime} \\
\end{align*}
\]

Subtyping rules for bundles and flows

- Subtyping is transitive
TRAFFIC: Type relationships

- How do we come up with these relationships?
  - Using domain knowledge or specifics related to standards or implementation details
    - TCP window size/rate is bounded by receiver window
    - TCP rate is bounded by TCP equation at steady state
  - By leveraging well-grounded theories and calculi — e.g., queuing, control, scheduling theory, network calculus, ...
    - Delay of M/M/1/K is always less than that of M/M/1
    - Error of PI controller is more than that of PID controller

- Using TRAFFIC for compositional analysis depends on the goodness of these relationships!
  - Garbage-In = Garbage-Out

TRAFFIC: Type constructs

- A system is an arrangement of gadgets

\[ x, y, z \in \text{FlowVar} \]
\[ A, B, C \in \text{LocalFlow} \]
\[ A, B, C \in \text{GlobalFlow} := A \mid x \]
\[ A \parallel B \]
\[ \text{let } x = A \text{ in } B \]

\( A_1 \parallel A_2 \) (parallel)
\( A_1 ; A_2 \) (sequential)
**TRAFFIC: Type inference**

- Type of a parallel arrangement of gadgets

\[
\Gamma, \Delta \vdash A : T \quad \Gamma, \Delta \vdash B : T' \quad \text{where} \quad \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \bullet \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} = \begin{bmatrix} \rho_5 & \rho_6 \\ \sigma_5 & \sigma_6 \end{bmatrix}
\]

- Type of a sequential arrangement of gadgets

\[
\Gamma, \Delta \vdash A : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \quad \Gamma, \Delta \vdash B : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} \quad \Delta \vdash \rho_2 \ll ; \rho_3 \quad \Delta \vdash \sigma_2 \ll ; \sigma_2
\]

\[
\Gamma, \Delta \vdash A ; B : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}
\]

**TRAFFIC: Let bindings**

- Flow variables
  - Allow us to represent and reason about unknown entities in the network
  - Examples:
    - HTTP agent is 1.0 or 1.1 compliant
    - TCP agent is Reno or Tahoe
    - Buffer is DropTail or RED
    - ...

- Useful as a placeholder for
  - Type Checking:
    - Check all possible types of gadgets with which we may interact
  - Type Inference:
    - Infer the type of gadget to "plug in" for things to work out
TRAFFIC: Type inference

\[ A \vdash (B \parallel C) ; D ; (E \parallel F \parallel G) \]
- Fully known network
- Do the pieces ”fit”? Are all requirements satisfied?

\[ A \vdash (x \parallel C) ; y ; (E \parallel F \parallel G) \]
- Partially known network
- Do the known pieces ”fit”?
- What is required of unknown pieces?

*Work Forward:* engineer to meet specs
*...or Backward:* which extant pieces will fit?

TRAFFIC instantiations

- An instantiation of TRAFFIC requires us to define a type system:
  - What are the set of possible types?
  - What sub-typing relationships exist?
  - What type transformation are possible?

- TRAFFIC(Network Calculus):
  - NetCal provides a nice set of possible types
  - NetCal allows derivation of sub-typing rules
  - NetCal enables derivation of type transforms
TRAFFIC over Network Calculus

- **What is Network Calculus?**
  - NetCal introduced by Jean-Yves Le Boudec & Patrick Thiran; building on the seminal works of Parekh & Gallager (circa mid 1990s)
  - NetCal is a collection of results based on Min-Plus algebra, which applies to deterministic queuing systems found in communication networks
  - Allows us to reason about bounds on capacity, demand, utilization, etc... with bounding functions over time intervals

NetCal + TRAFFIC

- **We are not making NetCal more powerful**
  - On the contrary, analysis “by hand” using NetCal would produce more refined/expressive results than will be possible with TRAFFIC over NetCal
  - Recall that trading off expressiveness for scalability is the stated goal of our approach!

- **We are distilling and codifying NetCal so as to use it to mechanically analyze systems in TRAFFIC**
  - Type expressions require a working familiarity with NetCal to be intelligible...
  - But, ultimately NetCal will be hidden from the average network programmer or architect, just as the details of data representation are hidden from programmers...
NetCal: Data flow types

- **Data Flow** \( R(t) \)
  - Bits seen in \([0, t]\)
  - Rate \((dR/dt)\) is a byproduct; need not be defined!

