Taking the Road Less Traveled in CS Research at BU
Bringing the Prowess of Types and Typings to Bear on Network Systems Design

Azer Bestavros & Assaf Kfoury

Computer Science Department
Boston University

http://www.cs.bu.edu/groups/ibench
Imagine a *networked* world of ...  

... sensors, actuators, processors and storage, which is part of a shared physical infrastructure

*Not hard to imagine!*
A common space equipped with video sensors (VS) for ubiquitous recognition and tracking of activities therein, \textit{circa 2002}.

**Infrastructure:**
- Range of VS Elements
- Programmable VS Network
- Backend compute engines
- Backend TByte storage
- Mobile/wireless query units
- Research Engineer
Sensoria...

Assistive Environments
  e.g. for home/hospice/elder care/...

Safety Monitoring
  e.g. in factories/daycare/hospitals/garages/subway...

Intelligent Spaces
  e.g. for classrooms/meeting rooms/theaters/farms...

Secure Facilities and Homeland Security Uses
  e.g. at airports/embassies/prisons/...

People Flow/Activity Studies
  e.g. at retail stores/museums/...
The Sensorium is the computer...

Design/implement the programming and run-time infrastructure necessary for developers to specify and deploy truly distributed applications over a shared heterogeneous network of Sensing Elements (SEs) and of Computing Elements (CEs)

What sensors could I use and what functionality do I get from them?
**snBench: Goals**

- “Write Once, Run Anywhere”
- “Program the network *not* the nodes!”

  - Start with building blocks – “typed gadgets”
    - Sensors (cameras, motion sensors, 802.11, …)
    - Stock algs (edge detect, face count, FFT, …)
  
  - Glue together with high-level language
    - Conditionals, loops, functions
    - Use annotations as constraints
  
  - Pretend the network isn’t there
    - Design and implement as a “Single System Image”
    - Reason “compositionally” about the entire system
snBench: We built it (Demo)

“When a wireless intruder is detected at a base station, then take a snapshot from all the cameras in the secure space being monitored and add a record to the intrusion log database.”
How About Analysis?

- It’s good to know that a network agent (e.g., HTTP proxy, ...) doesn’t deadlock on its own...

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

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

Compositional Analysis

- **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 analyses: Derive properties of Z by composing the analysis of X and the analysis of Y

- **Does analysis scale?**
  - Representation size? is it manageable?
  - Representation legibility? is a PhD required?
  - Computational tractability? is it feasible?
Software Engineering Analogy...

- We are able to reason about (and hence scale) compositions of large software artifacts by hiding internals and only thinking about interfaces
  
  ➔ 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
Sounding 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.
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
  - Feasibility
  - Scalability

  - Expressiveness: *what are you able to prove?*
  - Feasibility: *can you can prove it?*
  - Scalability: *for what size problem?*
TRAFFIC

Typed Representation and Analysis of Flows For Interoperability Checks
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

☐ QoS 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.

☐ Security types:
  - A data source/sink produces/consumes an authenticated (or encrypted) data stream

☐ Coding types:
  - An erasure encoder accepts $n$ data streams and produces $n+e$ streams

☐ Real-Time types:
  - A multiplexer accepts $n$ streams to produce a stream in which for stream $i$ there are $c_i$ cells in any $t_i$ time window
TRAFFIC: Instantiations

☐ TRAFFIC instance requires definitions of
  ■ 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
NetCal: Data flow types

