Programming Languages
Semantics of Subtyping
Robert Harper
Spring, 1999

Semantics of Subtyping

Subtyping is an important concept that deserves careful study on its own.

We will consider the extension of MinML with subtyping in order to expose these core issues:

1. Operational semantics of subtyping.
2. Type safety in the presence of subtyping.
3. Type checking with subtypes.

MinML with Subtyping

The extension of MinML with subtyping consists of:

1. An inductive definition of the subtyping relation $\tau <: \tau'$.
2. The addition of the **subsumption** rule:

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

The subsumption rule admits **code re-use** based on subtyping: a value of type τ may be provided whenever a value of type τ' is required, provided that $\tau <: \tau'$.

Subtyping Relation

A **subtyping relation** is defined by

1. One or more **axioms**, or fundamental principles of subtyping.
2. The rules of **reflexivity** and **transitivity** of subtyping:

$$\frac{}{\tau <: \tau} \qquad \frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}$$

3. Rules specifying the **interaction** between subtyping and the other type constructors of the language.

Some Subtyping Axioms

The subtyping relation is generated by one or more basic principles of subtyping, including:

1. **Primitive subtyping**, such as `int <: float`.
2. **Tuple subtyping**: if $m < n$, then $\tau_1 * \cdots * \tau_n$ <: $\tau_1 * \cdots * \tau_m$.
3. **Record subtyping**: if $m < n$, then $\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$ <: $\{l_1{:}\tau_1, \ldots l_m{:}\tau$
4. **Object subtyping**: if $m < n$, then `Obj`$\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$ <: `Obj`$\{l_1{:}\tau_1, \ldots l_m{:}\tau_m\}$.

Tuple, Record, and Object Types (Preview)

The **tuple type** $\tau_1 * \cdots * \tau_n$ is the type of n-tuples <$e_1, \ldots, e_n$>, where $e_i : \tau_i$. The ith component is accessed by the projection $e.i$ of type τ_i, where $e : \tau_1 * \cdots * \tau_n$ and $1 \leq i \leq n$.

The **record type** $\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$ is the type of records $\{l_1{=}e_1, \ldots, l_n{=}e_n\}$ where $e_i : \tau_i$. The component labelled l is accessed by the projection $e.l : \tau_i$, where $l = l_i$. **The order of fields in a record is immaterial.**

The **object type** `Obj`$\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$ is the type of objects `Obj`$\{l_1{=}e_1, \ldots,$ with methods named $l_1, \ldots, l_n$. We'll discuss these in more detail later.

Some Interaction Principles

The subtyping relation includes rules specifying the interaction between type constructors and subtyping such as:

1. **Depth subtyping**, such as
$$\frac{\tau_1 \texttt{ <: } \tau'_1 \quad \cdots \quad \tau_n \texttt{ <: } \tau'_n}{\tau_1 * \cdots * \tau_n \texttt{ <: } \tau'_1 * \cdots * \tau'_n}$$
2. **Co- and contra-variance** for functions:
$$\frac{\tau'_1 \texttt{ <: } \tau_1 \quad \tau_2 \texttt{ <: } \tau'_2}{\tau_1 \to \tau_2 \texttt{ <: } \tau'_1 \to \tau'_2}$$
Note the reversal of ordering in the domain!

Why Depth Subtyping?

An informal argument may be made, based on subsumption and the primitive operations.

If $e : \tau_1 * \cdots * \tau_n$, then $e.i : \tau_i$. If τ_i <: τ'_i, then $e.i : \tau'_i$, by subsumption.

Note that we must consider all the ways that values of the type might be used!

Why Co- and Contra-Variance?

Why is the subtype order **preserved** in the range of function types?

If $f : \tau_1 \rightarrow \tau_2$ and τ_2 <: τ_2', then $f : \tau_1 \rightarrow \tau_2'$. Why? Whenever f is applied to $e : \tau_1$, the result is of type τ_2. But then by subsumption, it is also of type τ_2'.

The function type is said to be **covariant** in the range type.

Why Co- and Contra-Variance?

Why is the subtype order **reversed** for the domain of function types?

If $f : \tau_1' \rightarrow \tau_2$ and τ_1 <: τ_1', then $f : \tau_1 \rightarrow \tau_2$. Why? If $e : \tau_1$, then, by subsumption, $e : \tau_1'$, and hence `apply(`f`,`e`)` has type τ_2.

The function type is said to be **contravariant** in the domain type.

Case Study: Primitive Subtyping

We'll begin our study of subtyping by considering an extension of MinML with

1. Primitive types `int` and `float` with associated arithmetic operations.
2. Subtyping axiom: `int <: float`.
3. Co- and contra-variant subtyping for functions.
4. Subsumption rule.

Failure of Syntax-Directedness

The typing rules are **not** any longer syntax-directed! A given expression can be type-checked in several different ways.