- One may use data flow functions as “bounds” to define classes of TRAFFIC types for data flows (denoted by “\(R\)”):
  - Consider the function \( f(t) = 0.25t + \sqrt{t} \)
    - \([0]_R\) is a clear lower bound \(\Rightarrow f(t): [0]_R\)
    - \([0.25]_R\) is another lower bound \(\Rightarrow f(t): [0.25]_R\)
    - \([0.75t + 0.5]_R\) is an upper bound \(\Rightarrow f(t): [0.75t + 0.5]_R\)
    - Using intersections of types \(\Rightarrow f(t): [0.25]_R \cap [0.75t + 0.5]_R\)

NetCal: Arrival curve types

- **Arrival (Process) Curves**
  - Bits seen in an interval (a window of time) \(t\)

- One may use arrival curves as “bounds” to define classes of TRAFFIC arrival processes (denoted by “\(a\)”):
  - One can show that for any arrival curve \(f(t)\), \([f]_a < [f]_N\)
  - One can show that for any data flow function \(f(t)\) and arrival curve \(g(t)\), \([f]_N \leq [g]_a\) iff \(f(t) - f(s) \leq g(t - s)\), for all intervals \(s\) and \(t\).
  - Thus for the leaky bucket function \(a(t) = rt + b\) with steady state rate \(r\) and burst size \(b\), we get \([rt + b]_N \leq [rt + b]_a\)
NetCal: Service curve types

- **Service (Process) Curves**
  - How an incoming data flow $R(t)$ is serviced (e.g., delayed and rate limited) to produce an outgoing data flow $R^*(t)$

$$R^*(t) \geq \min_{s \leq t} (R(s) + \beta(t-s))$$

- Outgoing data flow $R^*(t)$ is the convolution using min-plus algebra of incoming flow $R(t)$ and service curve $\beta(t)$

NetCal: On convolution of flows

- **Example**

Standard convolution:

$$(f * g)(t) = \int f(t-u)g(u)du$$

Min-plus convolution:

$$f \circledast g(t) = \min \{f(t) + g(u)\}$$
NetCal: Shaper curve types

- Shaper Curves
  - How an incoming data flow $R(t)$ is shaped through the use of a (in)finite buffer to produce an outgoing data flow $R^*(t)$

\[
R(t): \left[ \begin{array}{c} \mathbf{g} \\ \mathbf{g} \end{array} \right]_R \quad \rightarrow \quad \left\{ \begin{array}{c} \mathbf{f} \\ \mathbf{f} \end{array} \right\}_R \quad \rightarrow \quad R^*(t): \left[ \begin{array}{c} \mathbf{f} \\ \mathbf{g} \end{array} \right]_R
\]

- Outgoing data flow $R^*(t)$ is the convolution using min-plus algebra of incoming flow $g(t)$ and shaper curve $f(t)$

NetCal: Lossy shaper types

- Loss Curves
  - How an incoming data flow $R(t)$ is shaped through the use of a finite buffer to produce a loss flow $L(t)$

\[
R(t): \left[ \begin{array}{c} \mathbf{f} \\ \mathbf{f} \end{array} \right]_R \quad \rightarrow \quad \left\{ \begin{array}{c} \mathbf{x} \\ \mathbf{x} \end{array} \right\}_R \quad \rightarrow \quad L(t): \left[ \begin{array}{c} \mathbf{f}(t)-g(t) \end{array} \right]_L
\]

- One may use loss "bounds" to define classes of TRAFFIC loss flows (denoted by "$L$") or loss rates (denoted by "$l$")
  - A flow with a loss rate of no more than 5 bps would have the type $\left[ 0.05 \right]_L$ whereas one with 1% loss would be $\left[ 0.01 \right]_L$. 

03/13/2006 TRAFFIC @ NRG 36

03/13/2006 TRAFFIC @ NRG 37
NetCal: Additional inferences

<table>
<thead>
<tr>
<th>Incoming</th>
<th>Outgoing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F' R^2$</td>
<td>$F \otimes \beta R^2$</td>
</tr>
<tr>
<td>$F R^2$</td>
<td>$G \sigma \cap F \otimes G R^2$</td>
</tr>
<tr>
<td>$F R^2$</td>
<td>$Q_L \cap (F - Q) R^2$</td>
</tr>
<tr>
<td>$F' R^2$</td>
<td>$Q_L \cap (F - Q) \otimes G R^2$</td>
</tr>
</tbody>
</table>

TRAFFIC: Putting it together

Consider a simple video delivery network

```
video1 || video2; shaper; delivery; (clientA || clientB)
```

Where

- **video nodes**: Each outgoing socket is $\| t - 5 \| R^2 \cap \| t + 5 \|_\alpha$
- **shaper node**: Incoming socket is $\| 2t - 10 \| R^2 \cap \| 2t + 10 \|_\alpha$
  Service curve is $\| 2t - 10 \|_\beta$ shaper is $\| 2t \|_\sigma$
- **delivery network**: Loss rate is $\| 0.15t + 1 \| \cap \| 0.05t \|
- **client nodes**: Each incoming socket is $\| 1.2t - 16 \|_\alpha \cup \| 0.7t - 4.5 \|_\alpha$

