Lecture Notes on Subtyping

15-836: Substructural Logics
Frank Pfenning

Lecture 8
September 21, 2023

1 Introduction

So far, we have always worked under the presupposition that the provider and client of a channel agree on its type. This is fundamentally inspired by the cut rule in logic, and it also seems necessary to ensure that all messages are properly understood. For example, the progress property would fail spectacularly if one process sends a label while the recipient expects a channel.

In this lecture we consider if we can loosen this restriction without violating progress and preservation. If we can, it might allow us to simplify some programs, or to capture more properties of the programs we write in their type.

As an introductory example, consider the following two types.

$$\begin{array}{l} \mathsf{nat} = \{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \\ \mathsf{pos} = \{\phantom{\mathsf{zero} : \mathbf{1},} \mathsf{succ} : \mathsf{nat}\} \end{array}$$

If we have a provider $- \vdash P :: (n : \mathsf{pos})$ and a client $(n : \mathsf{nat}) \vdash Q :: -$ then nothing can go wrong. P restricts itself to start with the message succ but the client does not have to be aware of this—it will simply not receive a first message zero. On the other hand, if we have $- \vdash P :: (n : \mathsf{nat})$ and $(n : \mathsf{pos}) \vdash Q :: -$ then things can go wrong immediately because P could send the label zero that Q is not expecting.

The notion of subtyping we consider here has been developed by Gay and Hole [2005]. Although for a different underlying programming language, the result is essentially the same, except that in their setting the roles of sender and receiver are reversed from ours.

2 Message Understood

The key to subtyping in the message-passing setting is to make sure that the recipient of a message is ready for every possible message it could receive. Semantically, we define subtyping $A \leq B$ this way:

> *If $\Delta \vdash P :: (a : A)$ and $A \leq B$ and $\Delta', a : B \vdash Q :: (c : C)$ then every message along channel a is understood by the receiver.*

For our purposes of study here, we'd like the relation $A \leq B$ to be as large as possible. Or, to put it another way, if $A \not\leq B$ then there should be counterexample, that is, a message along the channel a that the recipient does not understand. By "does not understand" we mean that in the refactored rule from the last lecture

$$\mathsf{proc}(\mathbf{send}\ a\ m\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ K) \longrightarrow \mathsf{proc}(P), \mathsf{proc}(m \triangleright K)$$

the operation $m \triangleright K$ is undefined.

In order to see what kind of subtyping might hold we walk through the critical steps in type preservation and progress in a hand-wavy fashion. These form the core of the proof of progress and preservation in the presence of subtyping. An important point here is to think connective by connective, so that it is open-ended and adaptable to other languages.

Before we get to the specifics, there are a few general properties we expect. We expect these to be *admissible* and rather than primitive.

- Subtyping should be reflexive: $A \leq A$ for all types A. This vaguely corresponds to identity.
- Subtyping should be transitive: if $A \leq B$ and $B \leq C$, then $A \leq C$. This vaguely corresponds to cut.
- Right subsumption should be admissible: If $\Delta \vdash P :: (a : A)$ and $A \leq B$ then also $\Delta \vdash P :: (a : B)$.
- Left subsumption should be admissible: If $A \leq B$ and $\Delta, b : B \vdash P :: (c : C)$ then $\Delta, b : A \vdash P :: (c : C)$

We now build up a set of rules that allow us to conclude $A \leq B$ as a judgment.

One thing we can say immediately based on the semantic definition: there should be no rules for $A \leq B$ if the top level type constructor of A and B is different. For example, $1 \not\leq A \otimes B$ and $A \multimap B \not\leq \oplus\{\ell : A_\ell\}_{\ell \in L}$. In all these cases, a message sent along the channel will not be understood by the recipient.