Suppose that f : `float`$\rightarrow$`int` and e : `int`. Then we may either `apply(`f`,`e`)` : `float` in two ways:

1. **Demote** the function: Since `float`$\rightarrow$`int` `<:` `int`$\rightarrow$`float`, we have that f : `int`$\rightarrow$`float` by subsumption. Hence `apply(`f`,`e`)` : `float`.
2. **Promote** the argument: Since e : `int`, then e : `float` by subsumption. Therefore `apply(`f`,`e`)` : `int`, and hence `apply(`f`,`e`)` : `float`, by subsumption.

Restricted Reflexivity

It is enough to state reflexivity only for base types.

That is, if we replace the general reflexivity rule by these two rules

$$\frac{}{\texttt{int} \mathrel{\texttt{<:}} \texttt{int}} \qquad \frac{}{\texttt{float} \mathrel{\texttt{<:}} \texttt{float}}$$

then the resulting relation is still reflexive.

The proof is a simple exercise, by induction on derivations. **Do it!**

Admissibility of Transitivity

The rule of transitivity of subtyping is **admissible**, meaning that it may be omitted without changing the subtyping relation.

More precisely, let $\texttt{<:}^-$ be the subtype relation obtained by omitting the rule of transitivity.

Clearly, if $\tau \mathrel{\texttt{<:}^-} \tau'$, then $\tau \mathrel{\texttt{<:}} \tau'$.

Admissibility of Transitivity

We are to show that if $\tau \mathrel{\texttt{<:}} \tau'$, then $\tau \mathrel{\texttt{<:}^-} \tau'$. The proof is by induction on the derivation of $\tau \mathrel{\texttt{<:}} \tau'$.

The only interesting case is the rule of transitivity; the remaining cases follow directly by induction.

Inductively, we may assume that $\tau \mathrel{\texttt{<:}^-} \tau'$ and $\tau' \mathrel{\texttt{<:}^-} \tau''$. We are to show that $\tau \mathrel{\texttt{<:}^-} \tau''$.

We proceed by case analysis on the proofs of $\tau \mathrel{\texttt{<:}^-} \tau'$ and $\tau' \mathrel{\texttt{<:}^-} \tau''$, transforming these into a proof of $\tau \mathrel{\texttt{<:}^-} \tau''$.

Admissibility of Transitivity

Case 1:

$$\frac{\dfrac{}{\texttt{int} \mathrel{\texttt{<:}^-} \texttt{float}} \quad \dfrac{\vdots}{\texttt{float} \mathrel{\texttt{<:}^-} \tau}}{\texttt{int} \mathrel{\texttt{<:}} \tau}$$

Lemma If $\texttt{float} \mathrel{\texttt{<:}^-} \tau$, then $\tau = \texttt{float}$.

Therefore, the left-hand sub-derivation is the desired derivation.

Admissibility of Transitivity

Case 2:

$$\frac{\begin{array}{c}\vdots\\ \tau \;\texttt{<:}^{-}\; \texttt{int}\end{array} \qquad \dfrac{}{\texttt{int} \;\texttt{<:}^{-}\; \texttt{float}}}{\tau \;\texttt{<:}\; \texttt{float}}$$

Lemma If $\tau \;\texttt{<:}^{-}\; \texttt{int}$, then $\tau = \texttt{int}$.

Therefore the right-hand sub-derivation is the desired derivation.

Admissibility of Transitivity

Case 3:

$$\frac{\begin{array}{c}\vdots\\ \tau_1 \rightarrow \tau_2 \;\texttt{<:}^{-}\; \tau'\end{array} \qquad \begin{array}{c}\vdots\\ \tau' \;\texttt{<:}^{-}\; \tau_1'' \rightarrow \tau_2''\end{array}}{\tau_1 \rightarrow \tau_2 \;\texttt{<:}\; \tau_1'' \rightarrow \tau_2''}$$

Lemma If $\tau_1 \rightarrow \tau_2 \;\texttt{<:}^{-}\; \tau' \;\texttt{<:}^{-}\; \tau_1'' \rightarrow \tau_2''$, then $\tau' = \tau_1' \rightarrow \tau_2'$, with $\tau_1'' \;\texttt{<:}^{-}\; \tau_1' \;\texttt{<:}^{-}\; \tau_1$ and $\tau_2 \;\texttt{<:}^{-}\; \tau_2' \;\texttt{<:}^{-}\; \tau_2''$.

