CIS 500
Software Foundations
Fall 2006

November 15

From last time...

Decision Procedures (take 1)

A *decision function* for a relation $R \subseteq U$ is a *total* function p from U to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Decision Procedures (take 1)

A *decision function* for a relation $R \subseteq U$ is a *total* function p from U to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Example:

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

Note that, for now, we are saying absolutely nothing about *computability*. We'll come back to this in a moment.

Decision Procedures (take 1)

A *decision function* for a relation $R \subseteq U$ is a *total* function p from U to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Example:

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

The function p whose graph is

$$\begin{array}{l} \{\ ((1,2),\ true),\ ((2,3),\ true), \\ \quad ((1,1),\ false),\ ((1,3),\ false), \\ \quad ((2,1),\ false),\ ((2,2),\ false), \\ \quad ((3,1),\ false),\ ((3,2),\ false),\ ((3,3),\ false)\} \end{array}$$

is a decision function for R.

Decision Procedures (take 1)

A *decision function* for a relation $R \subseteq U$ is a *total* function p from U to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Example:

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

The function p' whose graph is

$$\{((1,2),\ true),\ ((2,3),\ true)\}$$

is *not* a decision function for R.

Decision Procedures (take 1)

A *decision function* for a relation $R \subseteq U$ is a *total* function p from U to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Example:

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

The function p'' whose graph is

$$\{((1,2),\ true),\ ((2,3),\ true),\ ((1,3),\ false)\}$$

is also *not* a decision function for R.

Decision Procedures (take 2)

Of course, we want a decision procedure to be a *procedure*.

A *decision procedure* for a relation $R \subseteq U$ is a *computable* total function p from U to $\{true, false\}$ such that $p(u) = true$ iff $u \in R$.

Example

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

Example

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

The function

$$\begin{array}{rl} p(x,y) = & \textit{if } x = 2 \textit{ and } y = 3 \textit{ then true} \\ & \textit{else if } x = 1 \textit{ and } y = 2 \textit{ then true} \\ & \textit{else false} \end{array}$$

whose graph is

$$\begin{array}{l} \{\ ((1,2),\ true),\ ((2,3),\ true), \\ \quad ((1,1),\ false),\ ((1,3),\ false), \\ \quad ((2,1),\ false),\ ((2,2),\ false), \\ \quad ((3,1),\ false),\ ((3,2),\ false),\ ((3,3),\ false)\} \end{array}$$

is a decision procedure for R.

Example

$$\begin{array}{rcl} U & = & \{1,2,3\} \\ R & = & \{(1,2),\ (2,3)\} \end{array}$$

The recursively defined partial function

$$\begin{array}{rl} p(x,y) = & \textit{if } x = 2 \textit{ and } y = 3 \textit{ then true} \\ & \textit{else if } x = 1 \textit{ and } y = 2 \textit{ then true} \\ & \textit{else if } x = 1 \textit{ and } y = 3 \textit{ then false} \\ & \textit{else } p(x,y) \end{array}$$

whose graph is

$$\{\ ((1,2),\ true),\ ((2,3),\ true),\ ((1,3),\ false)\}$$

is *not* a decision procedure for R.

Subtyping Algorithm

This recursively defined *total* function is a decision procedure for the subtype relation:

$subtype(\mathtt{S},\mathtt{T})$ =

- if $\mathtt{T} = \mathtt{Top}$, then *true*
- else if $\mathtt{S} = \mathtt{S}_1 \to \mathtt{S}_2$ and $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$
  - then $subtype(\mathtt{T}_1,\mathtt{S}_1)\ \wedge\ subtype(\mathtt{S}_2,\mathtt{T}_2)$
- else if $\mathtt{S} = \{\mathtt{k}_j{:}\mathtt{S}_j\ ^{j \in 1..m}\}$ and $\mathtt{T} = \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i \in 1..n}\}$
  - then $\{\mathtt{l}_i\ ^{i \in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j \in 1..m}\}$
    - $\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $\mathtt{k}_j = \mathtt{l}_i$ and $subtype(\mathtt{S}_j,\mathtt{T}_i)$
- else *false*.

To show this, we need to prove:

1. that it returns *true* whenever $\mathtt{S} <: \mathtt{T}$, and
2. that it returns either *true* or *false* on all inputs.