3 Internal Choice and Unit

As the example in the introduction suggests, $\oplus\{\ell : A_\ell\}_{\ell \in L} \leq \oplus\{k : B_k\}_{k \in K}$ requires that every label in L must also be in K. That's because the provider will send some $\ell \in L$ along channel a so the recipient must be ready for it. But that's not quite sufficient: after a label $\ell \in L$ is sent, the two processes will still be connected along

a, but now the provider will have type A_ℓ and the client B_ℓ, so one must be a subtype of the other. Pictorially:

$$\begin{tikzcd}
\text{provider } P & \text{client } Q \\
\oplus\{\ell : A_\ell\}_{\ell \in L} \arrow[r, dash, "\leq", "a"'] \arrow[r, bend right, "\textbf{send}\ a\ \ell"'] \arrow[d, dashed] & \oplus\{k : B_k\}_{k \in K} \arrow[d, dashed] \\
A_\ell \arrow[r, dash, "\leq", "a"'] & B_\ell
\end{tikzcd}$$

From this we extract the rule

$$\frac{L \subseteq K \quad A_\ell \leq B_\ell \ (\forall \ell \in L)}{\oplus\{\ell : A_\ell\}_{\ell \in L} \leq \oplus\{k : B_k\}_{k \in K}}$$

We also have $\mathbf{1} \leq \mathbf{1}$ without any condition since the channel a is closed.

$$\begin{tikzcd}
\text{provider} & \text{client} \\
\mathbf{1} \arrow[r, dash, "\leq", "a"'] \arrow[r, bend right, "\textbf{send}\ a\ (\,)"'] & \mathbf{1}
\end{tikzcd}$$

The corresponding rule

$$\frac{}{\mathbf{1} \leq \mathbf{1}}$$

has no premise because the unit message closes the channel.

At this point we can already show some examples. Since type definitions are equirecursive we just unfold them.

$$\cfrac{\cfrac{\mathsf{nat} \leq \mathsf{nat}}{\oplus\{\mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{pos} \leq \mathsf{nat}}$$

Here we stopped at reflexivity. But if we think of reflexivity as just admissible, we would continue:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \leq 1} \qquad \mathsf{nat} \leq \mathsf{nat}}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{nat} \leq \mathsf{nat}}}{\oplus\{\mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{pos} \leq \mathsf{nat}}$$

At this point we realize we can continue indefinitely building a deeper and deeper derivation by expanding the recursive definition. But somehow that should be okay: if there is no subproof where we actually get stuck then there should be no "message not understood" problem when the processes communicate. So the proof system is *infinitary*. Even infinite derivations are sufficient to guarantee that there is no finite counterexample.

Another way to express this is to say that the proof rules here are interpreted *coinductively*. A proof is valid if we can always proceed further along all the open branches. This is in contrast to the proof systems we have seen so far, where proofs are defined *inductively*: we are only satisfied if we have a finite proof constructed from the rules.

In general, coinductive proofs system are more difficult to work with because we cannot actually write down infinite proofs. But they can still serve a useful purpose when they capture an intuitive notion. Here, and in some other cases I am aware of, they capture the absence of a counterexample. Constructively, this is not the same as a direct proof, but a refutation of its negation and therefore in some sense "weaker" than a (constructive inductive) proof.

In this particular example, we actually have a finitary representation of an infinitary proof since we have reached a cycle: the judgment at the top is the same as one lower in the same proof branch. In that case we can mark it as a loop and not explore this branch further. This way to proceed is sound since we could always unfold the looping proof into an infinite one. Or we can say that if there were a counterexample, there would be a shortest one. But the shortest one wouldn't go through the same judgment more than once.

In lecture we noted this with arcs, but in LaTeX we just label the lower judgment in the proof and then use this to justify the leaf.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \leq 1} \qquad \dfrac{(x)}{\mathsf{nat} \leq \mathsf{nat}}}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{nat} \leq \mathsf{nat} \quad (x)}}{\oplus\{\mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{pos} \leq \mathsf{nat}}$$

In this particular example it seems like we should have been able to avoid the loop altogether by promoting reflexivity to be a rule. In other example, this is not possible. For example:

$$\begin{array}{l}
\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \\
\mathsf{even} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\} \\
\mathsf{odd} = \oplus\{\mathsf{succ} : \mathsf{even}\}
\end{array}$$

We start to construct circular proof of $\mathsf{even} \le \mathsf{nat}$:

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1} \le \mathbf{1}} \qquad \begin{array}{c}\vdots \\ \mathsf{odd} \le \mathsf{nat}\end{array}}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\} \le \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{even} \le \mathsf{nat}}$$

We see that we have "reduced" the question of $\mathsf{even} \le \mathsf{nat}$ to the question if $\mathsf{odd} \le \mathsf{nat}$. We go on until we can complete all branches in the proof.

