Advanced Topics in Programming Languages

Spring semester, 2012

Subtyping

Lecturer: Mooly Sagiv *Scribe: Roman Kecher and Anna Itin*

11.1 Lecture Overview

This lecture addresses a fundamental extension: subtyping (sometimes called subtype polymorphism). Unlike the features we have studied up to now, which could be formulated more or less orthogonally to each other, subtyping is a cross-cutting extension, interacting with most other language features in non-trivial ways.

Without subtyping, the rules of the simply typed lambda-calculus can be annoyingly rigid. The type systems insistence that argument types exactly match the domain types of functions will lead the typechecker to reject many programs that, to the programmer, seem obviously well-behaved. For example, recall the typing rule for function application:

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

According to this rule, the well-behaved term

$$(\lambda r : \{x : Nat\}.\ r.x)\ \ \{x = 0, y = 1\}$$

is not typable, since the type of the argument is {x:Nat,y:Nat}, whereas the function accepts {x:Nat}. But, clearly, the function just requires that its argument is a record with a field x; it doesnt care what other fields the argument may or may not have. Moreover, we can see this from the type of the function – we dont need to look at its body to verify that it doesnt use any fields besides x. It is always safe to pass an argument of type {x:Nat,y:Nat} to a function that expects type {x:Nat}.

The goal of subtyping is to refine the typing rules so that they can accept terms like the one above. We accomplish this by formalizing the intuition

that some types are more informative than others: we say that S is a subtype of T, written S <: T, to mean that any term of type S can safely be used in a context where a term of type T is expected. This view of subtyping is often called the principle of safe substitution.

11.2 Previous Lectures Recap

11.2.1 Simple Typed Lambda Calculus

$t ::=$		terms
	x	variable
	$\lambda x : T.t$	abstraction
	$t\ t$	application

$T ::=$		types
	$T \to T$	types of functions

$v ::=$		values
	$\lambda x : T.t$	abstraction values

$\Gamma ::=$		context
	$\emptyset$	empty context
	$\Gamma, x : T$	term variable binding

11.2.2 Type Rules

$$\Gamma \vdash t : T$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\ t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t1\ t2 : T_{12}} \quad \text{(T-APP)}$$

11.2.3 Varieties of Polymorphism

There are at least three different types of polymorphism, listed below:

1. Parametric polymorphism: Parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety. Using parametric polymorphism, a function or a data type can be written generically so that it can handle values identically without depending on their type. This mechanism is very popular, and can be found in many functional languages, such as Haskel or ML.

2. Ad-hoc polymorphism: Ad-hoc polymorphism is a kind of polymorphism in which polymorphic functions can be applied to arguments of different types, because a polymorphic function can denote a number of distinct and potentially heterogeneous implementations depending on the type of argument(s) to which it is applied. It is also known as function overloading or operator overloading. The term "ad hoc" in this context is not intended to be pejorative; it refers simply to the fact that this type of polymorphism is not a fundamental feature of the type system. With this kind of polymorphism, which appears in Haskell and in C, it is usually the compiler that has to decide which code should be invoked in any given situation.

3. Subtype polymorphism: Subtyping or subtype polymorphism is a form of type polymorphism in which a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability, meaning that program constructs, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written S <: T, to mean that any term of type S can be safely used in a context where a term of type T is expected.

In this lecture we shall examine the subtype polymorphism in detail.

11.3 Subtyping

The subtype relation is formalized as a collection of inference rules for deriving statements of the form S <: T, pronounced "S is a subtype of T" (or "T is a supertype of S"). We consider each form of type (function types,

record types, etc.) separately; for each one, we introduce one or more rules formalizing situations when it is safe to allow elements of one type of this form to be used where another is expected. This view of subtyping is often called the principle of safe substitution.

A simpler intuition is to read S <: T as "every value described by S is also described by T", that is, "the elements of S are a subset of the elements of T". The bridge between the typing relation and this subtype relation is provided by adding a new typing rule – the so-called rule of subsumption:

$$\frac{\Gamma \vdash t : S \qquad s <: T}{\Gamma \vdash t : T} \tag{T-SUB}$$

This rule tells us that, if S <: T, then every element t of S is also an element of T.