Subtyping Algorithm

But this recursively defined *partial* function is not:

subtype(S, T) =

  if $T = Top$, then *true*
  else if $S = S_1 \to S_2$ and $T = T_1 \to T_2$
    then *subtype*(T_1, S_1) $\wedge$ *subtype*(S_2, T_2)
  else if $S = \{k_j : S_j\ ^{j \in 1..m}\}$ and $T = \{l_i : T_i\ ^{i \in 1..n}\}$
    then $\{l_i\ ^{i \in 1..n}\} \subseteq \{k_j\ ^{j \in 1..m}\}$
      $\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $k_j = l_i$
        and *subtype*(S_j, T_i)
  else *subtype*(T,S)

Algorithmic Typing

Algorithmic typing

- How do we implement a type checker for the lambda-calculus with subtyping?
- Given a context Γ and a term t, how do we determine its type T, such that $\Gamma \vdash t : T$?

Issue

For the typing relation, we have just one problematic rule to deal with: subsumption.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \qquad \text{(T-SUB)}$$

We observed above that this rule is sometimes *required* when typechecking applications:

E.g., the term

`(λr:{x:Nat}. r.x) {x=0,y=1}`

is not typable without using subsumption.

But we *conjectured* that applications were the only critical uses of subsumption.

Plan

1. Investigate how subsumption is used in typing derivations by looking at examples of how it can be "pushed through" other rules
2. Use the intuitions gained from this exercise to design a new, algorithmic typing relation that
   - omits subsumption
   - compensates for its absence by enriching the application rule
3. Show that the algorithmic typing relation is essentially equivalent to the original, declarative one

Example (T-SUB with T-ABS)

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2} \qquad \dfrac{\vdots}{S_2 <: T_2}}{\Gamma, x{:}S_1 \vdash s_2 : T_2}\ \text{(T-SUB)}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to T_2}\ \text{(T-ABS)}$$

Example (T-Sub with T-Abs)

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2} \quad \dfrac{\vdots}{S_2 <: T_2}}{\Gamma, x{:}S_1 \vdash s_2 : T_2}\ (\text{T-Sub})}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to T_2}\ (\text{T-Abs})$$

becomes

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1\to S_2}\ (\text{T-Abs}) \quad \dfrac{\dfrac{}{S_1 <: S_1}\ (\text{S-Refl}) \quad \dfrac{\vdots}{S_2 <: T_2}}{S_1\to S_2 <: S_1 \to T_2}\ (\text{S-Arrow})}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1\to T_2}\ (\text{T-Sub})$$

Example (T-Sub with T-Rcd)

$$\dfrac{\text{for each } i \quad \dfrac{\dfrac{\vdots}{\Gamma\vdash t_i : S_i} \quad \dfrac{\vdots}{S_i <: T_i}}{\Gamma \vdash t_i : T_i}\ (\text{T-Sub})}{\Gamma \vdash \{l_i = t_i\ ^{i\in 1..n}\} : \{l_i{:}T_i\ ^{i\in 1..n}\}}\ (\text{T-Rcd})$$

Intuitions

These examples show that we do not need T-Sub to "enable" T-Abs or T-Rcd: given any typing derivation, we can construct a derivation *with the same conclusion* in which T-Sub is never used immediately before T-Abs or T-Rcd.

What about T-App?
We've already observed that T-Sub is required for typechecking some applications. So we expect to find that we *cannot* play the same game with T-App as we've done with T-Abs and T-Rcd. Let's see why.

Example (T-Sub with T-App on the left)

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : S_{11}\to S_{12}} \quad \dfrac{\dfrac{\vdots}{T_{11}<:S_{11}} \quad \dfrac{\vdots}{S_{12}<:T_{12}}}{S_{11}\to S_{12} <: T_{11}\to T_{12}}\ (\text{S-Arrow})}{\Gamma\vdash s_1 : T_{11}\to T_{12}}\ (\text{T-Sub}) \quad \dfrac{\vdots}{\Gamma\vdash s_2 : T_{11}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\ (\text{T-App})$$

Example (T-Sub with T-App on the left)