- **Data Flow** $R(t)$
  - # of 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 $\Rightarrow f(t): \|0\|_R$
    - $\|0.25t\|_R$ is another lower bound $\Rightarrow f(t): \|0.25t\|_R$
    - $\|0.75t + 0.5\|_R$ is an upper bound $\Rightarrow f(t): \|0.75t + 0.5\|_R$
    - Using intersections of types $\Rightarrow f(t): \|0.25t\|_R \cap \|0.75t + 0.5\|_R$
let const stream = get("videostream","cam1") in
display(either(max_loss(0.2,(max_delay(10,stream))),
          min_rate(1.2,(max_delay(20,stream)))))

\[
\begin{align*}
\text{Cam1} & : \alpha(t-5) \land \alpha(t+5) = \alpha(t-5) \\
\text{Cam2} & : \alpha(t-10) \land \alpha(t+10) = \alpha(t-10) \\
\text{Shaper} & : \alpha(0.15t+1) \land \alpha(0.05t) = \alpha(0.15t+1) \\
\text{Network} & : \alpha(\|t-24\|_{\alpha} \cup \|0.8t-8\|_{\alpha}) = \alpha(\|t-24\|_{\alpha} \cup \|0.8t-8\|_{\alpha}) \\
\text{ClientA} & : \alpha(\|1.2t-24\|_{\alpha} \cup \|0.8t-8\|_{\alpha}) = \alpha(\|1.2t-24\|_{\alpha} \cup \|0.8t-8\|_{\alpha}) \\
\text{ClientB} & : \alpha(\|1.2t-16\|_{\alpha} \cup \|0.8t-8\|_{\alpha}) = \alpha(\|1.2t-16\|_{\alpha} \cup \|0.8t-8\|_{\alpha})
\end{align*}
\]
Beyond NetCal

- Different techniques are better at checking different types of properties
  - Control theory: Convergence, stability, dynamics, ...
  - Network calculus: Max/min delays, b/w, loss rates, ...
  - Queuing theory: Average delay, utilization,
  - Real-time theory: Schedulability/timing analysis, QoS, ...
  - State-space analysis: Deadlocks, synchronization, ...
  - Game theory: Price of anarchy, mistreatment, ...
  - ... put your pet theory here

- Need a seamless way to leverage all such theories and techniques
Need to “compose” theories
Our Hour-Glass Approach
Model Once Verify Everywhere

Applications
Access Control; SDN-Enabled Moving Target Defense; Embedded/CPS Systems; Cloud SLA Verification
Putting it Together: Verificare

Software Defined Networks enable CPS applications to monitor and control net flows

Need to verify safety/security of whole system and not just network layer correctness

The Verificare Approach

“Model once verify as needed”

1. VML Model
2. Domain Abstraction
3. Verification
4. Counterexample
5. Feedback

Off-the-Shelf Verification Engines

Explicit-State Model Checker
Symbolic Model Checker
SMT Solver

Domain Abstraction Library
Reusable Requirement Library

Accessible modeling language
Multiple domain abstractions
Reusable safety properties
Multiple backend calculi

FRENETIC, NICE, ndb, Procera, Flog, Mininet

CPS System Designer
Network Architect

CPS Model
SLA
SDN Models
SDN Implementation
Network Infrastructure
CPS Implementation
Host

Compile Verify
Compile Verify
Compile Verify
What Are Network Typings and What Are They Good For?
a subway network, **uniform** criteria used for selecting optimal routings

wanted: an optimal routing from point $X$ to point $Y$
Prelude: a small example

A subway network, mixed criteria used for selecting optimal routings.

Orange areas / blue areas: optimal routings determined by theory A / theory B.
Prelude: a small corner of the Internet

another use for network types/typings
**Prelude**: a small corner of the Internet

another use for network types/typings

what if only three clusters designed and available for analysis so far?

is it possible to start an analysis without waiting for the entire network to be assembled?
Prelude: a small corner of the Internet

another use for network types/typings

what if only two more assembled and available for analysis after a week?
Prelude: a small corner of the Internet

another use for network types/typings

what if only two more assembled and available for analysis after a week?

is it possible to combine analyses of adjacent clusters into a single analysis?
Prelude: a small corner of the Internet

another use for network types/typings

what if a cluster in a combined analysis breaks down or is re-configured two weeks later?

is it still possible to use a combined analysis if one of its clusters is re-configured?
again, our questions:

is it possible to start an analysis without waiting for the entire network to be assembled?
Yes ...

is it possible to combine analyses of adjacent clusters into a single analysis?
Yes ...