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1} \le \mathbf{1}} \qquad \dfrac{\dfrac{\begin{array}{c}(x) \\ \mathsf{even} \le \mathsf{nat}\end{array}}{\oplus\{\mathsf{succ} : \mathsf{even}\} \le \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{odd} \le \mathsf{nat}}}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\} \le \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{even} \le \mathsf{nat} \quad (x)}$$

In order to explore a failure of subtyping, consider the judgment $\mathsf{nat} \le \mathsf{even}$. Clearly, this should not be provable.

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1} \le \mathbf{1}} \qquad \dfrac{\begin{array}{c}\text{fails, since } \{\mathsf{zero}, \mathsf{succ}\} \not\subseteq \{\mathsf{succ}\} \\ \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \le \oplus\{\mathsf{succ} : \mathsf{even}\}\end{array}}{\mathsf{nat} \le \mathsf{odd}}}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \le \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\}}}{\mathsf{nat} \le \mathsf{even}}$$

Here we observe that this particular *failed* derivation contains enough information to extract a sequence of messages where the last one is not expected by the recipient: succ followed by zero. And, indeed, this sequence of message is (the start of) the number 1 which is not even.

4 Example: Binary Numbers in Standard Form

As another example we consider binary numbers. Their representation is not uniquely determined because of the possibility of leading zeros. For example, $(0100)_2 = (100)_2 = 4$. We say a number is in standard form if it has no leading zeros. We can define this using positive numbers as an additional type.

$$\begin{aligned}
&\mathsf{bin} = \oplus\{\mathsf{b0} : \mathsf{bin}, \mathsf{b1} : \mathsf{bin}, \mathsf{e} : \mathbf{1}\}\\
&\mathsf{std} = \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}, \mathsf{e} : \mathbf{1}\}\\
&\mathsf{pos} = \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\}
\end{aligned}$$

Then $\mathsf{pos} \leq \mathsf{std}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{(x)}{\mathsf{pos} \leq \mathsf{pos}} \quad \overset{\vdots}{\mathsf{std} \leq \mathsf{std}}}{\oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\} \leq \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\}}}{\mathsf{pos} \leq \mathsf{pos}}\;(x) \qquad \overset{\vdots}{\mathsf{std} \leq \mathsf{std}}}{\oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\} \leq \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}, \mathsf{e} : \mathbf{1}\}}}{\mathsf{pos} \leq \mathsf{std}}$$

This example is a (mild) illustration of a concern about circular proofs: we do not transfer what we learn on one branch to another. One technique to deal with this is to turn an infinitary (circular) proof system into a saturation procedure that works with forward inference. In such a system there is more reuse. DeYoung et al. [2023] then justify the saturating rules with respect to the infinitary rules.

In our system for subtyping (incomplete, at this point), attempts are constructing circular proofs will always either fail finitely or end up with a (finite) circular proof. The reason is that in a signature in which n syntactically different types occur, there can be at most n^2 pairs $A \leq B$ that might appear on a branch in the proof. This will continue to be the case when our set of rules is complete.

5 Tensor

Let's consider the interaction when $a : A_1 \otimes A_2$. The provider will send a channel $b : A_1$ and the channel a will afterwards have type A_2.

$$\begin{tikzcd}
\text{provider } P & \text{client } Q \\
A_1 \otimes A_2 \arrow[r, dash, "\leq", "a"'] \arrow[r, bend right=50, "\textbf{send}\ a\ b"'] \arrow[d, dashed] & B_1 \otimes B_2 \arrow[d, dashed] \\
A_2 \arrow[r, dash, "\leq", "a"'] & B_2
\end{tikzcd}$$

From this we see that $A_2 \leq B_2$ is required for the connection to continue to be well-typed. But what is the situation with the channel b, provided by, say R? We have the following chain of reasoning:

1. P was the original client of b at type A_1.
2. Q will be its new client of b at type B_1.
3. R provided the channel to P at some type C_1, so $C_1 \leq A_1$.

So for R and Q to be properly connected over the channel b we must have $A_1 \leq B_1$ because then $C_1 \leq B_1$ follows by transitivity.