11.4 Healthiness of Subtypes

Before we get to the rules for particular type constructors, we make two general stipulations: first, that subtyping should be reflexive,

$$S <: S \tag{S-REFL}$$

and second, that it should be transitive:

$$\frac{S <: U \qquad U <: T}{S <: T} \tag{S-TRANS}$$

These rules follow directly from the intuition of safe substitution.

11.5 Record Types

This is an example where subtypes could be very handy.

We define new syntactic forms:

$t ::= \ldots$		Terms:
	$\{l_i = t_i \ ^{i \in 1..n}\}$	record
	$t.l$	projection

$v ::= ...$		Values:
	$\{l_i = v_i \ ^{i \in 1..n}\}$	records
$T ::= ...$		Types:
	$\{l_i : T_i \ ^{i \in 1..n}\}$	record type

New typing rules:

$$\frac{For\ each\ i\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i \ ^{i \in 1..n}\} : \{l_i : T_i \ ^{i \in 1..n}\}} \quad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash t : \{l_i : T_i \ ^{1 \in 1..n}\}}{\Gamma \vdash t.l_j : T_j} \quad \text{(T-Proj)}$$

And new evaluation rules:

$$\{l_i = v_i \ ^{i \in 1..n}\}.l_j \to v_j \quad \text{(E-ProjRCD)}$$

$$\frac{t \to t'}{t.l \to t'.l} \quad \text{(E-Proj)}$$

$$\frac{t_j \to t'_j}{\{l_i = v_i \ ^{i \in 1..j-1}, l_i = t_i \ ^{i \in j..n}\} \to \{l_i = v_i \ ^{i \in 1..j-1}, l_j = t'_j, l_k = t_k \ ^{k \in j+1..n}\}} \quad \text{(E-Tuple)}$$

11.5.1 Motivating Record Example

Consider the aforementioned example:

$$(\lambda r : \{x : Nat\}.\ r.x)\ \ \{x = 0, y = 1\}$$

If we define the subtype relation so that {x:Nat,y:Nat} <: {x:Nat}, then we can use rule T-SUB to derive {x=0,y=1} : {x:Nat}, which is what we need to make this motivating example typecheck. The following rules should assist in achieving just that.

11.5.2 Record Width Subtyping

As we have just seen, we want to consider the type $S = \{k_1 : S_1, ..., k_m : S_m\}$ to be a subtype of $T = \{k_1 : S_1, ..., k_n : S_n\}$ if T has fewer fields than S. In

particular, it is safe to "forget" some fields at the end of a record type. The so-called width subtyping rule captures this intuition:

$$\{l_i : T_i \ ^{i \in 1..n+k}\} <: \{l_i : T_i \ ^{i \in 1..n}\} \qquad \text{(S-RCDWIDTH)}$$

It may seem surprising that the "smaller" type – the subtype – is the one with more fields. The easiest way to understand this is to adopt a more liberal view of record types, regarding a record type {x:Nat} as describing "the set of all records with at least a field x of type Nat". Values like {x=3} and {x=5} are elements of this type, and so are values like {x=3,y=100} and {x=3,a=true,b=true}. Similarly, the record type {x:Nat, y:Nat} describes records with at least the fields x and y, both of type Nat. Values like {x=3,y=100} and {x=3,y=100,z=true} are members of this type, but {x=3} is not, and neither is {x=3,a=true,b=true}. Thus, the set of values belonging to the second type is a proper subset of the set belonging to the first type. A longer record constitutes a more demanding–i.e., more informative–specification, and so describes a smaller set of values.

11.5.3 Record Depth Subtyping

The width subtyping rule applies only to record types where the common fields are identical. It is also safe to allow the types of individual fields to vary, as long as the types of each corresponding field in the two records are in the subtype relation. The depth subtyping rule expresses this intuition:

$$\frac{for\ each\ i\ S_i <: T_i}{\{l_i : S_i \ ^{i \in 1..n}\} <: \{l_i : T_i \ ^{i \in 1..n}\}} \qquad \text{(S-RCDDEPTH)}$$

11.5.4 Record Examples

Example 1

We will show that:

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}$$

.