is it still possible to use a combined analysis if one of its clusters is re-configured?
Yes ...
Prelude: what we want to achieve

An integrated environment for network modeling and analysis which is:

1. **modular**, *i.e.*, distributed in space,
2. **incremental**, *i.e.*, distributed in time,
3. **order-oblivious**,  
   *i.e.*, clusters can be assembled and analyzed in *any* order,
4. proceeding **inside-out**, *i.e.:
   - by **composing** constraints, 
     possibly from different theories for different clusters in the network, 
     rather than **outside-in** (followed by **inside-out**), *i.e.:
   - by **de-composing** constraints, 
     from a single theory for the entire network, and 
     then **re-composing** results from the de-composed constraints.

The latter approach: an instance of **divide-and-conquer**,  
our proposed approach: **conquer-with-no-need-to-divide**.
Prelude: what we want to achieve – in short

An environment supporting:

- what we call **Compositional Analysis**, in particular:
  - **not requiring** knowledge of the entire network, which may still be in the process of assembly/reconfiguration,
  - **not requiring** all constraints to be from the same optimization theory for all clusters/components, but as long as they share a common formalism to express invariant properties across interfaces.
  - **not requiring** . . .

- as opposed to **Whole-Network Analysis**:
  - **requiring** knowledge of the entire network,
  - **requiring** an appropriate de-composition of the constraints that allows for the re-composition of their results.
  - **requiring** . . .
Prelude: what we want to achieve – in short

An environment supporting:

- what we call **Compositional Analysis**, in particular:
  - **not requiring** knowledge of the entire network, which may still be in the process of assembly/reconfiguration,
  - **not requiring** all constraints to be from the same optimization theory for all clusters/components, but as long as they share a common formalism to express invariant properties across interfaces.
  - **not requiring**...

- as opposed to **Whole-Network Analysis**:
  - **requiring** knowledge of the entire network,
  - **requiring** an appropriate de-composition of the constraints that allows for the re-composition of their results.
  - **requiring**...

How we achieve it:

- by leveraging the power of **network types** and **network typings**
Prelude: benefits of types and typings – so far

Two benefits mentioned so far, which we anticipate from a theory of network types and network typings:

**benefit 1**: the ability to deal with *mixed/heterogeneous* systems\(^1\) of constraints, which regulate different parts of the network, calling for different optimization theories, and the ability to compose their results,

\(^1\)We avoid saying “hybrid” system, a term already used to mean something else. What we mean by “mixed” or “heterogenous” does not exclude hybrid components (exhibiting both continuous- and discrete-time behaviors) in the network.
Two benefits mentioned so far, which we anticipate from a theory of network types and network typings:

**benefit 1**: the ability to deal with mixed/heterogeneous systems\(^1\) of constraints, which regulate different parts of the network, calling for different optimization theories, and the ability to compose their results,

**benefit 2**: the ability to deal with under-specified and changing network topologies.

---

\(^1\)We avoid saying “hybrid” system, a term already used to mean something else. What we mean by “mixed” or “heterogenous” does not exclude hybrid components (exhibiting both continuous- and discrete-time behaviors) in the network.
Two benefits mentioned so far, which we anticipate from a theory of network types and network typings:

**benefit 1**: the ability to deal with mixed/heterogeneous systems\(^1\) of constraints, which regulate different parts of the network, calling for different optimization theories, and the ability to compose their results,

**benefit 2**: the ability to deal with under-specified and changing network topologies.

We come back later to discuss other benefits, beyond the preceding two, after making several notions more concrete with a few examples – and an Interlude.

---

\(^1\) We avoid saying “hybrid” system, a term already used to mean something else. What we mean by “mixed” or “heterogenous” does not exclude hybrid components (exhibiting both continuous- and discrete-time behaviors) in the network.
Interlude: but is it a fair comparison?

Compositional Analysis (CA) versus any form of Whole-Network Analysis (WA):

- **CA** deals with:
  - changing and growing networks (e.g., with failure-prone components, re-configured components, etc.),
  - possibly different optimization theories for different components.