Admissibility of Transitivity

Therefore we have

$$\frac{\dfrac{\begin{array}{c}\vdots\\ \tau_1' \;\texttt{<:}^{-}\; \tau_1\end{array} \quad \begin{array}{c}\vdots\\ \tau_2 \;\texttt{<:}^{-}\; \tau_2'\end{array}}{\tau_1 \rightarrow \tau_2 \;\texttt{<:}^{-}\; \tau_1' \rightarrow \tau_2'} \qquad \dfrac{\begin{array}{c}\vdots\\ \tau_1'' \;\texttt{<:}^{-}\; \tau_1'\end{array} \quad \begin{array}{c}\vdots\\ \tau_2' \;\texttt{<:}^{-}\; \tau_2''\end{array}}{\tau_1' \rightarrow \tau_2' \;\texttt{<:}^{-}\; \tau_1'' \rightarrow \tau_2''}}{\tau_1 \rightarrow \tau_2 \;\texttt{<:}\; \tau_1'' \rightarrow \tau_2''}$$

Hence, by induction we have

$$\frac{\begin{array}{c}\vdots\\ \tau_1'' \;\texttt{<:}^{-}\; \tau_1\end{array} \quad \begin{array}{c}\vdots\\ \tau_2 \;\texttt{<:}^{-}\; \tau_2''\end{array}}{\tau_1 \rightarrow \tau_2 \;\texttt{<:}^{-}\; \tau_1'' \rightarrow \tau_2''}$$

Admissibility of Transitivity

We also have to consider instances of reflexivity, which are easily handled:

$$\frac{\dfrac{}{\texttt{int} \;\texttt{<:}^{-}\; \texttt{int}} \quad \dfrac{}{\texttt{int} \;\texttt{<:}^{-}\; \texttt{int}}}{\texttt{int} \;\texttt{<:}\; \texttt{int}} \qquad \frac{\dfrac{}{\texttt{float} \;\texttt{<:}^{-}\; \texttt{float}} \quad \dfrac{}{\texttt{float} \;\texttt{<:}^{-}\; \texttt{float}}}{\texttt{float} \;\texttt{<:}\; \texttt{float}}$$

$$\frac{\dfrac{}{\texttt{int} \;\texttt{<:}^{-}\; \texttt{float}} \quad \dfrac{}{\texttt{float} \;\texttt{<:}^{-}\; \texttt{float}}}{\texttt{int} \;\texttt{<:}\; \texttt{float}} \qquad \frac{\dfrac{}{\texttt{int} \;\texttt{<:}^{-}\; \texttt{int}} \quad \dfrac{}{\texttt{int} \;\texttt{<:}^{-}\; \texttt{float}}}{\texttt{int} \;\texttt{<:}\; \texttt{float}}$$

Decidability of Subtyping

These properties imply that the subtyping relation may be defined by the following rules:

$$\frac{}{\texttt{int} \mathrel{\texttt{<:}} \texttt{float}} \qquad \frac{}{\texttt{int} \mathrel{\texttt{<:}} \texttt{int}} \qquad \frac{}{\texttt{float} \mathrel{\texttt{<:}} \texttt{float}}$$

$$\frac{\tau_1' \mathrel{\texttt{<:}} \tau_1 \quad \tau_2 \mathrel{\texttt{<:}} \tau_2'}{\tau_1 \rightarrow \tau_2 \mathrel{\texttt{<:}} \tau_1' \rightarrow \tau_2'}$$

Decidability of Subtyping

To decide whether or not $\tau \mathrel{\texttt{<:}} \tau'$:

1. If $\tau = \tau'$, say "yes".
2. If $\tau = \texttt{int}$ and $\tau' = \texttt{float}$, say "yes".
3. If $\tau = \tau_1 \rightarrow \tau_2$, $\tau' = \tau_1' \rightarrow \tau_2'$, then recursively check whether $\tau_1' \mathrel{\texttt{<:}} \tau_1$ and $\tau_2 \mathrel{\texttt{<:}} \tau_2'$, saying "yes" iff both say "yes".
4. Otherwise say "no".

Decidability of Subtyping

Theorem This algorithm yields "yes" or "no" according to whether or not $\tau \mathrel{\texttt{<:}} \tau'$.

Admissibility of transitivity is essential for the proof!

(Otherwise the algorithm would have to "guess" a mediating type and attempt to use transitivity.)

Decidability of Type Checking

How may we determine whether or not $\Gamma \vdash e : \tau$?

Previously, we relied on syntax-directedness: attempt to construct a typing derivation for e by induction on its structure.

1. There is at most one derivation.
2. Every expression has at most one type (in a given context).

Decidability of Type Checking

This method no longer works, because it is not clear when to apply subsumption.

1. The syntax of e no longer determines the typing derivation.

2. The type of e is no longer unique, even for a fixed context.

Decidability of Type Checking

Modified method:

1. Define a **normal form** for typing derivations in which subsumption is used only in a limited way.

2. To type check, search for a normal form derivation, essentially as before.

Normal-Form Typing Derivations

Idea: check subtyping only at function applications to determine whether the type of the argument is a subtype of the domain of the function. Otherwise expressions keep their "natural" type.

This idea is captured by the **type synthesis** relation $\Gamma \vdash e \Rightarrow \tau$, inductively defined by the following rules:

$$\frac{}{\Gamma \vdash x \Rightarrow \Gamma(x)}$$

$$\frac{\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash e \Rightarrow \tau_2}{\Gamma \vdash \mathbf{fun}\, f\ (x{:}\tau_1){:}\tau_2\ \mathbf{is}\, e\ \mathbf{end} \Rightarrow \tau_1{\rightarrow}\tau_2}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2{\rightarrow}\tau \quad \Gamma \vdash e_2 \Rightarrow \tau_2' \quad \tau_2' \texttt{<:}\ \tau_2}{\Gamma \vdash \texttt{apply}(e_1\texttt{,}e_2) \Rightarrow \tau}$$

(Rules for constants and primitive operations omitted.)