Actually, we can be more lenient that what we just described. Provider P can send a channel $b : A_1'$ as long as $A_1' \leq A_1$. Then we get:

1. P was the original client of b at type A_1'.
2. Q is the new client of b at type B_1.
3. $A_1' \leq A_1$ (the condition for P to send b along $a : A_1 \otimes A_2$)
4. R is the provider at some type C_1, so $C_1 \leq A_1'$.

Still, $A_1 \leq B_1$ is sufficient to guarantee the chain of subtyping from the provider R to the new client Q: $C_1 \leq A_1' \leq A_1 \leq B_1$. Without the condition, if $C_1 = A_1' = A_1 \not\leq B_1$ an incorrect situation would arise, leading to the potential of a "message not understood" error along channel b later. So our rule is just:

$$\frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 \otimes A_2 \leq B_1 \otimes B_2}$$

We can exploit subtyping as sketched above to generalize the $\otimes R^*$ rule.

$$\frac{A_1' \leq A_1 \quad \Delta \vdash P :: (x : A_2)}{\Delta, y : A_1' \vdash \textbf{send}\ x\ y\ ;\ P :: (x : A_1 \otimes A_2)}\ \otimes R^*$$

6 Negative Types

For negative types, the role of sender and receive are reversed, so we need to reexamine the situation carefully. We start with external choice, that should be analogous to internal choice.

$$\begin{tikzcd}
\text{provider } P & \text{client } Q \\
\&\{\ell : A_\ell\}_{\ell \in L} \arrow[r, dash, "\leq", "a"'] \arrow[d, dashed] & \&\{k : B_k\}_{k \in K} \arrow[l, bend left=40, "\textbf{send}\ a\ k"] \arrow[d, dashed] \\
A_k \arrow[r, dash, "\leq", "a"'] & B_k
\end{tikzcd}$$

We see that for the label k to be understood, we need that $k \in L$, so we must require that $L \supseteq K$, the opposite inclusion from the internal choice. However, provider and client remain the same, so the continuation types must be related in the same order.

$$\frac{L \supseteq K \quad A_k \leq B_k\ (\forall k \in K)}{\&\{\ell : A_\ell\}_{\ell \in L} \leq \&\{k : A_k\}_{k \in K}}$$

In the dynamics of linear implication a channel b is received by the *provider* along a.

$$\begin{tikzcd}
\text{provider} & \text{client} \\
A_1 \multimap A_2 \arrow[r, dash, "\leq", "a"'] \arrow[d, dashed] & B_1 \multimap B_2 \arrow[l, bend left=40, "\textbf{send}\ a\ b"] \arrow[d, dashed] \\
A_2 \arrow[r, dash, "\leq", "a"'] & B_2
\end{tikzcd}$$

The handoff of the channel b leads to the following reasoning.

1. A process R provides b at type $C_1 \leq B_1'$.
2. The client Q sees b at type $B_1' \leq B_1$.
3. The new client P sees b at type A_1.

So for the new connection to be well-typed, we need $C_1 \leq A_1$. We can get this by $C_1 \leq B_1' \leq B_1 \leq A_1$, so we should require $B_1 \leq A_1$. This is the manifestation of *contravariance of subtyping for arguments at function type* in this context.

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \multimap A_2 \leq B_1 \multimap B_2}$$

We have already anticipated the generalization of the typing rule for sending a channel.

$$\frac{A_1' \leq A_1 \quad \Delta, x : A_2 \vdash P :: (z : C)}{\Delta, y : A_1', x : A_1 \multimap A_2 \vdash \mathbf{send}\ x\ y\ ;\ P :: (z : C)} \multimap L^*$$

We can look among the rules for opportunities for generalization. The only other place where types are compared for equality in the rules so far is the identity rule and, depending on how one looks at it, the cut rule. We generalize identity.

$$\frac{A' \leq A}{y : A' \vdash \mathsf{fwd}\ x\ y :: (x : A)} \mathsf{id}$$

You should convince yourself that this rule is correct by simple transitivity reasoning.

We do not generalize cut, because due to the admissibility of left and right subsumption, this would complicate the syntax without changing the set of well-typed processes. However, we do generalize the call rule (see Figure 2).