- **WA** deals with:
  - fully-known/fully-assembled/stable networks,
  - one optimization theory throughout a network.

- So, **CA** and **WA** are adapted to different situations.

- Perhaps **CA** and **WA** are incomparable approaches after all . . . ?
Interlude: but is it a fair comparison?

Compositional Analysis (CA) versus any form of Whole-Network Analysis (WA):

- **CA** deals with:
  - changing and growing networks (e.g., with failure-prone components, re-configured components, etc.),
  - possibly different optimization theories for different components.

- **WA** deals with:
  - fully-known/fully-assembled/stable networks,
  - one optimization theory throughout a network.

So, **CA and WA** are adapted to different situations.

- Perhaps **CA and WA** are incomparable approaches after all . . . ?

But not so fast, **here is an extra:**

- Even for fully-known/fully-assembled/stable networks, and even for a single optimization theory used throughout, **CA will often have advantages over WA . . .**
**Prelude**: a small corner of the Internet, once more
Prelude: a small corner of the Internet, once more

- assume we use a single optimization theory throughout the network – just as it is for any WA.

- assume the network is fully assembled, failproof, stable, unchanging – just as it is for any WA.

- CA may still have advantages over WA.
• assume we use a single optimization theory throughout the network – just as it is for any WA.

• assume the network is fully assembled, failproof, stable, unchanging – just as it is for any WA.

• CA may still have advantages over WA.

the grid on the network represents a possible decomposition, possibly dictated by the network structure, but not necessarily now – (more later) (other benefits)
In the next few slides we present several examples of the **forms** in which network types and network typings can be formulated.

There is a large variety, depending on the application and the context – sometimes resembling types and typings in **strongly-typed programming languages**, but not always.
Varieties of network types and typings

- In the next few slides we present several examples of the **forms** in which network types and network typings can be formulated.

- There is a large variety, depending on the application and the context – sometimes resembling types and typings in **strongly-typed programming languages**, but not always.

- We distinguish between network **types** and network **typings**:
  
  - a **type** is a:
    - **range** of admissible values, and/or
    - **abstraction** separating admissible from non-admissible values, for a **single** external link of a cluster/module,
  
  - a **typing** is a **relationship/dependence**, **functional** or **logical**, between **all** the types assigned to **all** the external links of the same cluster/module.
Varieties of network types and typings

Simple example with booleans:

- typing of cluster/module $A$ is $T \subseteq \mathbb{B}^4$, 

\[
\begin{align*}
\text{A} & \quad \text{a}_1 \quad \text{a}_2 \\
& \quad \text{a}_3 \quad \text{a}_4
\end{align*}
\]
Varieties of network types and typings

Simple example with booleans:

- typing of cluster/module $A$ is $T \subseteq \mathbb{B}^4$,
- “plugging” $A : T$ in $A' : T'$ is safe:
  - if $T = T'$,
  - or if $A$ is consumer and $A'$ producer:
    \[ T \supseteq T', \]
  - or if $A$ is producer and $A'$ consumer:
    \[ T \subseteq T'. \]
Another simple example with booleans:

- $A$ is both **consumer** and **producer**, 
- typing of $A$ is $T \in (X \rightarrow Y)$, *i.e.*, a function from $X$ to $Y$ with $X, Y \subseteq \mathbb{B}^2$, 

![Diagram of network typing]

Varieties of network types and typings
Varieties of network types and typings

Another simple example with booleans:

- **A** is both **consumer** and **producer**,
- typing of **A** is $T \in (X \rightarrow Y)$, i.e., a function from $X$ to $Y$ with $X, Y \subseteq \mathbb{B}^2$,
- inserting $A : T$ in a “hole” of $A' : T'$, with matching input/output connections, where $T \in (X \rightarrow Y)$ and $T' \in (X' \rightarrow Y')$,
  is **safe** if $T$ is a subtyping of $T'$:

  $$T <: T' \text{ iff } X' \subseteq X \text{ and } Y \subseteq Y',$$

  *i.e.*, contravariant **input**, covariant **output**.
Varieties of network types and typings

