Type Conversion

Arithmetic subtyping requires type conversion.

- Arguments to floating point operations check for integers.
- Integers are silently converted to floating point.

Costly to implement directly!

Type Conversion

Observation: conversions can be made at compile time.

- Mixed-mode expressions rely on subsumption during type checking.
- We can detect statically where subsumption is used.

Another strategy is to use coercions.

- For each subtyping $\sigma <: \tau$, then there is a (unique) coercion function $\sigma \rightarrow \tau$.
- Uses of subsumption insert coercion from sub- to super-type.

Coherence

Issue: there might be many (different?) ways to insert coercions!

- Typing rules are not syntax-directed.
- Each typing derivation determines a different program.

Coherence of subtyping means that all ways are "equivalent".

Consider the expression $\sin(3)$.
We may "read" this in two ways:

1. Convert 3 to floating point, then apply $\sin$ to the result.
2. Convert $\sin$ to the function that, when applied, converts its integer argument to floating point, then applies floating point $\sin$ to the result.

These are equivalent, but they are not identical.
Coercion Interpretation

Assess with each subtype relation a coercion mapping the sub- to the super-type:

\[ \sigma \leq_{\tau} \tau \rightarrow v \]

The expression \( v \) is a value of type \( \sigma \rightarrow \tau \).

Coercion Interpretation of Subtyping

1. Primitive conversion: to_float.
2. Identity: id\( \tau \) in \( \tau \rightarrow v \).
3. Composition: \( v_1 \circ v_2 = \text{fn } x : \tau \rightarrow \text{fn } x : \tau \rightarrow \tau (f (v_2 (x))) \).
4. Functions: \( v_1 \rightarrow v_2 = \text{fn } f : \tau \rightarrow \sigma_2 \rightarrow \text{fn } x : \tau \rightarrow \tau (f (v_2 (x))) \).

Subsumption Elimination

Subsumption inserts coercion function:

\[
\frac{\Gamma \vdash e : \sigma \rightarrow v}{\Gamma \vdash e : \sigma \leq_{\tau} \tau \rightarrow v (v (e))}
\]

The remaining rules simply compose the translations. For example,

\[
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \rightarrow \tau_2 \rightarrow \tau \rightarrow v}{\Gamma \vdash e (e_1 (e_2)) : \tau \rightarrow v (v (e_1 (e_2)))}
\]

Coercion Interpretation of Subtyping

1. If \( \sigma \leq_{\tau} \tau \rightarrow v \), then \( \Gamma \vdash v : \sigma_{\tau} \rightarrow \tau \).
2. If \( \sigma \leq_{\tau} \tau \rightarrow v_1 \) and \( \sigma \leq_{\tau} \tau \rightarrow v_2 \), then \( \Gamma \vdash v_1 = v_2 : \sigma_{\tau} \rightarrow \tau \).

That is, any two coercions from \( \tau \) to \( \tau' \) are equivalent.

Subsumption Elimination

1. If \( \Gamma \vdash e : \tau \rightarrow e' \), then \( \Gamma \vdash e' : \tau \).
2. If \( \Gamma \vdash e : \tau \rightarrow e_1 \) and \( \Gamma \vdash e : \tau \rightarrow e_2 \), then \( \Gamma \vdash e_1 = e_2 : \tau \).

The proof of the first part is a straightforward induction on derivations. The second is more difficult. Both are omitted here.
Coercion Interpretation

Where else might this method apply?

- Tuple subtyping: truncate to supertype.
- Record subtyping: drop "extra" fields.

Record Subtyping

The width subtyping axiom states that “fatter” records are a subtype of “skinnier” ones:

\[ m \geq n \]

\[ \{l_1: \tau_1, \ldots, l_m: \tau_m\} \subseteq \{l_1: \tau_1, \ldots, l_n: \tau_n\} \]

In other words, a wider record may be provided where a narrower record with the same fields is required.

Dynamic Semantics of Records

In the presence of subtyping the type does not reveal the position of the fields!

- Fields might have been dropped anywhere.
- Can always “weaken” the type to hide additional fields.

Coercion Interpretation

Not necessary for tuples, but useful for records.

- Tuples: projection operations typically are not affected by extra components at the end.
- Records: efficient field selection relies on pre-computing position of each field.

Dynamic Semantics of Records

This suggests that we must search for the lth field on each access.

- Logarithmic in the number of fields using a balanced tree.
- Non-trivial constant factors.

Can we avoid run-time search?
Mutable Records

A compromise solution:

- Represent records as a pair:
  - A dictionary (view, dope vector, or access vector) that determines the position of the \(i\)th field.
  - A data array that records the contents of the fields.

- Coercion truncates the dictionary, but not the data array so that sharing is preserved.

Coercion Interpretation

The coercions are defined as follows:

\[
\begin{align*}
\text{drop}_{m,n,l,\sigma} & \quad = \quad \text{fn} \; x : \{l_1: \sigma_1, \ldots, l_m: \sigma_m\} \in \{l_1: x.l_1, \ldots, l_n: x.l_n\} \quad \text{end} \\
\text{copy}_{m,n,\sigma,v} & \quad = \quad \text{fn} \; x : \{l_1: \sigma_1, \ldots, l_n: \sigma_n\} \in \{l_1: v_1(x.l_1), \ldots, l_n: v_n(x.l_n)\} \quad \text{end}
\end{align*}
\]

Coercion to the rescue!

- If records are immutable, we can coerce records by copying.
- Type now reveals the position of each field.

But only if record fields are immutable!

- Otherwise sharing is lost by copying.

Avoiding Search

Coercions to the rescue!

- If records are immutable, we can coerce records by copying.
- Type now reveals the position of each field.

But only if record fields are immutable!

- Otherwise sharing is lost by copying.

What if records are mutable?

- Coercion by copying is precluded.
- Must retain "hidden" fields.

All are issues of aliasing, multiple active views of the same mutable data structure.

Dictionaries

Represent \(\{l_1:v_1, \ldots, l_n:v_n\}\) as a pair

\[
\{l_1:j_1, \ldots, l_n:j_n\}, v_1 \cdots v_n
\]

where, initially, \(j_i = i\) for \(1 \leq i \leq n\).

Translate \(v.l\) into

\[
\text{snd}(v).\text{fst}(v).l
\]

which fetches the index, and then the value, of field \(l\).

(This has to be a primitive!)
**Dictionaries**

Coercion copies dictionary, but leaves data array intact.

- Maintains constant-time access to record components.
- Preserves sharing of (mutable) data.

Specifically, if $\sigma = \{l_1:s_1, \ldots, l_n:s_n\}$ and $\tau = \{l_1:int, \ldots, l_n:int\}$, then

$$\text{drop}_{m,n,l,\sigma} = \text{fn } x \text{ in } (\text{drop}_{m,n,l,\sigma}(\text{fst}(x)), \text{snd}(x)) \text{ end.}$$

The dictionary is coerced, but the data is left intact.

**Summary**

Coercions support efficient subtyping.

- Avoids run-time type checks.
- Applicable only indirectly to the mutable case.

**Dictionaries**

Similar issues arise in OOP:

- The dictionary is called the _vtable_.
- The record type is determined by the class (defines methods, fields).
- Coercion is often limited to implicit truncation (position matters).