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : S_{11}\to S_{12}} \quad \dfrac{\dfrac{\vdots}{T_{11}<:S_{11}} \quad \dfrac{\vdots}{S_{12}<:T_{12}}}{S_{11}\to S_{12} <: T_{11}\to T_{12}}\ (\text{S-Arrow})}{\Gamma\vdash s_1 : T_{11}\to T_{12}}\ (\text{T-Sub}) \quad \dfrac{\vdots}{\Gamma\vdash s_2 : T_{11}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\ (\text{T-App})$$

becomes

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : S_{11}\to S_{12}} \quad \dfrac{\dfrac{\vdots}{\Gamma\vdash s_2 : T_{11}} \quad \dfrac{\vdots}{T_{11}<:S_{11}}}{\Gamma\vdash s_2 : S_{11}}\ (\text{T-Sub})}{\Gamma\vdash s_1\ s_2 : S_{12}}\ (\text{T-App}) \quad \dfrac{\vdots}{S_{12}<:T_{12}}}{\Gamma\vdash s_1\ s_2 : T_{12}}\ (\text{T-Sub})$$

Example (T-Sub with T-App on the right)

$$\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : T_{11}\to T_{12}} \quad \dfrac{\dfrac{\vdots}{\Gamma\vdash s_2 : T_2} \quad \dfrac{\vdots}{T_2<:T_{11}}}{\Gamma\vdash s_2 : T_{11}}\ (\text{T-Sub})}{\Gamma\vdash s_1\ s_2 : T_{12}}\ (\text{T-App})$$

Example (T-Sub with T-App on the right)

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash s_2 : T_2} \quad \dfrac{\vdots}{T_2 <: T_{11}}}{\Gamma \vdash s_2 : T_{11}}\,\text{(T-Sub)}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

becomes

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : T_{11}\to T_{12}} \quad \dfrac{\dfrac{\vdots}{T_2 <: T_{11}} \quad \dfrac{}{T_{12} <: T_{12}}\,\text{(S-Refl)}}{T_{11}\to T_{12} <: T_2\to T_{12}}\,\text{(S-Arrow)}}{\Gamma \vdash s_1 : T_2\to T_{12}}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_2}}{\Gamma \vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

Intuitions

So we've seen that uses of subsumption can be "pushed" from one of immediately before T-App's premises to the other, but cannot be completely eliminated.

Example (nested uses of T-Sub)

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash s : U}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash s : T}\,\text{(T-Sub)}$$

Example (nested uses of T-Sub)

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash s : U}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash s : T}\,\text{(T-Sub)}$$

becomes

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \qquad \dfrac{\dfrac{\vdots}{S <: U} \quad \dfrac{\vdots}{U <: T}}{S <: T}\,\text{(S-Trans)}}{\Gamma \vdash s : T}\,\text{(T-Sub)}$$

Summary

What we've learned:

- Uses of the T-Sub rule can be "pushed down" through typing derivations until they encounter either
  1. a use of T-App or
  2. the root fo the derivation tree.
- In both cases, multiple uses of T-Sub can be collapsed into a single one.

Summary

What we've learned:

- Uses of the T-Sub rule can be "pushed down" through typing derivations until they encounter either
  1. a use of T-App or
  2. the root fo the derivation tree.
- In both cases, multiple uses of T-Sub can be collapsed into a single one.

This suggests a notion of "normal form" for typing derivations, in which there is

- exactly one use of T-Sub before each use of T-App
- one use of T-Sub at the very end of the derivation
- no uses of T-Sub anywhere else.

Algorithmic Typing

The next step is to "build in" the use of subsumption in application rules, by changing the T-APP rule to incorporate a subtyping premise.

$$\frac{\Gamma \vdash t_1 : T_{11}{\to}T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

Given any typing derivation, we can now

1. normalize it, to move all uses of subsumption to either just before applications (in the right-hand premise) or at the very end
2. replace uses of T-APP with T-SUB in the right-hand premise by uses of the extended rule above

This yields a derivation in which there is just *one* use of subsumption, at the very end!

Minimal Types

But... if subsumption is only used at the very end of derivations, then it is actually *not needed* in order to show that any term is typable!
It is just used to give *more* types to terms that have already been shown to have a type.