7 Example: Subtyping of Stores

As an example of subtyping with negative types we consider the store interface from before.

$$\begin{aligned} \mathsf{store} = \&\{\ & \mathsf{ins} : \mathsf{bin} \multimap \mathsf{store}, \\ & \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}\ \}\ \} \end{aligned}$$

There are uses of a stack where it is important that we only insertions followed only by deletions until the stack is empty. For example, the amortized analysis of queues, implemented by two stacks, relies on a property along these lines [Okasaki, 1998].

This is an example of an interaction protocol with a data structure prescribed by a type. In object-oriented programming related techniques have been referred to as *typestate analysis* Strom and Yemini [1986].

We have two phases of communication, store^1 where we only insert (until the first deletion) and store^2 where we only delete. This is expressed in the following two types.

$$\begin{array}{lcl}\mathsf{store}^1 & = & \&\{\ \mathsf{ins} : \mathsf{bin} \multimap \mathsf{store}^1, \\ & & \phantom{\&\{\ }\mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\ \}\ \} \\ \mathsf{store}^2 & = & \&\{\ \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\ \}\ \}\end{array}$$

What are the expected subtyping relationships between store, store^1 and store^2? We suggest you work this out for yourself before you read on.

Maybe, like me, you got it wrong and conjectured, for examples, that $\mathsf{store}^2 \leq \mathsf{store}^1$. Let's see if we can prove this or find a counterexample.

$$\frac{\begin{array}{c}\text{fails, since } \{\mathsf{del}\} \not\supseteq \{\mathsf{del}, \mathsf{ins}\} \\ \&\{\mathsf{del} : \ldots\} \leq \&\{\mathsf{del} : \ldots, \mathsf{ins} : \ldots\}\end{array}}{\mathsf{store}^2 \leq \mathsf{store}^1}$$

Oops! And, indeed, if the client sees the type with both insert and delete options, it could send del. This message will not be understood by a provider in the phase 2, expecting only deletions.

So the relationship is the other way around. We skip some intermediate unfolding of type definitions. The missing part is a simple instance of reflexivity, which we have marked as an instance of an admissible rule.

$$\frac{\{\mathsf{ins}, \mathsf{del}\} \supseteq \{\mathsf{del}\} \qquad \dfrac{}{\oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\} \leq \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\}}}{\mathsf{store}^1 \leq \mathsf{store}^2}$$

We also have $\mathsf{store} \leq \mathsf{store}^1$ by a derivation that should not be surprising at this point.

$$\frac{\dfrac{\dfrac{}{\mathsf{bin} \leq \mathsf{bin}} \qquad \dfrac{(x)}{\mathsf{store} \leq \mathsf{store}^1}}{\mathsf{bin} \multimap \mathsf{store} \leq \mathsf{bin} \multimap \mathsf{store}^1} \qquad \dfrac{\dfrac{}{\mathbf{1} \leq \mathbf{1}} \qquad \dfrac{\dfrac{}{\mathsf{bin} \leq \mathsf{bin}} \qquad \dfrac{\mathcal{D}}{\mathsf{store} \leq \mathsf{store}^2}}{\mathsf{bin} \otimes \mathsf{store} \leq \mathsf{bin} \otimes \mathsf{store}^2}}{\oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}\,\} \leq \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\}}}{\mathsf{store} \leq \mathsf{store}^1 \quad (x)}$$

$$\mathcal{D} = \frac{\dfrac{\dfrac{}{\mathbf{1} \leq \mathbf{1}} \qquad \dfrac{\dfrac{}{\mathsf{bin} \leq \mathsf{bin}} \qquad \dfrac{(y)}{\mathsf{store} \leq \mathsf{store}^2}}{\mathsf{bin} \otimes \mathsf{store} \leq \mathsf{bin} \otimes \mathsf{store}^2}}{\oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}\,\} \leq \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\}}}{\mathsf{store} \leq \mathsf{store}^2 \quad (y)}$$

8 Subtyping in MPASS

Subtyping is implemented in MPASS and will be used when it is called with `--subtyping`, or `-s` for short.