Normal-Form Typing Derivations

Theorem

1. (Soundness) If $\Gamma \vdash e \Rightarrow \tau$, then $\Gamma \vdash e : \tau$.

2. (Completeness) If $\Gamma \vdash e : \tau$, then there exists τ' such that $\Gamma \vdash e \Rightarrow \tau'$ and $\tau' \texttt{<:}\ \tau$.

The first part is proved by a straightforward induction on type synthesis derivations.

The second is proved by induction on typing derivations. The two interesting cases are the subsumption rule and the application rule.

Normal-Form Typing Derivations

Suppose that we derive $\Gamma \vdash e : \tau$ from $\Gamma \vdash e : \tau'$ and $\tau' <: \tau$ by the subsumption rule.

By induction we have that $\Gamma \vdash e \Rightarrow \tau''$ with $\tau'' <: \tau'$.

By transitivity of subtyping, $\tau'' <: \tau$.

Hence we have $\Gamma \vdash e \Rightarrow \tau''$ with $\tau'' <: \tau$, as desired.

Normal-Form Typing Derivations

Suppose that we derive $\Gamma \vdash \texttt{apply}(e_1,e_2) : \tau$ from

1. $\Gamma \vdash e_1 : \tau_2 \rightarrow \tau$
2. $\Gamma \vdash e_2 : \tau_2$

by the application rule.

Normal-Form Typing Derivations

By induction we have

1. $\Gamma \vdash e_1 \Rightarrow \tau_1'$ with $\tau_1' <: \tau_2 \rightarrow \tau$.
2. $\Gamma \vdash e_2 \Rightarrow \tau_2'$ with $\tau_2' <: \tau_2$.

Normal-Form Typing Derivations

By **Lemma**, $\tau_1' = \tau_2'' \rightarrow \tau''$ with $\tau_2 <: \tau_2''$ and $\tau'' <: \tau$.

By transitivity of subtyping, $\tau_2' <: \tau_2''$.

Hence $\Gamma \vdash \texttt{apply}(e_1,e_2) \Rightarrow \tau''$.

Since $\tau'' <: \tau$, this completes the proof.

Decidability of Typing

Observe that the type synthesis rules are syntax-directed! Therefore we may find the unique (if it exists) τ such that $\Gamma \vdash e \Rightarrow \tau$, given Γ and e.

By the soundness and completeness of type synthesis, to check whether or not $\Gamma \vdash e : \tau$, it suffices to

1. Find τ' such that $\Gamma \vdash e \Rightarrow \tau'$.
2. Check whether or not $\tau' <: \tau$.

Summary

1. Subtyping is a pre-order on types validating the subsumption principle.
2. There are many different choices of subtyping relation.
3. The proof of decidability of type checking for MinML extended with primitive subtyping is typical of how languages with subtyping are handled.

Prospectus

1. Operational semantics and type safety for subtyping.
2. Coercion interpretations.
3. Tuple and record subtyping.
4. Object types and subtyping.

Programming Languages
Semantics of Subtyping II
Robert Harper
Spring, 1999

MinML With `int` and `float`

Constants:

1. n : `int`, where n is an integer.

2. $m.n$: `float`, where m and n are integers.

MinML With `int` and `float`

Operations:

1. $+_{\texttt{int}}(e_1, e_2)$: `int`, if e_1, e_2 : `int`.

2. $+_{\texttt{float}}(e_1, e_2)$: `float`, if e_1, e_2 : `float`.

3. (*etc.*)

Canonical Forms Lemma

Lemma

1. If v : `int`, then $v = n$ for some n.

2. If v : `float`, then **either** $v = n$ for some n, **or** $v = m.n$ for some m and n.

Canonical Forms Lemma

These are proved simultaneously by induction on typing, relying on the following facts to handle subsumption:

1. If τ `<:` `int`, then τ = `int`.

2. If τ `<:` `float`, then either τ = `int` or τ = `float`.

Operational Semantics

The rules for addition are as follows:

1. $+_{\texttt{int}}(m,n) \mapsto p$, where $p = m + n$.

2. (a) $+_{\texttt{float}}(m,n) \mapsto p.0$, where $p = m + n$.

   (b) $+_{\texttt{float}}(m,n_1.n_2) \mapsto p_1.p_2$, where $p_1.p_2 = m.0 + n_1.n_2$.

   (c) $+_{\texttt{float}}(m_1.m_2,n) \mapsto p_1.p_2$, where $p_1.p_2 = m_1.m_2 + n.0$.