1. $\{a : Nat, b : Nat\} <: \{a : Nat\}$ (S-RCDWIDTH)
2. $\{m : Nat\} <: \{\}$ (S-RCDWIDTH)

3. $\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}$ (1,2,S-RCDDEPTH)

Example 2

We will show that:

$$\{x : \{a : Nat, b : Nat\}, y : Nat\} <: \{x : \{a : Nat\}, y : Nat\}$$

1. $\{a : Nat, b : Nat\} <: \{a : Nat\}$ (S-RCDWIDTH)
2. $Nat <: Nat$ (S-REF)
3. $\{x : \{a : Nat, b : Nat\}, y : Nat\} <: \{x : \{a : Nat\}, y : Nat\}$ (1,2,S-RCDDEPTH)

Example 3

We will show that:

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}\}$$

1. $\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat, b : Nat\}\}$ (S-RCDWIDTH)
2. $\{a : Nat, b : Nat\} <: \{a : Nat\}$ (S-RCDWIDTH)
3. $\{x : \{a : Nat, b : Nat\}\} <: \{x : \{a : Nat\}\}$ (2,S-RCDEPTH)
4. $\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}\}$ (1,3,S-TRANS)

11.5.5 Record Field Permutation

Our final record subtyping rule arises from the observation that the order of fields in a record does not make any difference to how we can safely use it, since the only thing that we can do with records once we've built them–i.e., projecting their fields–is insensitive to the order of fields.

$$\frac{\{k_j : S_j \ ^{j \in 1..n}\} \textit{ is a permutation of } \{l_i : T_i \ ^{i \in 1..n}\}}{\{k_j : S_j \ ^{j \in 1..n}\} <: \{l_i : T_i \ ^{i \in 1..n}\}} \qquad \text{(S-RCDPERM)}$$

S-RCDPERM can be used in combination with S-RCDWIDTH and S-TRANS to drop fields from anywhere in a record type, not just at the end.

11.5.6 Subtyping In Various Programming Languages

Not every programming language supports all three record subtyping rules:

- Most languages support at least the first two (S-RCDWIDTH, S-RCDEPTH).
- The S-RCDPERM rule depends on the language: sometimes the compiler orders the fields in some order and then it is important to keep that specific correct, and sometimes (like in our case) there is no importance to the order
- Another point worth making is that the S-RCDPERM rule is not antisymmetric.

11.6 Function Types

Since we are working in a higher-order language, where not only numbers and records but also functions can be passed as arguments to other functions, we must also give a subtyping rule for function types–i.e., we must specify under what circumstances it is safe to use a function of one type in a context where a different function type is expected.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-ARROW)}$$

Notice that the sense of the subtype relation is reversed (contravariant) for the argument types in the left-hand premise, while it runs in the same direction (covariant) for the result types as for the function types themselves. The intuition is that, if we have a function f of type $S_1 \to S_2$, then we know that f accepts elements of type S_1; clearly, f will also accept elements of any subtype T_1 of S_1. The type of f also tells us that it returns elements of type S_2; we can also view these results belonging to any supertype T_2 of S_2. That is, any function f of type $S_1 \to S_2$ can also be viewed as having type $T_1 \to T_2$.

An alternative view is that it is safe to allow a function of one type $S_1 \to S_2$ to be used in a context where another type $T_1 \to T_2$ is expected as long as none of the arguments that may be passed to the function in this context will surprise it ($T_1 <: S_1$) and none of the results that it returns will surprise the context ($S_2 <: T_2$).

11.7 Top Type

It is convenient to have a type that is a supertype of every type. We introduce a new type constant Top, plus a rule that makes Top a maximum element of the subtype relation:

$$S <: Top \qquad \text{(S-TOP)}$$

We add this rule in order to handle Object Oriented Languages. For example, that is the type that corresponds to the Object type in Java. The Top type is the "largest" type of all the other types, and it is the most "generic" type – making it fairly useful in some scenarios.

It is also possible to discuss the addition of a "Bottom" type, but it significantly complicates the problem of building a typechecker for the system. The Bottom type is most useful in dealing with exceptions.

