Guiding Analogy

- Type checker for functional programming language yields well-typedness judgements for programs
  - Very special kind of theorem prover

- Theorem prover such as Isabelle/Isar/HOL (www.cl.cam.ac.uk/Research/HVG/Isabelle) yields proofs for formulae
  - May be written in strongly typed functional programming language such as Ocaml (www.ocaml.org)

- If type system allows dependent types, correspondence is exact (Martin-Löf isomorphism between proofs and programs)
  - Basis of Coq and Nuprl theorem provers
Introducing Application to Service Networks

- Will describe service manager by declarations in imaginary functional programming language
  - Similar to ML, but includes dependent types for convenience
  - type $\alpha$ list $n$ is a list of $n$ elements of type $\alpha$

- Service manager uses service providers to construct expressions using services to yield other services

- Service manager proves well-typedness of expressions
Services and Service Types

- **type service**
  - an abstract type

- **type servicelogicformula =**
  - Opaque of service -> bool
  - And of servicelogicformula list

  - Basic formulae such as $\varphi(x)$ are in more complicated “service logic”
  - Actually have structure, but to service manager are atomic: therefore “opaque”
  - Service is of a type iff it satisfies the corresponding formula $\varphi_1(x) \land \varphi_2(x) \land \cdots \varphi_n(x)$
Service Providers

type serviceprovider n =
(premises: servicelogicformula list n) *
(conclusion: servicelogicformula) *
(useconstraint: service list -> bool) *
(provide: ( (resources: service list n)
  * (currentusers: service list) -> service ) ) *
derivation: servicelogicderivation

An n-ary service provider with premises $\varphi_1, \ldots, \varphi_n$, conclusion $\varphi$, use constraint $u$ and provision function $\text{provide}$ carries a derivation in the service logic of the formula:

$$
\frac{\varphi_1(x_1) \land \cdots \land \varphi_n(x_n) \land u(Y)}{\varphi(\text{provide}(x_1, \ldots, x_n, Y))}
$$

where the $x_i$’s are services and $Y$ is a list of services of arbitrary length.
Semantics of Service Provision

- The $x_i$’s are the *resources* used
- The $\varphi_i$’s are the *types* of those resources, encoded as formulae in the service logic
- $\varphi$ is the type of service this service provider provides
- $Y$ is the set of this service provider’s current users
- $u(Y)$ is whether the service provider can provide the service to another user despite already serving current users $Y$
- $f(x_1, \ldots, x_n, Y)$ is the service provided to the newest user
Service Logic Example

- Let services be integers, and the service logic be elementary number theory
- The service provider sum uses as resources two even integers and provides another even integer
  - The use constraint $u$ always evaluates to true
- The service provider proves the formula:
  \[
  \frac{\text{even}(n_1) \land \text{even}(n_2) \land \text{true}}{\text{even}(\text{sum}(n_1, n_2))}
  \]
Semantics of Service Logic Example

Three levels of understanding:

- You who designed the service provider may have had a mental picture such as the following:
  
  which guided you in creating a proof of the service logic formula

- The service logic oracle knows the Peano axioms for the integers and the definitions of `even` and `sum`, so it can check your proof

- The service manager treats integers, predicates such as `even`, and functions such as `sum` as black boxes
Service Management

type servicemanager = blockedserviceprovider list
  * runningserviceprovider list

- Will show mutually recursive type declarations
  leading up to blockedserviceprovider and
  runningserviceprovider

- An \( n \)-ary service provider may be \textit{blocked} awaiting
  some of its resources \( x_1, \ldots, x_n \)

- When all these are provided, it will start \textit{running} and
  can provide its resource to others until it is all used
  up (its use constraint fails)
Proof-related Declarations

type goal = servicelogicformula * int

type proof_state n =
  ( resourcepool: runningserviceprovider list ) *
  ( goals: goal list ) *
  ( assignment n: int -> int )

and tactic = proof_state -> proof_state

and derivation n =
  ( tactic_sequence: tactic list ) *
  ( current_goals: goal list ) *
  ( current_assignment n: int -> int )

and...
Semantics of Proof-related Declarations

- **goal** $(\varphi, i)$ stands for $\varphi(x_i)$
- **assignment** maps $i$ with $1 \leq i \leq n$ to $j$ if $\text{runningserviceprovider } j$ is to provide service $x_i$, and maps $i$ to $-1$ if no such $\text{runningserviceprovider}$ has been found
- **tactic** advances proof state toward easier goals
  - backward tactic works on goals
  - forward tactic works on hypotheses (embedded in running service provider list)
- **derivation** is (almost) a proof state, plus sequence of tactics leading to it
Managed Service Provider Declarations

...and blockedserviceprovider n =
    serviceprovider n * derivation n
and serviceuse = runningserviceprovider * service
and runningserviceprovider n =
    serviceprovider n * derivation n true *
    ( resources: serviceuse list n ) *
    ( users: serviceuse list )
Completing Semantics of Derivations

- Recall: service provider proves rule
  \[\varphi_1(x_1) \land \cdots \land \varphi_n(x_n) \land u(Y) \land \varphi(f(x_1, \ldots, x_n, Y))\]

- Initial goal of registered service provider’s derivation is rule’s premise \(\varphi_1(x_1) \land \cdots \land \varphi_n(x_n)\)

- Initial proof state includes service manager’s runningserviceprovider list, assignment always to \(-1\)
Managed Service Provider Semantics

• When derivation can make no more progress, blocks
  – Current goals can be seen as pending service requests

• derivation n true means assignment maps 1,...,n to positive integers—the indices of resources in the runningserviceprovider list—and current goal list is empty

• if ((p_1, y_1),..., (p_k, y_k)) is a runningserviceprovider's user list, and u is its use constraint, then it must be the case that u(y_1,..., ˆy_i,..., y_k), for any i, i.e., the use constraint is not violated