An example of typings over the infinite domain $\mathbb{N}$:

- typing of $A$ is $T \in (X \to Y)$ with $X, Y \subseteq \mathbb{N}^2$, 
An example of typings over the infinite domain $\mathbb{N}$:

- Typing of $A$ is $T \in (X \rightarrow Y)$ with $X, Y \subseteq \mathbb{N}^2$,
- Inserting $A : T$ in a “hole” of $A' : T'$, where $T \in (X \rightarrow Y)$ and $T' \in (X' \rightarrow Y')$, is safe if $T$ is a subtyping of $T'$:

$$T <: T' \text{ iff } X' \subseteq X \text{ and } Y \subseteq Y'.$$
Another example of typings over $\mathbb{N}$:

- typing of $A$ is $T = (T_{\text{in}}, T_{\text{out}})$ with $T_{\text{in}}, T_{\text{out}} \in \mathcal{I}(\mathbb{N}) \times \mathcal{I}(\mathbb{N})$,
Another example of typings over $\mathbb{N}$:

- typing of $A$ is $T = (T_{in}, T_{out})$ with $T_{in}, T_{out} \in \mathcal{I}(\mathbb{N}) \times \mathcal{I}(\mathbb{N})$,

- inserting $A : T$ in a “hole” of $A' : T'$, where $T = (T_{in}, T_{out})$ and $T' = (T'_{in}, T'_{out})$,

is **safe** if $T$ is a subtyping of $T'$:

$$T <: T' \iff T_{in} \subseteq T'_{in} \text{ and } T_{out} \subseteq T'_{out},$$
Another example of typings over $\mathbb{N}$:

- Typing of $A$ is $T = (T_{in}, T_{out})$ with $T_{in}, T_{out} \in \mathcal{I}(\mathbb{N}) \times \mathcal{I}(\mathbb{N})$,

- Inserting $A : T$ in a “hole” of $A' : T'$, where $T = (T_{in}, T_{out})$ and $T' = (T'_{in}, T'_{out})$,

  is **safe** if $T$ is a subtyping of $T'$:

  $T <: T'$ iff $T'_{in} \subseteq T_{in}$ and $T'_{out} \subseteq T_{out}$,

- Inferring $T$ from the concrete behavior of $A$:
Another example – unusual, perhaps unexpected – of typings over the domain $\mathbb{Q}$ of rationals (further elaborated later in the presentation):

- typing of $A$ is a partial function from the powerset of $\{i_1, i_2, o_1, o_2\}$ to $\mathcal{I}(\mathbb{Q})$:

$$T \in \left( \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \to \mathcal{I}(\mathbb{Q}) \right),$$
Varieties of network types and typings

Another example – unusual, perhaps unexpected – of typings over the domain \( \mathbb{Q} \) of rationals (further elaborated later in the presentation):

- Typing of \( A \) is a partial function from the powerset of \( \{i_1, i_2, o_1, o_2\} \) to \( \mathcal{I}(\mathbb{Q}) \):

\[
T \in \left( \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \rightarrow \mathcal{I}(\mathbb{Q}) \right),
\]

- Inserting \( A : T \) in a “hole” of \( A' : T' \) is certainly safe if \( T = T' \),

- \( T \) is a subtyping of \( T' \), \( T <: T' \), is a little more complicated . . . .
Recapping the two benefits mentioned in the Prelude:

**benefit 1**: the ability to deal with *mixed/heterogeneous* systems of constraints, which regulate different parts of the network, calling for different optimization theories, and the ability to compose their results,

**benefit 2**: the ability to deal with *under-specified* and *changing* network topologies.

But there is more that we can get from types and typings . . .
Two alternatives, (a) and (b), of dealing with types and typings:

(a) types and typings can be inferred during/after the process of software construction and used to confirm its correctness, i.e.,

first: software parts are written/ designed,

second: types and typings are inferred during/after.
Two alternatives, \( (a) \) and \( (b) \), of dealing with types and typings:

\( (a) \) types and typings can be inferred during/after the process of software construction and used to confirm its correctness, \( i.e., \)

\begin{itemize}
  \item \textbf{first}: software parts are written/designed,
  \item \textbf{second}: types and typings are inferred during/after,
\end{itemize}

For an altogether different perspective, suggested by more recent experiences with strongly-typed programming languages:

\( (b) \) types and typings can be written/given prior to the process of software construction and used to guide it, \( i.e., \)

\begin{itemize}
  \item \textbf{first}: types and typings are specified,
  \item \textbf{second}: software parts are written/designed during/after.
\end{itemize}
Two alternatives, (a) and (b), of dealing with types and typings:

(a) types and typings can be inferred during/after the process of software construction and used to confirm its correctness, i.e.,

first: software parts are writtenDesigned ,
second: types and typings are inferred during/after ,

For an altogether different perspective, suggested by more recent experiences with strongly-typed programming languages:

(b) types and typings can be written/given prior to the process of software construction and used to guide it, i.e.,

first: types and typings are specified ,
second: software parts are writtenDesigned during/after .

Alternative (a) is illustrated by benefit 1 and benefit 2 and the examples in this presentation so far. Alternative (b) suggests a third benefit . . .
Typings to the rescue: a third benefit

**benefit 3**: the ability to model/design a network, or parts of it, from a pre-given, or simultaneously given, *certificate/contract* that will formally guarantees satisfaction of safety requirements (*formulated as types and typings*).\(^2\)

\(^2\)The *a-priori* specification of types and typings minimizes the effort for what has been called *a-posteriori verification*. Joseph Sifakis, *The Quest for Correctness – Beyond a-posteriori Verification*, Spin 09, June 2009.
**benefit 3**: the ability to model/design a network, or parts of it, from a pre-given, or simultaneously given, *certificate/contract* that will formally guarantees satisfaction of safety requirements (formulated as *types* and *typings*).²

- In the next few slides we give a small example, illustrating the two alternatives and some of their differences:
  - build the module *first*, infer the typing *second*, for benefits 1 and 2,
  - specify the typing *first*, build the module *second*, for benefit 3.

²The *a-priori* specification of types and typings minimizes the effort for what has been called *a-posteriori verification*. Joseph Sifakis, *The Quest for Correctness – Beyond a-posteriori Verification*, Spin 09, June 2009.
**Typings** to the rescue: recapping of how we proceed for benefits 1 and 2

*First*, we build a network module, or someone gives it to us to check . . .

(Missing capacities are “very large”.)
**Typings** to the rescue: recapping of how we proceed for benefits 1 and 2

**First**, we build a network module, or someone gives it to us to check . . .

**Second**, we compute a principal typing:

\[ T : \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \rightarrow \mathcal{I}(\mathbb{Q}), \]

with \( T \) making the following mappings:

\[
\begin{align*}
\{i_1\} &\mapsto [0, 15] & \{i_2\} &\mapsto [0, 25] \\
\{o_1\} &\mapsto [-15, 0] & \{o_2\} &\mapsto [-25, 0] \\
\{i_1, i_2\} &\mapsto [0, 30] \\
\{i_1, o_1\} &\mapsto [-10, 12] \\
\{i_1, o_2\} &\mapsto [-23, 15]
\end{align*}
\]

(Missing capacities are “very large”.)
Typings to the rescue: recapping of how we proceed for benefits 1 and 2

**First**, we build a network module, or someone gives it to us to check . . .

![Network Module Diagram]

**Second**, we compute a principal typing:

\[ T : \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \rightarrow \mathcal{I}(\mathbb{Q}), \]

with \( T \) making the following mappings:

- \( \{i_1\} \mapsto [0, 15] \quad \{i_2\} \mapsto [0, 25] \)
- \( \{o_1\} \mapsto [-15, 0] \quad \{o_2\} \mapsto [-25, 0] \)
- \( \{i_1, i_2\} \mapsto [0, 30] \)
- \( \{i_1, o_1\} \mapsto [-10, 12] \)
- \( \{i_1, o_2\} \mapsto [-23, 15] \)

(Missing capacities are “very large”.)