   (d) $+_{\texttt{float}}(m_1.m_2,n_1.n_2) \mapsto p_1.p_2$, where $p_1.p_2 = m_1.m_2 + n_1.n_2$.

Type Conversion

The rules for floating point addition correspond to a **run-time type check** and a **run-time conversion** from integer to floating point!

This check and conversion are **required** to ensure type safety: the CF Lemma tells us that the arguments might be integer or floating point constants.

Of course this is quite unrealistic and entirely unnecessary! We can use static type information to insert **coercions** whenever subsumption is used.

Type Conversion

However, there is a technical problem arising from the failure of syntax-directedness of the typing rules.

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**. In general, we must prove that any two ways of inserting coercions yield the same result. This is called **coherence**.

Coercion Interpretation

Augment the subtyping relation with a **coercion** that maps values of the subtype to values of the supertype:

$$\tau \texttt{<:} \tau' \rightsquigarrow v$$

The expression v is a value of type $\tau \to \tau'$.

Augment the typing relation with a **translation** that inserts coercions to eliminate the need for subtyping:

$$\Gamma \vdash e : \tau \rightsquigarrow e'$$

Notation: $\Gamma \vdash^{-} e : \tau$ for type checking in MinML without subsumption (but with `int` and `float` types). Assume there is a primop `to_float` that converts integers to floating point.

Coercion Interpretation of Subtyping

$$\frac{}{\texttt{int} \mathrel{\texttt{<:}} \texttt{float} \leadsto \texttt{to\_float}}$$

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau \leadsto \texttt{id}_\tau} \qquad \frac{\tau \mathrel{\texttt{<:}} \tau' \leadsto v \quad \tau' \mathrel{\texttt{<:}} \tau'' \leadsto v'}{\tau \mathrel{\texttt{<:}} \tau' \leadsto v\,;v'}$$

$$\frac{\tau_1' \mathrel{\texttt{<:}} \tau_1 \leadsto v_1 \quad \tau_2 \mathrel{\texttt{<:}} \tau_2' \leadsto v_2}{\tau_1 \to \tau_2 \mathrel{\texttt{<:}} \tau_1' \to \tau_2' \leadsto v_1 \to v_2}$$

Coercion Interpretation of Subtyping

The coercions are defined in terms of the following functions:

1. $\texttt{id}_\tau = \texttt{fn}\,x{:}\tau\,\texttt{in}\,x\,\texttt{end}$.

2. $v\,;v' = \texttt{fn}\,x{:}\tau\,\texttt{in}\,v'(v(x))\,\texttt{end}$.

3. $v_1 \to v_2 = \texttt{fn}\,f{:}\tau_1 \to \tau_2\,\texttt{in}\,\texttt{fn}\,x{:}\tau_1'\,\texttt{in}\,v_2(f(v_1(x)))\,\texttt{end}\,\texttt{end}$.

Coercion Interpretation of Subtyping

Theorem

1. If $\tau \mathrel{\texttt{<:}} \tau' \leadsto v$, then $\vdash^- v : \tau \to \tau'$.

2. If $\tau \mathrel{\texttt{<:}} \tau' \leadsto v_1$ and $\tau \mathrel{\texttt{<:}} \tau' \leadsto v_2$, then $\vdash^- v_1 \cong v_2 : \tau \to \tau'$.

That is, any two coercions from τ to τ' are equivalent (in the sense of Leibniz equality).

Coercion Interpretation of Subtyping

The proof relies on these equations:

1. Composition is associative with `id` as left- and right-unit element.

2. $\texttt{id} \to \texttt{id} \cong \texttt{id}$.

3. $v_1 \to v_2\,;v_1' \to v_2' \cong v_1'\,;v_1 \to v_2\,;v_2'$.

This is the **computational content** of the proof of admissibility of transitivity and reflexivity (at other than base type)! Once these have been eliminated, there is only one way to establish the subtype relation.

Subsumption Elimination

Key rule:

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e' \quad \tau <: \tau' \rightsquigarrow v}{\Gamma \vdash e : \tau' \rightsquigarrow v(e')}$$

The remaining rules simply compose the translations. For example,

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash e_1(e_2) : \tau \rightsquigarrow e_1'(e_2')}$$

Subsumption Elimination

Theorem

1. If $\Gamma \vdash e : \tau \rightsquigarrow e'$, then $\Gamma \vdash^{-} e' : \tau$.
2. If $\Gamma \vdash e : \tau \rightsquigarrow e_1$ and $\Gamma \vdash e : \tau \rightsquigarrow e_2$, then $\Gamma \vdash^{-} e_1 \cong e_2 : \tau$.
3. If $e : \mathtt{int} \rightsquigarrow e'$ is a complete program, then $e \Downarrow n$ iff $e' \Downarrow n$.

The proof of the first part is a straightforward induction on derivations. The second and third are more difficult and are omitted here.

Coercion Interpretation

The foregoing demonstrates that run-time checks and conversions may be avoided by inserting coercions during type checking.

The coherence property ensures that different type checkers will yield the same result; the behavior of the program is independent of the manner in which coercions are inserted.

