Algorithmic Typing

How do we implement a type checker for the lambda-calculus with subtyping?

Issue

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

\[
\frac{\Gamma \vdash t : T}{\Gamma \vdash t : S}
\]  

subsumption

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

\[
\forall T, S, T : \text{Nat} \quad \frac{\Gamma \vdash x : T}{\Gamma \vdash (x \cdot x : T)}
\]  

The term application: 

We observed last time that this rule is sometimes required when typing applications. We have just one problematic rule to deal with:

\[
\frac{\Gamma \vdash f : T \rightarrow S}{\Gamma \vdash f \cdot t : S}
\]  

subsumption

Given a context \( \Gamma \) and a term \( t \), how do we determine its type? Such that

\[
\frac{\Gamma \vdash t : P}{\Gamma \vdash \text{Issue}}
\]  

subtyping

How do we implement a type checker for the lambda-calculus with subtyping?
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 subsumption relation.
3. Show that the algorithmic subsumption relation is essentially equivalent to the original, declarative one.

Example (T-Sub with T-Rcd)

Example (T-Sub with T-Abs)

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

When about T-App?

Some conclusions in which T-Sub is never used immediately before T-App or T-Red.

These examples show that we do not need T-Sub to enable T-App or

(1) T-App

(2) T-Sub

(3) T-Red

Let's see why.

We've already observed that T-Sub is required for preserving some applications. So we expect to find that we cannot play the same game with

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

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

Example (nested uses of T-Sub)

Intuitions

Example (nested uses of T-Sub)

Intuitions

Example (nested uses of T-Sub)
For purposes of building a type-checking algorithm, this is enough. If we drop subsumption, then the remaining rules will assign a unique type.

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
2. the root of the derivation tree.

In both cases, multiple uses of T-Sub can be collapsed into a single one.

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. In other words, if we dropped subsumption completely (after refining the T-App rule to incorporate a subtyping premise), we can now replace uses of T-App with T-Sub in the right-hand premise of all the rules.

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

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, 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.

Summary

What we've learned:

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

In both cases, multiple uses of T-Sub can be collapsed into a single one.

Minimal Types

If we drop subsumption, then it is actually not needed in order to show that any term is typable.

Inotherwords,ifwedroppedsubsumptioncompletely(afterrefiningtheapplicationrule),wewouldstillbeabletogivetypestoexactlythesamesetof

terms—wewouldjustneedtogettoasmanyTyposofsometermine.

But...ifsubsumptionisonlyusedattheveryendofderivations,thenit

is actually not needed in order to show that any term is typable.

Minimal Types

If we drop subsumption, then it is actually not needed in order to show that any term is typable.

In other words, if we dropped subsumption completely (after refining the T-App rule to incorporate a subtyping premise), we can now replace uses of T-App with T-Sub in the right-hand premise of all the rules.

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

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, 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.

Summary

What we've learned:

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

In both cases, multiple uses of T-Sub can be collapsed into a single one.

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. In other words, if we dropped subsumption completely (after refining the application rule), we would still be able to get as many Typos of some term.

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.

Minimal Types

If we drop subsumption, then it is actually not needed in order to show that any term is typable.

In other words, if we dropped subsumption completely (after refining the T-App rule to incorporate a subtyping premise), we can now replace uses of T-App with T-Sub in the right-hand premise of all the rules.

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

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, 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.

Summary

What we've learned:

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

In both cases, multiple uses of T-Sub can be collapsed into a single one.
Theorem: Final Algorithmic Typing Rules

\[ T \vdash ! T \] \[ T \vdash \top \] \[ T \vdash \bot \]

Soundness of the algorithmic rules

Proof: Induction on typing derivation.

Theorem: Soundness of the algorithmic rules

\[ T \vdash \top \] \[ T \vdash \bot \]

Completeness of the algorithmic rules

Theorem: Completeness of the algorithmic rules

\[ T \vdash \bot \] \[ T \vdash \top \]

Proof: Induction on typing derivation.

(N.b.: All the mess 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 straightforward inductive proof on typing derivations.)
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.

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

A Problem with Conditional Expressions

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 of the algorithmic conditional:

\[
\begin{align*}
\text{if} & \text{ } t_1 \text{ then } t_2 \text{ else } t_3 \\
\text{where } & \text{ } t_1 : \text{Bool} \\
\text{then } & \text{ } t_2 : T_2 \\
\text{else } & \text{ } t_3 : T_3
\end{align*}
\]

So the minimal type of the conditional is the least common supertype (or join) of the minimal type of \( T_2 \) and the minimal type of \( T_3 \). Any type that is a possible type of both \( T_2 \) and \( T_3 \):

\[
\text{if} \text{ } t_1 \text{ then } t_2 \text{ else } t_3
\]

What is the minimal type of \( T_2 \) and \( T_3 \)?

For the declarative presentation of the system, we just add in the appropriate syntactic forms, evaluation rules, and typing rules.

Meet and Joins

Meets and Joins
More generally, we can use subsumption to give an expression

The Algorithmic Conditional Rule

However...

There exists a greatest lower bound (greatest lower bounds)

To calculate joins of arrow types, we also need to be able to calculate meets

Examples

Does such a type exist for every $T_2$ and $T_3$?

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$

$L = T_1 \land T_2 \land T_3$
Existence of Meets

Theorem:

Existence of Meets

Theorem: For every pair of types \( S \) and \( T \) there is any type \( N \) such that

- \( N \) is a type such that \( 0 < N \),
- \( N > 0 \),
- \( N > S \) and \( N > T \),
- \( \exists N \) such that there are any \( N \) such that


Examples

1. What are the meets of the following pairs of types?
   - \( \{x: \text{Bool}, y: \text{Bool}\} \text{ and } \{y: \text{Bool}, z: \text{Bool}\} \)
   - \( \{x: \text{Bool}\} \text{ and } \{y: \text{Bool}\} \)
   - \( \{x: \text{Bool}, y: \text{Bool}, z: \text{Bool}\} \text{ and } \{y: \text{Bool}, z: \text{Bool}\} \)