11.8 Subtype Relation As A Preorder

In mathematics, especially in order theory, a preorder or quasi-order is a binary relation that is reflexive and transitive. All partial orders and equivalence relations are preorders, but preorders are more general.

The name "preorder" comes from the idea that preorders are 'almost' (partial) orders, but not quite; they're neither anti-symmetric nor symmetric. Because a preorder is a binary relation, the symbol $\leq$ can be used as the notational device for the relation. However, because they are not anti-symmetric, some of the ordinary intuition that a student may have with regards to the symbol $\leq$ may not apply. On the other hand, a pre-order can be used, in a straight-forward fashion, to define a partial order and an equivalence relation

The subtype relation we have defined here is a pre-order on the types: It is reflexive, transitive, and non-anti-simmetric.

11.8.1 More Preorder Examples

- The Natural order: (N,$\leq$).
- $P(N)$ - power set:

$$X \subseteq Y \; IFF \; \forall z \in X \exists t \in Y : z \leq t$$

11.9 Type Verification Example

$$\vdash (\lambda r : \{x : Nat\}.\ r.x)\{x = 0, y = 0\} : T$$

In this example we would like to check if the given program has the type T. In other words , we want to know if there exists an inference rule that allows to accept the example. In class we proved the example using the invertion lemma, but here we will show it inthe "correct" order. First of all we will show that:

$$\vdash \{x = 0, y = 0\} : \{x : Nat\}$$

$$\vdash 0 : Nat$$

Because of the (T-Tuple) rule we get that:

$$\vdash \{x = 0, y = 0\} : \{x : Nat, y : Nat\}$$

But because of the (S-RCDWIDTH) rule it holds that : {x:NAt,y:Nat}¡:{x:Nat} Because pf the two previous and the (T-SUB) rule we get that:

$$\vdash \{x = 0, y = 0\} : \{x : Nat\}$$

No we will define a record of size 1 like this: r={x:Nat} We will define the type of the record T' and we will call the type of the value T, so we get the following expression: r:T' = {x:T=Nat} Now we will use the (T-VAR) rule and we will get that:

$$\vdash r : T' = \{x : T = Nat\}$$

Using the (T-Proj) rule we get that:

$$\vdash r.x : T$$

Using both of them and the rule (T-ABS) we get that:

$$\vdash \lambda r : \{x : Nat\}.\ r.x) : \{x : Nat\} \to T$$

Using this one with the previous expresion that we proved

$$\vdash \{x = 0, y = 0\} : \{x : Nat\}$$

and the (T-APP) rule we get that:

$$\vdash (\lambda r : \{x : Nat\}.\ r.x)\{x = 0, y = 0\} : T$$

That proves that the given programm typechecks correctly and that its type is Nat.

11.10 Properties Of The Given Type System

- Linear time type checking is feasible.
- Type safety:
  - Well typed programs cannot go wrong: no undefined semantics and no runtime checks.
  - Progress: If t is well typed then either t is a value or there exists an evaluation step $t \to t'$.
  - Preservation: If t is well typed and there exists an evaluation step $t \to t'$ then t' is also well typed.
- It is an important (and fairly obvious) observation that the addition of the listed subtyping rules invalidates the uniqueness of types.

11.11 Upcasting: Information Hiding

Upcasting provides the programmer with the ability to move from a specific type to a more general type.

It appears that we do not have to make any specific extension to our system to be able to support upcasts – the ascription rule is enough.

For example, given

$$\vdash t : S$$

and

$$S <: T$$

invoking the (T-SUB) rule we get

$$\Gamma \vdash t : T$$

at this point we are able to use the (T-ASCRIBE) rule

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \ as \ T : T} \qquad \text{(T-ASCRIBE)}$$

to get the final result:

$$\Gamma \vdash t \ as \ T : T$$

.

11.12 Downcasting

Downcasting provides the programmer with the ability to move from a general type to a more specific one, or even to 'cast' between totally unrelated types.

Examples for common uses of downcasting are: when we did upcasting before and now we want to get back to the original type, or in case of Generics or Reflection. Another example would be found in languages that do not support polymorphysm (for instance, when popping from a list that contains objects).