The correctness property states that the ultimate behavior of complete programs is not affected by the insertion of coercions at compile time.

Type-Directed Compilation

This setup is an example of **type-directed compilation** between **typed intermediate languages**.

The coercion intepretation may be viewed as a compiler transformation that relies on the type, and not just the form, of the term to determine its translation.

The translation preserves typability: if the program is type-correct, so is its translation, but in a subtyping-free version of the language.

Type-Directed Compilation

Benefits of TDC:

1. The compiler can take advantage of type information during translation.
2. The compiler is "self-checking". By verifying the type correctness of the translation, bugs in the compiler can be caught early.
3. The object code can be typed using **typed assembly language**; the recipient can type check to ensure safety.

Type-Directed Compilation

Drawbacks of TDC:

1. The intermediate languages must be typed.
2. Maintaining type information presents significant engineering problems.
3. Old ideas must be re-thought in the typed setting.

Record Subtyping

A **record type** $\{l_1 : \tau_1, \ldots l_n : \tau_n\}$ is the type of record expressions $\{l_1 = e_1, \ldots, l_n = e_n\}$, where $e_i : \tau_i$. The order of fields in a record is immaterial. The component labelled l is accessed by the projection $e.l : \tau_i$, where $l = l_i$.

The **width** subtyping axiom states that "fatter" records are a subtype of "skinnier" ones:

$$\frac{m \geq n}{\{l_1 : \tau_1, \ldots l_m : \tau_m\} \mathrel{<:} \{l_1 : \tau_1, \ldots l_n : \tau_n\}}$$

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

Record Subtyping

The **depth** subtyping rule states that subtyping is compatible with record formation:

$$\frac{\tau_1 \mathrel{<:} \tau_1' \quad \ldots \quad \tau_n \mathrel{<:} \tau_n'}{\{l_1 : \tau_1, \ldots l_n : \tau_n\} \mathrel{<:} \{l_1 : \tau_1', \ldots l_n : \tau_n'\}}$$

In other words, if corresponding components stand in the subtype relation, then so do the record types.

Record Subtyping

To ensure that transitivity of subtyping is admissible, we **consolidate** the width and depth subtyping rules into one, as follows:

$$\frac{\tau_1 <: \tau_1' \quad \ldots \quad \tau_n <: \tau_n' \quad (n \leq m)}{\{l_1 : \tau_1, \ldots l_m : \tau_m\} <: \{l_1 : \tau_1', \ldots l_n : \tau_n'\}}$$

That is, to obtain a supertype of a record type, we may

1. Drop fields.

2. Promote the types of the remaining fields.

Record Subtyping

Lemma Reflexivity may be limited to base types.
Proof Use depth subtyping for records, co- and contra-variance for function types.

Lemma Transitivity is admissible.
Proof As before, using the following facts:

1. If $\tau <: \{l_1 : \tau_1', \ldots l_n : \tau_n'\}$, then $\tau = \{l_1 : \tau_1, \ldots l_m : \tau_m\}$, with $m \geq n$ and $\tau_i <: \tau_i'$ for $1 \leq i \leq n$.

2. If $\{l_1 : \tau_1, \ldots l_m : \tau_m\} <: \tau'$, then $\tau' = \{l_1 : \tau_1', \ldots l_n : \tau_n'\}$ with $m \geq n$ and $\tau_i <: \tau_i'$ for $1 \leq i \leq n$.

Decidability of Subtyping

To check whether $\tau <: \tau'$,

1. If $\tau = \tau'$, yield "yes".

2. If $\tau = \{l_1 : \tau_1, \ldots l_m : \tau_m\}$ and $\tau' = \{l_1 : \tau_1', \ldots l_n : \tau_n'\}$, then check that $m \geq n$ and, **for some re-ordering of fields**, recursively check that $\tau_i <: \tau_i'$ for $1 \leq i \leq n$.

3. If $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$, then recursively check that $\tau_1' <: \tau_1$ and $\tau_2 <: \tau_2'$.

Operational Semantics of Record Subtyping

The primitive instruction for record field projection is given by

$$\{l_1 = v_1, \ldots, l_n = v_n\}.l \mapsto v_i$$

where $l = l_i$ for some $1 \leq i \leq n$.

This rule requires **search** at run-time to find the field labelled l!

1. Order of fields is immaterial.

2. By subtyping we may "forget" any field.

Therefore, projection takes time $O(n)$, where n is the number of fields, in the worst case!

Avoiding Search

How can we avoid searching at run-time?

1. **Eliminate order-independence**. Then we may only drop fields from the **end** of the record using subtyping. The projection operation can determine the position of the lth field just from the (apparent) type of the record.

2. **Use a coercion interpretation**. Whenever record subsumption is used, create a **new** record with exactly the specified fields. This is OK provided that there is no assignment!