There is no separate declaration to test subtyping, but we can use forwarding because $x : A \vdash \mathbf{fwd}\ y\ x :: (y : B)$ is well-typed if and only if $A \leq B$. Examples can be found in the file lecture8.mps; excerpts are in Listing 1 and Listing 2.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}

type std = +{'b0 : pos, 'b1 : std, 'e : 1}
type pos = +{'b0 : pos, 'b1 : std        }

proc pos_std (y : std) (x : pos) = fwd y x  % pos <: std
proc std_bin (y : bin) (x : std) = fwd y x  % std <: bin

fail
proc bin_std (y : std) (x : bin) = fwd y x  % bin </: std
```

Listing 1: Subtyping for some subsets of binary numbers

The last declaration in Listing 2 illustrates how we can test that A is *not* a subtype of B. We do this by using the construct **fail** `<dec>` which succeeds if the declaration `<dec>` fails. If you run this through MPASS with the `-d` flag it will still show the error message it would have printed if the declaration were not preceded by **fail**.

The only example of testing subtyping on programs are the `list2store` and `store2list` processes. The first, only *inserts* numbers into a store, while the second only *deletes* them, so they use the store^1 and store^2 types from this lecture. You can find the code in Listing 2.

9 Summary

We summarize the rules for subtyping, interpreted coinductively, in Figure 1 and the updated rules for process typing in Figure 2. The other rules remain unchanged.

References

Henry DeYoung, Andreia Mordido, Frank Pfenning, and Ankush Das. Parametric subtyping for structural parametric polymorphism. *CoRR*, abs/2307.13661, July 2023. URL `https://arxiv.org/abs/2307.13661`. Submitted.

Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12 (1):157–171, 1986.

```
type list = +{'nil : 1, 'cons : bin * list}

type store =  &{'ins : bin -o store,
                'del : +{'none : 1, 'some : bin * store}}
type store1 = &{'ins : bin -o store1,
                'del : +{'none : 1, 'some : bin * store2}}
type store2 = &{'del : +{'none : 1, 'some : bin * store2}}

(* note use of 'store1' below! *)
proc list2store (s : store1) (l : list) (t : store1) =
     recv l ( 'nil => recv l (() => fwd s t)
            | 'cons => recv l (x =>
                       send t 'ins ;
                       send t x ;
                       call list2store s l t) )

(* note use of 'store2' below! *)
proc store2list (l : list) (s : store2) =
     send s 'del ;
     recv s ( 'none => recv s (() => send l 'nil ; send l ())
            | 'some => recv s (x => send l 'cons ; send l x ;
                                    call store2list l s) )

proc roundtrip (l : list) (k : list) =
     e <- call empty e ;             % start with empty store
     s <- call list2store s k e ;    % add all elements from k
     call store2list l s             % retrieve all element from s
```

Listing 2: Phase 1 and 2 store typing

$$\frac{L \subseteq K \quad A_\ell \leq B_\ell \ (\forall \ell \in L)}{\oplus\{\ell : A_\ell\}_{\ell \in L} \leq \oplus\{k : B_k\}_{k \in K}}$$

$$\frac{}{\mathbf{1} \leq \mathbf{1}} \qquad \frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 \otimes A_2 \leq B_1 \otimes B_2}$$

$$\frac{L \supseteq K \quad A_k \leq B_k \ (\forall k \in K)}{\&\{\ell : A_\ell\}_{\ell \in L} \leq \&\{k : A_k\}_{k \in K}} \qquad \frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \multimap A_2 \leq B_1 \multimap B_2}$$

Figure 1: Subtyping, rules interpreted coinductively

$$\frac{A' \leq A}{y : A' \vdash \mathsf{fwd}\ x\ y :: (x : A)}\ \mathsf{id}$$

$$\frac{f\ (x : A')\ \overline{(y_i : B_i')} = P \in \Sigma \quad B_i \leq B_i'\ (\forall i) \quad A' \leq A}{\overline{y_i : B_i} \vdash \mathbf{call}\ f\ x\ \overline{y_i} :: (x : A)}\ \mathsf{call}$$

$$\frac{A_1' \leq A_1 \quad \Delta \vdash P :: (x : A_2)}{\Delta, y : A_1' \vdash \mathbf{send}\ x\ y\ ;\ P :: (x : A_1 \otimes A_2)}\ {\otimes}R^*$$

$$\frac{A_1' \leq A_1 \quad \Delta, x : A_2 \vdash P :: (z : C)}{\Delta, y : A_1', x : A_1 \multimap A_2 \vdash \mathbf{send}\ x\ y\ ;\ P :: (z : C)}\ {\multimap}L^*$$

Figure 2: Process typing, extended for subtyping