In other words, if we dropped subsumption completely (after refining the application rule), we would still be able to give types to exactly the same set of terms — we just would not be able to give as many types to some of them.

If we drop subsumption, then the remaining rules will assign a *unique, minimal* type to each typable term.

For purposes of building a typechecking algorithm, this is enough.

Final Algorithmic Typing Rules

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1{\to}T_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad T_1 = T_{11}{\to}T_{12} \qquad \Gamma \mapsto t_2 : T_2 \qquad \mapsto T_2 <: T_{11}}{\Gamma \mapsto t_1\ t_2 : T_{12}} \qquad \text{(TA-APP)}$$

$$\frac{\text{for each } i \quad \Gamma \mapsto t_i : T_i}{\Gamma \mapsto \{l_1{=}t_1 \ldots l_n{=}t_n\} : \{l_1{:}T_1 \ldots l_n{:}T_n\}} \qquad \text{(TA-RCD)}$$

$$\frac{\Gamma \mapsto t_1 : R_1 \qquad R_1 = \{l_1{:}T_1 \ldots l_n{:}T_n\}}{\Gamma \mapsto t_1.l_i : T_i} \qquad \text{(TA-PROJ)}$$

Soundness of the algorithmic rules

Theorem: If $\Gamma \mapsto t : T$, then $\Gamma \vdash t : T$.

Completeness of the algorithmic rules

Theorem [Minimal Typing]: If $\Gamma \vdash t : T$, then $\Gamma \mapsto t : S$ for some $S <: T$.

Completeness of the algorithmic rules

Theorem [Minimal Typing]: If $\Gamma \vdash t : T$, then $\Gamma \mapsto t : S$ for some $S <: T$.

Proof: Induction on typing derivation. *(Details on this week's homework.)*

(N.b.: All the messing around with transforming derivations was just to build intuitions and decide what algorithmic rules to write down and what property to prove: the proof itself is a straightforward induction on typing derivations.)

Meets and Joins

Adding Booleans

Suppose we want to add booleans and conditionals to the language we have been discussing.
For the *declarative* presentation of the system, we just add in the appropriate syntactic forms, evaluation rules, and typing rules.

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{Bool} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T} \qquad \Gamma \vdash \texttt{t}_3 : \texttt{T}}{\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

A Problem with Conditional Expressions

For the *algorithmic* presentation of the system, however, we encounter a little difficulty.

What is the minimal type of

```
if true then {x=true,y=false} else {x=true,z=true}
```

?

The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

$$\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$$

any type that is a possible type of both $\texttt{t}_2$ and $\texttt{t}_3$.

So the *minimal* type of the conditional is the *least common supertype* (or *join*) of the minimal type of $\texttt{t}_2$ and the minimal type of $\texttt{t}_3$.

$$\frac{\Gamma \blacktriangleright \texttt{t}_1 : \texttt{Bool} \qquad \Gamma \blacktriangleright \texttt{t}_2 : \texttt{T}_2 \qquad \Gamma \blacktriangleright \texttt{t}_3 : \texttt{T}_3}{\Gamma \blacktriangleright \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}_2 \vee \texttt{T}_3} \qquad \text{(T-If)}$$

The Algorithmic Conditional Rule

More generally, we can use subsumption to give an expression

$$\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$$

any type that is a possible type of both $\texttt{t}_2$ and $\texttt{t}_3$.

So the *minimal* type of the conditional is the *least common supertype* (or *join*) of the minimal type of $\texttt{t}_2$ and the minimal type of $\texttt{t}_3$.

$$\frac{\Gamma \blacktriangleright \texttt{t}_1 : \texttt{Bool} \qquad \Gamma \blacktriangleright \texttt{t}_2 : \texttt{T}_2 \qquad \Gamma \blacktriangleright \texttt{t}_3 : \texttt{T}_3}{\Gamma \blacktriangleright \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}_2 \vee \texttt{T}_3} \qquad \text{(T-If)}$$

Does such a type exist for every $\texttt{T}_2$ and $\texttt{T}_3$??

Existence of Joins

Theorem: For every pair of types $\texttt{S}$ and $\texttt{T}$, there is a type $\texttt{J}$ such that

1. $\texttt{S} <: \texttt{J}$
2. $\texttt{T} <: \texttt{J}$
3. If $\texttt{K}$ is a type such that $\texttt{S} <: \texttt{K}$ and $\texttt{T} <: \texttt{K}$, then $\texttt{J} <: \texttt{K}$.