The following "trivial" rule assumes no relation between the type S and the type T:

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

The type checking will have to be deferred to runtime. Therefore, we add another rule to the operational semantics (instead of the E-ASCRIBE rule):

$$\frac{\vdash v : T}{v \ as \ T \to v} \qquad \text{(E-DOWNCAST)}$$

If v is not of type T, the evaluation will get stuck in runtime. It is important to note that the progress is indeed hurt: we might get stuck on things that we were able to typecheck.

Some people do not like this approach. For example, runtime dynamic type checking has been added to the Modula programming language through a specific structure.

$$\frac{\Gamma \vdash t_1 : S \quad \Gamma, x : T \vdash t_2 : U \quad \Gamma \vdash t_3 : U}{\Gamma \vdash if \ t_1 \ in \ T \ then \ x \mapsto t_2 \ else \ t_3 : U} \qquad \text{(T-TYPETEST)}$$

And the computing rules:

$$\frac{\vdash v : T}{if \ v \ in \ T \ then \ x \mapsto t_2 \ else \ t_3 \to [x \to v]t_2} \qquad \text{(E-TYPETESTT)}$$

$$\frac{\nvdash v : T}{if \ v \ in \ T \ then \ x \mapsto t_2 \ else \ t_3 \to t_3} \qquad \text{(E-TYPETESTF)}$$

At this point, the given syntax should remind you of Java's *instanceof* operator.

11.13 Downcasting vs. Polymorphysm

Uses of down-casts are actually quite common in languages like Java. In particular, down-casts support a kind of "poor-man's polymorphism". For example, "collection classes" such as Set and List are monomorphic in Java: instead of providing a type List T (lists containing elements of type T) for every type T, Java provides just List, the type of lists whose elements belong to the maximal type Object. Since Object is a supertype of every other type of objects in Java, this means that we have to use upcasts to add elements, and downcasts to use them.

It seems better to extend the Java type system with real polymorphism, which is both safer and more efficient than the down-cast idiom, requiring no run-time tests. On the other hand, such extensions add significant complexity to an already-large language, interacting with many other features of the language and type system; this fact supports a view that the down-cast idiom offers a reasonable pragmatic compromise between safety and complexity.

Down-casts also play a critical role in Java's facilities for reflection. Using reflection, the programmer can tell the Java run-time system to dynamically load a bytecode file and create an instance of some class that it contains. Clearly, there is no way that the typechecker can statically predict the shape of the class that will be loaded at this point (the bytecode file can be obtained on demand from across the net, for example), so the best it can do is to assign the maximal type Object to the newly created instance. Again, in order to do anything useful, we must downcast the new object to some expected type T, handle the run-time exception that may result if the class provided by the bytecode file does not actually match this type, and then go ahead and use it with type T.

To close the discussion of down-casts, a note about implementation is in order. It seems, from the rules we have given, that including down-casts to a language involves adding all the machinery for typechecking to the runtime system. Worse, since values are typically represented differently at run time than inside the compiler (in particular, functions are compiled into bytecodes or native machine instructions), it appears that we will need to write a different typechecker for calculating the types needed in dynamic checks. To avoid this, real languages combine down-casts with type tags–single-word tags (similar in some ways to ML's datatype constructors and the variant tags discussed earlier) that capture a run-time "residue" of compile-time types and that are sufficient to perform dynamic subtype tests.

11.14 Variants

The subtyping rules for variants are nearly identical to the ones for records; the only difference is that the width rule (S-VARIANTWIDTH) allows new variants to be added, not dropped, when moving from a subtype to a supertype.

The intuition is that a tagged expression $< l = t >$ belongs to a variant type $< l_i : T_i \ ^{i \in 1..n} >$ if its label l is one of the possible labels $\{l_i\}$ listed in the type; adding more labels to this set decreases the information it gives us about its elements.

Another consequence of combining subtyping and variants is that we can drop the annotation from the tagging construct, writing just $< l = t >$ instead of $< l = t > \ as \ < l_i : T_i \ ^{i \in 1..n} >$, and changing the typing rule for tagging so that it assigns $< l_1 = t_1 >$ the precise type $< l_1 : T_1 >$. We can then use subsumption plus (S-VARIANTWIDTH) to obtain any larger variant type.