Coercion Interpretation

$$\frac{m > n}{\{l_1{:}\tau_1, \ldots l_m{:}\tau_m\} \texttt{ <: } \{l_1{:}\tau_1, \ldots l_n{:}\tau_n\} \leadsto \texttt{drop}_{m,n,l,\tau}}$$

$$\frac{\tau_1 \texttt{ <: } \tau_1' \leadsto v_1 \quad \ldots \quad \tau_n \texttt{ <: } \tau_n' \leadsto v_n}{\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\} \texttt{ <: } \{l_1{:}\tau_1', \ldots l_n{:}\tau_n'\} \leadsto \texttt{copy}_{n,l,\tau,v}}$$

Coercion Interpretation

The coercions are defined as follows:

$$\texttt{drop}_{m,n,l,\tau} \;=\; \texttt{fn}\, x{:}\{l_1{:}\tau_1, \ldots l_m{:}\tau_m\}\, \texttt{in}\, \{l_1\texttt{=}x.l_1, \ldots, l_n\texttt{=}x.l_n\}\, \texttt{end}$$

$$\texttt{copy}_{n,l,\tau,v} \;=\; \texttt{fn}\, x{:}\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}\, \texttt{in}\, \{l_1\texttt{=}v_1(x.l_1), \ldots, l_n\texttt{=}v_n(x.l_n)\}\, \texttt{end}$$

NB: this approach does not work if record fields are mutable (assignable) because the coercion destroys sharing of state!

Coercion Interpretation

Theorem

1. If $\tau \texttt{ <: } \tau' \leadsto v$, then $\vdash^{-} v : \tau \to \tau'$.

2. If $\tau \texttt{ <: } \tau' \leadsto v_1$ and $\tau \texttt{ <: } \tau' \leadsto v_2$, then $\vdash^{-} v_1 \cong v_2 : \tau \to \tau'$.

Theorem If $\vdash e : \texttt{int} \leadsto e'$, then $e \Downarrow n$ iff $e' \Downarrow n$.

Objects

Simplified object model:

1. Methods only, no fields.

2. Self-reference (`this`).

3. No mutation.

This is enough to illustrate the main points; the model can be enriched to handle (mutable) fields.

Object Types

The **object type** $\texttt{Obj}\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$ is the type of objects with methods $l_1, \ldots, l_n$ of types $\tau_1, \ldots, \tau_n$, respectively.

If $e : \texttt{Obj}\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$, then $e.l : \tau_i$, where $l = l_i$ for some $1 \le i \le n$.

The order of methods in an object type is **insignificant**, as with records.

Object Types

Object expressions have the form $\texttt{Obj}\{l_1\texttt{=}e_1, \ldots, l_n\texttt{=}e_n\}$, with typing rule

$$\frac{\Gamma, \texttt{this}{:}\tau \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma, \texttt{this}{:}\tau \vdash e_n : \tau_n}{\Gamma \vdash \texttt{Obj}\{l_1\texttt{=}e_1, \ldots, l_n\texttt{=}e_n\} : \tau}$$

where $\tau = \texttt{Obj}\{l_1{:}\tau_1, \ldots l_n{:}\tau_n\}$.

Note the similarity with the typing rule for recursive functions!

1. Multiple entry points, rather than just one for recursive functions.

2. The variable `this` refers to the object itself. Mutual recursion is achieved by selecting methods from `this`.

Object Example (in Pseudo-ML)

An object type:

```
type Point =
  object
    method getx : unit -> int
    method gety : unit -> int
    method move : int * int -> unit
    method bump : unit -> unit
  end
```

Object Example (in Pseudo-ML)

```
fun make_point (x:int, y:int):Point =
  let val x = ref x
      val y = ref y
  in  object
        method getx () = !x
        method gety () = !y
        method move (dx, dy) = (x := !x + dx; y := !y + dy)
        method bump () = this.move (1, 1)
      end
  end
val p : Point = make_point (0, 0)
```


Operational Semantics

Method dispatch consists of

1. Searching for the appropriate method.

2. Invoking it with `this` bound to the object itself.

Formally,

$$v.l \mapsto [v/\mathtt{this}]v_i,$$

where $l = l_i$ and

$$v = \mathtt{Obj}\{l_1{=}v_1, \ldots, l_n{=}v_n\}$$

Operational Semantics

Compare: mutually-recursive functions and objects.

```
fun even (n:int):bool is n=0 orelse odd (n-1) end
and odd  (n:int):bool is n<>0 andalso even (n-1) end

object
  method even(n:int):bool is n=0 orelse this.odd (n-1) end
  method odd (n:int):bool is n<>0 andalso this.even (n-1) end
end
```

Objects really are generalized, recursive functions! In other words, object-based programming is a form of functional programming, and vice versa!

Object Subtyping

Object types support both **width** and **depth** subtyping:

$$\frac{\tau_1 \mathrel{<:} \tau_1' \quad \ldots \quad \tau_n \mathrel{<:} \tau_n' \quad (n \le m)}{\mathtt{Obj}\{l_1{:}\tau_1, \ldots l_m{:}\tau_m\} \mathrel{<:} \mathtt{Obj}\{l_1{:}\tau_1', \ldots l_n{:}\tau_n'\}}$$

The types of methods have the form

$$\tau_{args} \rightarrow \tau_{result},$$

where τ_{args} is the type of the arguments and τ_{result} is the type of the result of the method.