**Will it work?**
TRAFFIC: Putting it together

Note: Backward path is unconstrained

TRAFFIC: Code

```plaintext
// System specification as interconnections of gadgets
specification Spec = begin
  (Video || Video) ; Connector1 ; Shaper ; Delivery ; Connector2 ; (Client || Client)
end

// Defining components along with their types ([fw-in , fw-out ; bw-out , bw-in ])
localflows L: D = begin
  let Video      = ["nil"      , "t1"        ; (0,0) , (0,0)]
  let Connector1 = ["t2"  , "t3"        ; (0,0)* (0,0) , (0,0)]
  let Shaper     = ["t4"       , "t5"        ; (0,0), (0,0)]
  let Delivery   = ["top"      ; "t6"        ; (0,0), (0,0)]
  let Connector2 = ["t7"       , ("t8" ) ; (0,0), ((0,0)*(0,0))]
  let Client     = ["t9"       , "nil"       ; (0,0), (0,0) ]
end

// Socket types and subtype relationships obtained using NetCal engine or library:
relations D = begin
  BwSocketType ::= Range // backward types don’t play a role in this example
  FwSocketType ::= "nil" | "top" | "t1" | "t2" | "t3" | "t4" | "t5" | "t6" | "t7" | "t8" | "t9"
  where // below are the subtype relationships:
  "t1" < "t2" ;
  "t3" < "t4" ;
  "t5" < "top" ; // "top" is the supertype of all types.
  "t6" < "t7" ;
  "t8" < "t9"
end

// Check network configuration:
typing T1 = check Spec : D using L
```
TRAFFIC: In action

☐ Type checker uses NetCal relationships and TRAFFIC rules to check if sub-typing constraints are satisfied
  - System checks if all is well
  - System does not check otherwise

☐ Let’s try it – thank you Likai & Yarom!

TRAFFIC: In action

☐ What if we have multiple choices – e.g.,

\[
\begin{align*}
\text{Shaper is} & \quad \left\{ \begin{array}{l}
\left[ [2t-10]_a \cap [2t+10]_a \right]_{\tau_{c,b}} \\
\text{or} \\
\left[ [2t-5]_a \cap [2t+10]_a \right]_{\tau_{c,b}} \cap [2t-10]_b_{\tau_{c,d}}
\end{array} \right. \\
\end{align*}
\]

☐ Let’s try it
TRAFFIC: Another example

```
specification LossyVideoDelivery = begin
  let x = LossyNet in (spec VideoDelivery)
end
```

```
specification ReliableVideoDelivery = begin
  let x = ReliableNet in (spec VideoDelivery)
end
```

```
specification VideoDelivery = begin
  Video ; Compress ; x ; Decompress; Client
end
```

```
localflows L : D = begin
  let Video       = ["nil", "t1" ; (0,0), (0,0)]
  let Compress    = ["top", "t2" ; (0,0), (0,0)]
  let LossyNet = ["top", "t3a" ; (0,0), (0,0)]
  let ReliableNet = ["top", "t3b" ; (0,0), (0,0)]
  let Decompress  = ["t4", "t5" ; (0,0), (0,0)]
  let Client      = ["t6", "nil" ; (0,0), (0,0)]
end
```

```
relations D = begin
  BwSocketType ::= Range  // backward types don’t play a role in this example
  FwSocketType ::= "nil" | "top" | "t1" | "t2" | "t3a" | "t3b" | "t4" | "t5" | "t6"
  where // below are the subtype relationships:
  "t1" <: "top" ; "t2" <: "top" ; "t3a" <: "top" ; "t3b" <: "top" ;
  "t4" <: "top" ; "t5" <: "top" ; "t6" <: "top" ; // top is supertype
  "t3a" <: "t2" ; // compressor output data flow is a subtype of its input
  "t3b" <: "t2" ; // network is lossy
  "t2" <: "t3b" ; // network is reliable
  "t2" <: "t4" ; // decompressor input accepts any output of a compressor
  "t5" <: "t1" ; // data flow out of compressor cannot be more than original
  "t6" <: "t1" ; // client input data flow can take the original video output
  "t6" <: "t1" // client input data cannot exceed original data
end
typing T1 = check LossyVideoDelivery : D using L
typing T2 = check ReliableVideoDelivery : D using L
```

TRAFFIC: What’s next?

- Write TRAFFIC specs for a host of network gadgets and compositions
- Develop a NetCal oracle to find out if type A is a subtype of type B, using min-plus algebra
- Leverage other theories, e.g., scheduling theory, to develop other oracles
- Add TRAFFIC checking/inference to a network programming environment, e.g., snBench
- Make it transparent to users/programmers, e.g., develop GUI, couple with ns, ...
- Allow for more expressive constructs in TRAFFIC, e.g., allowing for type variables
Type-disciplined, Scalable, Practical Composition of Networked Services

Azer Bestavros

Joint work with
Adam Bradley, Yarom Gabay, Likai Liu, Assaf Kfoury, and Ibrahim Matta
(and maybe some of you 😊)

NRG, BU CS Department
March 13, 2006