I.e., $\texttt{J}$ is the smallest type that is a supertype of both $\texttt{S}$ and $\texttt{T}$.

Examples

What are the joins of the following pairs of types?

1. {x:Bool,y:Bool} and {y:Bool,z:Bool}?
2. {x:Bool} and {y:Bool}?
3. {x:{a:Bool,b:Bool}} and {x:{b:Bool,c:Bool}, y:Bool}?
4. {} and Bool?
5. {x:{}} and {x:Bool}?
6. Top→{x:Bool} and Top→{y:Bool}?
7. {x:Bool}→Top and {y:Bool}→Top?

Meets

To calculate joins of arrow types, we also need to be able to calculate *meets* (greatest lower bounds)!

Unlike joins, meets do not necessarily exist.
E.g., Bool→Bool and {} have *no* common subtypes, so they certainly don't have a greatest one!

However...

Existence of Meets

Theorem: For every pair of types S and T, if there is any type N such that N <: S and N <: T, then there is a type M such that

1. M <: S
2. M <: T
3. If O is a type such that O <: S and O <: T, then O <: M.

I.e., M (when it exists) is the largest type that is a subtype of both S and T.

Jargon: In the simply typed lambda calculus with subtyping, records, and booleans...

- The subtype relation *has joins*
- The subtype relation *has* bounded *meets*

Examples

What are the meets of the following pairs of types?

1. {x:Bool,y:Bool} and {y:Bool,z:Bool}?
2. {x:Bool} and {y:Bool}?
3. {x:{a:Bool,b:Bool}} and {x:{b:Bool,c:Bool}, y:Bool}?
4. {} and Bool?
5. {x:{}} and {x:Bool}?
6. Top→{x:Bool} and Top→{y:Bool}?
7. {x:Bool}→Top and {y:Bool}→Top?

Calculating Joins

$$
S \vee T = \begin{cases}
\mathtt{Bool} & \text{if } S = T = \mathtt{Bool} \\
M_1 \to J_2 & \text{if } S = S_1 \to S_2 \quad T = T_1 \to T_2 \\
 & \quad S_1 \wedge T_1 = M_1 \quad S_2 \vee T_2 = J_2 \\
\{j_l : J_l\ ^{l \in 1..q}\} & \text{if } S = \{k_j : S_j\ ^{j \in 1..m}\} \\
 & \quad T = \{l_i : T_i\ ^{i \in 1..n}\} \\
 & \quad \{j_l\ ^{l \in 1..q}\} = \{k_j\ ^{j \in 1..m}\} \cap \{l_i\ ^{i \in 1..n}\} \\
 & \quad S_j \vee T_i = J_l \quad \text{for each } j_l = k_j = l_i \\
\mathtt{Top} & \text{otherwise}
\end{cases}
$$

Calculating Meets

$$
S \wedge T = \begin{cases}
S & \text{if } T = \mathtt{Top} \\
T & \text{if } S = \mathtt{Top} \\
\mathtt{Bool} & \text{if } S = T = \mathtt{Bool} \\
J_1 \to M_2 & \text{if } S = S_1 \to S_2 \quad T = T_1 \to T_2 \\
 & \quad S_1 \vee T_1 = J_1 \quad S_2 \wedge T_2 = M_2 \\
\{m_l : M_l\ ^{l \in 1..q}\} & \text{if } S = \{k_j : S_j\ ^{j \in 1..m}\} \\
 & \quad T = \{l_i : T_i\ ^{i \in 1..n}\} \\
 & \quad \{m_l\ ^{l \in 1..q}\} = \{k_j\ ^{j \in 1..m}\} \cup \{l_i\ ^{i \in 1..n}\} \\
 & \quad S_j \wedge T_i = M_l \quad \text{for each } m_l = k_j = l_i \\
 & \quad M_l = S_j \quad \text{if } m_l = k_j \text{ occurs only in } S \\
 & \quad M_l = T_i \quad \text{if } m_l = l_i \text{ occurs only in } T \\
fail & \text{otherwise}
\end{cases}
$$