**Third**, we now know the concrete input-output behavior of the module (as a processor of network flows), we can confirm whether or not an expected behavior is met, and we can decide whether or not it can be safely inserted in network “holes” and in which ones . . .
Typings to the rescue: how we proceed for benefit 3

**First**, we are given a typing:

\[ T : \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \to I(\mathbb{Q}), \]

with \( T \) making the following mappings:

- \( \{i_1\} \mapsto [0, 15] \quad \{i_2\} \mapsto [0, 25] \)
- \( \{o_1\} \mapsto [-15, 0] \quad \{o_2\} \mapsto [-25, 0] \)
- \( \{i_1, i_2\} \mapsto [0, 30] \)
- \( \{i_1, o_1\} \mapsto [-10, 12] \)
- \( \{i_1, o_2\} \mapsto [-23, 15] \)

This is a **contract** to be satisfied, and if we succeed, it will be a **certificate** for the specified input-output behavior.
**Second**, we build a module whose principal typing is the specified typing, but which one? There are infinitely many such modules . . .

**First**, we are given a typing:

\[ T : \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \to \mathcal{I}(\mathbb{Q}), \]

with \( T \) making the following mappings:

\[
\begin{align*}
\{i_1\} & \mapsto [0, 15] & \{i_2\} & \mapsto [0, 25] \\
\{o_1\} & \mapsto [-15, 0] & \{o_2\} & \mapsto [-25, 0] \\
\{i_1, i_2\} & \mapsto [0, 30] \\
\{i_1, o_1\} & \mapsto [-10, 12] \\
\{i_1, o_2\} & \mapsto [-23, 15]
\end{align*}
\]

This is a **contract** to be satisfied, and if we succeed, it will be a **certificate** for the specified input-output behavior.
Typings to the rescue: how we proceed for benefit 3

**Second**, we build a module whose principal typing is the specified typing, but which one? There are infinitely many such modules . . .

A “good” implementation of $T$ is:

First, we are given a typing:

$$T : \mathcal{P} \{i_1, i_2, o_1, o_2\} \rightarrow \mathcal{I}(\mathbb{Q}),$$

with $T$ making the following mappings:

- $\{i_1\} \mapsto [0, 15]$, $\{i_2\} \mapsto [0, 25]$
- $\{o_1\} \mapsto [-15, 0]$, $\{o_2\} \mapsto [-25, 0]$
- $\{i_1, i_2\} \mapsto [0, 30]$
- $\{i_1, o_1\} \mapsto [-10, 12]$
- $\{i_1, o_2\} \mapsto [-23, 15]$

This is a **contract** to be satisfied, and if we succeed, it will be a **certificate** for the specified input-output behavior.
Typings to the rescue: how we proceed for benefit 3

**Second**, we build a module whose principal typing is the specified typing, but which one? There are infinitely many such modules . . .

A “good” implementation of $T$ is:

First, we are given a typing:

$$T : \mathcal{P}(\{i_1, i_2, o_1, o_2\}) \rightarrow \mathcal{I}(\mathbb{Q}),$$

with $T$ making the following mappings:

- $\{i_1\} \mapsto [0, 15]$, $\{i_2\} \mapsto [0, 25]$
- $\{o_1\} \mapsto [−15, 0]$, $\{o_2\} \mapsto [−25, 0]$
- $\{i_1, i_2\} \mapsto [0, 30]$
- $\{i_1, o_1\} \mapsto [−10, 12]$
- $\{i_1, o_2\} \mapsto [−23, 15]$

This is a contract to be satisfied, and if we succeed, it will be a certificate for the specified input-output behavior.

**Third**, we now know the specified contract/typing is “inhabited” since we are able to implement a module satisfying it. We can insert a copy of the module in all designated network “holes” with a matching typing.
Thank you!