Object Subtyping

The **depth** subtyping principle implies that the type of a method in the subtype must satisfy these requirements:

1. Its result type must be a **subtype** of the result type of the corresponding method in the supertype.

2. Its argument type must be a **supertype** of the arguments type of the corresponding method in the supertype.

This is just co- and contra-variance again, but in the guise of methods, rather than functions!

Object Subtyping

If subclasses yield subtypes (as in Java), then this imposes the following restrictions on method override:

1. The argument type of the method in the subclass must be a **supertype** of the argument type in the superclass.

2. The result type of the method in the subclass must be a **subtype** of the result type in the superclass.

Object Subtyping

This provides another angle on why `This` types violate the subclass-as-subtype principle when used to implement binary methods.

If `This` occurs as the **argument** of a method in the subclass, then subtyping **fails** because of contra-variance!

- `This` stands for the instance type of the subclass.

- The subclass-as-subtype principle implies that the instance type of the subclass should be a **subtype** of the instance type of the superclass.

- But by contravariance, it is required to be a **supertype**!

Avoiding Search

The operational semantics of method dispatch requires **search** for the method. How can this be avoided?

1. Restrict subtyping to drop components only from the end of the object. This significantly reduces flexibility.

2. The coercion interpretation used for records is not available! **You cannot drop methods, even if they are not apparent in the object's type!**

Avoiding Search

Why is a coercion interpretation unavailable for objects?

- The supertype may specify a method named l, but not l'.
- The underlying code for method l might call method l'.
- Therefore we cannot drop l' from the object itself, otherwise invocation of l fails.

Avoiding Search

```
type Point = object
  method getx : unit -> int
  method gety : unit -> int
  method move : int * int -> unit
  method bump : unit -> unit
end
type Point' = object
  method getx : unit -> int
  method gety : unit -> int
  method bump : unit -> unit
end
val p : Point = ... as above ...
val q : Point' = p;
... q.bump ...        (* invokes q.move = p.move! *)
```


Dictionaries

Dictionaries: replace search by an additional level of indirection on method dispatch.

- Represent an object as a **record** mapping labels to **offsets** together with ...
- ... an **bare object** mapping **offsets** to methods.

Subtyping coerces the record component (as before), but the underlying object remains intact.

Dictionaries

However, dictionaries require that we distinguish between

- **External** method calls, which are initiated by clients of the object.
- **Internal** method calls, which are initiated by the methods of the object itself via `this`.

Roughly speaking, internal method calls need not (in fact, **cannot**) make use of the dictionary since it is tied to the **apparent** type of the object, not its **true** type.

Dictionaries

To invoke a method **externally**,

1. Select immediately the appropriate record slot to find the offset into the object.

2. Select the method based on the offset just determined.

3. Pass the entire object (record + bare object) to the method as `this`.

Dictionaries

To invoke a method **internally**,

1. The calls must have the form $\texttt{this}.i$, where i is the **offset** of the method in the object.

2. The "bare" object is substituted for `this` in the method body.

Once you're "inside" the object, you can avoid the overhead of method lookup since the offsets of the other methods are determined when the object is created.

Dictionaries

Objects with dictionaries have the form

$$\texttt{Obj}\{v_1, \ldots, v_n\}\,\texttt{with}\,\{l_1\texttt{=}k_1, \ldots, l_m\texttt{=}k_m\}$$

where $m \leq n$, each v_i is a method body, and $1 \leq k_j \leq n$ for each $1 \leq i \leq m$.

External method dispatch:

$$(\texttt{Obj}\{v_1, \ldots, v_n\}\,\texttt{with}\,\{l_1\texttt{=}k_1, \ldots, l_m\texttt{=}k_m\}).l_i \mapsto [v/\texttt{this}]v_{k_i},$$

where $1 \leq i \leq m$ and $v = \texttt{Obj}\{v_1, \ldots, v_n\}$.

Internal method dispatch:

$$i.v \mapsto [v/\texttt{this}]v_i$$

where $v = \texttt{Obj}\{v_1, \ldots, v_n\}$.

Method Cacheing

Heuristic techniques are also used. **Method cacheing** is one popular technique.

- Perform search to find a method when invoked.

- Cache the offset of the method in a table.

- Use run-time hashing to index the cache.

Empirical evidence shows that this works well in practice.

Summary

Subtyping increases flexibility by enabling convenient code re-use.

But it is not free! Costs can be apportioned based on expected patterns of use and on the semantics of the language.

1. Coercion approaches make uses of subtyping expensive in both space and time (due to copying). But access is relatively cheap.

2. Search approches make subtyping free, at the expense of lookup.

As usual, there's no free lunch.