11.15 Lists

We have seen a number of examples of covariant type constructors (records and variants, as well as function types, on their right-hand sides) and one contravariant constructor (arrow, on the left-hand side). The List constructor is also covariant: if we have a list whose elements have type S_1, and $S_1 <: T_1$, then we can safely regard our list as having elements of type T_1.

$$\frac{S <: T}{ListS <: ListT} \qquad \text{(S-LIST)}$$

11.16 References

Not all type constructors are covariant or contravariant. The Ref constructor, for example, must be taken to be invariant in order to preserve type safety.

$$\frac{S <: T \quad T <: S}{refS <: refT} \qquad \text{(S-REF)}$$

For $RefS$ to be a subtype of $RefT$, we demand that S and T be equivalent under the subtype relation–each a subtype of the other. This gives us

the flexibility to reorder the fields of records under a Ref constructor–for example, $Ref\{a : Bool, b : Nat\} <: Ref\{b : Nat, a : Bool\}$–but nothing more.

The reason for this very restrictive subtyping rule is that a value of type $RefT$ can be used in a given context in two different ways: for both reading (!) and writing (:=). When it is used for reading, the context expects to obtain a value of type T, so if the reference actually yields a value of type S then we need $S <: T$ to avoid violating the context's expectations. On the other hand, if the same reference cell is used for writing, then the new value provided by the context will have type T. If the actual type of the reference is $RefS$, then someone else may later read this value and use it as an S; this will be safe only if $T <: S$.

11.17 Arrays

Clearly, the motivations behind the invariant subtyping rule for references also apply to arrays, since the operations on arrays include forms of both dereferencing and assignment.

$$\frac{S <: T \quad T <: S}{ArrayS <: ArrayT} \qquad \text{(S-ARRAY)}$$

Interestingly, Java actually permits covariant subtyping of arrays:

$$\frac{S <: T}{ArrayS <: ArrayT} \qquad \text{(S-ARRAYJAVA)}$$

This feature was originally introduced to compensate for the lack of parametric polymorphism in the typing of some basic operations such as copying parts of arrays, but is now generally considered a flaw in the language design, since it seriously affects the performance of programs involving arrays. The reason is that the unsound subtyping rule must be compensated with a run-time check on every assignment to any array, to make sure the value being written belongs to (a subtype of) the actual type of the elements of the array.

11.18 Base Types

In a full-blown language with a rich set of base types, it is often convenient to introduce primitive subtype relations among these types.

For example, in many languages the boolean values $true$ and $false$ are actually represented by the numbers 1 and 0. We can, if we like, expose this fact to the programmer by introducing a subtyping axiom $Bool <: Nat$. Now we can write compact expressions like $5 * b$ instead of $if\ b\ then\ 5\ else\ 0$.

11.19 Coercion Semantics For Subtyping

Throughout this lecture, our intuition has been that subtyping is "semantically insignificant". The presence of subtyping does not change the way programs are evaluated; rather, subtyping is just a way of obtaining additional flexibility in typing terms. This interpretation is simple and natural, but it carries some performance penalties–particularly for numerical calculations and for accessing record fields–that may not be acceptable in high-performance implementations.

We can address these kind of problems by adopting a different semantics, in which we "compile away" subtyping by replacing it with run-time coercions. If an Int is promoted to a $Float$ during typechecking, for example, then at run time we physically change this number's representation from a machine integer to a machine float. Similarly, a use of the record permutation subtyping rule will be compiled into a piece of code that literally rearranges the order of the fields. Primitive numeric operations and field accesses can now proceed without the overhead of unboxing or search.

Intuitively, the coercion semantics for a language with subtyping is expressed as a function that transforms terms from this language into a lower-level language without subtyping. Ultimately, the low-level language might be machine code for some concrete processor.

11.20 Summary

Subtyping is heavily researched and fairly well understood. It greatly improves the re-use of code, but has some tricky semantics and may complicate type checking and inference.

Bibliography

[1] Pierce, Benjamin C., *Types and Programming Languages*, The MIT Press, 2002, pp. 118-146, 153-170.