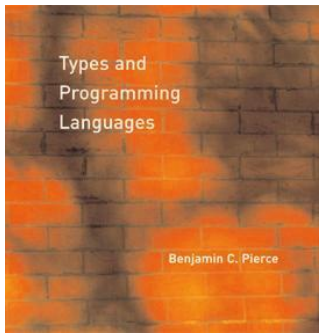
Typed λ-Calculus: Subtyping

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA

DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on chapter 15 of: Benjamin C. Pierce, *Types and Programming Languages*. The MIT Press, 2002



Subtyping

We address a fundamental extension: subtyping (sometimes called subtype polymorphism)

A cross-cutting extension, interacting with most other language features in non-trivial ways

Characteristically found in object-oriented languages, it is often considered an essential feature of the object-oriented style

Subsumption

Without subtyping, the rules of the simply typed lambda-calculus can be annoyingly rigid

The typechecker rejects many programs that, to the programmer, seem obviously well-behaved:

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

is not typable, since the type of the argument is $\{x{:}\mathtt{Nat}, y{:}\mathtt{Nat}\}$, whereas the function accepts $\{x{:}\mathtt{Nat}\}$

But the function just requires that its argument is a record with a field x

The goal of subtyping

The goal of subtyping is to refine the typing rules so that they can accept terms like the one in the previous slide

Formalizing the intuition that some types are more informative than others:

- `S` is a subtype of `T` (`S<:T`) ny term of type `S` can safely be used in a context where a term of type `T` is expected
- `S<:T`: "every value described by `S` is also described by `T`"

  "the elements of `S` are a subset of the elements of `T`".

Rule of subsumption

The bridge between the typing relation and this subtype relation is provided by adding a new typing rule—the so-called rule of subsumption:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \quad (\text{T-SUB})$$

- if $\mathtt{S} <: \mathtt{T}$, then every element $\mathtt{t}$ of $\mathtt{S}$ is also an element of $\mathtt{T}$
- If we define the subtype relation so that $\{\mathtt{x}:\mathtt{Nat}, \mathtt{y}:\mathtt{Nat}\} <: \{\mathtt{x}:\mathtt{Nat}\}$, then we can use rule T-SUB to derive $\Gamma \vdash \{\mathtt{x} = 0, \mathtt{y} = 1\} : \{\mathtt{x} : \mathtt{Nat}\}$, which is what we need to make our motivating example typecheck

The subtype relation

Formalized as a collection of inference rules `S<:T`: "`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 type, 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

Two general stipulations:

1. Subtyping should be reflexive

$$\mathtt{S} <: \mathtt{S} \qquad \text{(S-Refl)}$$

2. Subtyping should be transitive:

$$\frac{\mathtt{S}<:\mathtt{U} \qquad \mathtt{U}<:\mathtt{T}}{\mathtt{S}<:\mathtt{T}} \quad \text{(S-Trans)}$$

Records: Width subtyping rule

We want to consider $\mathtt{S} = \{\mathtt{l}_1 : \mathtt{T}_1, \mathtt{l}_2 : \mathtt{T}_2, \ldots, \mathtt{l}_m : \mathtt{T}_m\}$ to be a subtype of $\mathtt{T} = \{\mathtt{l}_1 : \mathtt{T}_1, \mathtt{l}_2 : \mathtt{T}_2, \ldots, \mathtt{l}_n : \mathtt{T}_n\}$ $(\mathtt{m} > \mathtt{n})$

Width subtyping rule:

$$\{\mathtt{l}_i : \mathtt{T}_i^{\,i=1..n+k}\} \mathtt{<:} \{\mathtt{l}_i : \mathtt{T}_i^{\,i=1..n}\} \qquad \text{(S-RcdWidth)}$$

The subtype is the one with more fields

The rule applies only to record types where the common fields are identical. Only fields at the end are "deleted"

Records: examples

- {x:Nat} describes "the set of all records with at least a field x of type Nat"

  {x=3}, {x=5}, {x=3,y=100} and {x=3,a=true,b=true} are elements of this type
- {x:Nat,y:Nat} describes "records with at least the fields x and y, both of type Nat

  {x=3,y=100} and {x=3,y=100,z=true} are elements of this type

  but {x=3,a=true,b=true}, {x=3} and {x=5} are not
- 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

Records: Depth subtyping rule

Depth subtyping rule: the types of individual fields can vary, as long as the types of each corresponding field in the two records are in the subtype relation

$$\frac{\forall \mathtt{i} \quad \mathtt{S_i} <: \mathtt{T_i}}{\{\mathtt{l_i}:\mathtt{S_i}^{\mathtt{i}=1..\mathtt{n}}\} <: \{\mathtt{l_i}:\mathtt{T_i}^{\mathtt{i}=1..\mathtt{n}}\}} \quad (\text{S-RcdDepth})$$

Records: more examples

Example: Using S-RcdDepth and S-RcdWidth together:

$$\dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \text{S-RcdWidth} \qquad \dfrac{}{\texttt{\{m:Nat\}} <: \texttt{\{\}}}\ \text{S-RcdWidth}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat\},y:\{\}\}}}\ \text{S-RcdDepth}$$

Example: Using S-Refl:

$$\dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \text{S-RcdWidth} \qquad \dfrac{}{\texttt{\{m:Nat\}} <: \texttt{\{m:Nat\}}}\ \text{S-Refl}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat\},y:\{m:nat\}\}}}\ \text{S-RcdDepth}$$

Records: even more examples

Example: using S-TRANS to combine width and depth subtyping:

$$\dfrac{\dfrac{}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat,b:Nat\}\}}}\ \text{S-RcdWidth} \qquad \dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \text{S-RcdWidth}}{\texttt{\{x:\{a:Nat,b:Nat\}\}} <: \texttt{\{x:\{a:Nat\}\}}}\ \text{S-RcdDepth}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat\}\}}}\ \text{S-Trans}$$

Records: Permutation subtyping rule

Permutation subtyping rule: Order of fields in a record does not make any difference to how we can safely use it

$$\frac{\{k_j{:}S_j^{j=1..n}\} \text{ is a permutation of } \{l_i{:}T_i^{i=1..n}\}}{\{k_j{:}S_j^{j=1..n}\}<:\{l_i{:}T_i^{i=1..n}\}} \quad \text{(S-RcdPerm)}$$

- $\{\texttt{c:Bool,b:Bool,a:Nat}\}$ is a subtype of $\{\texttt{a:Nat,b:Bool,c:Bool}\}$ nd vice versa

- S-RcdPerm can be used in combination with S-RcdWidth and S-Trans to drop fields from anywhere in a record type

Functions

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 \;<:\; \mathtt{T}_1 \to \mathtt{T}_2} \quad (\text{S-ARROW})$$

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.

Functions: Example

Let us verify that the example at the beginning now typechecks

Abbreviations:

```
f = λr:{x:Nat}. r.x
xy = {x=0,y=1}
```

```
Rx = {x:Nat}
Rxy = {x:Nat,y:Nat}
```

Derivation for the typing statement $\vdash$ `f xy:Nat`

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{0:Nat}} \quad \dfrac{}{\vdash \texttt{1:Nat}}}{\vdash \texttt{xy:Rxy}}\ \text{T-Rcd} \qquad \dfrac{}{\texttt{Rxy <: Rx}}\ \text{S-RcdWidth}}{\dfrac{\dfrac{\ldots}{\vdash \texttt{f:Rx}\to\texttt{Nat}} \qquad \dfrac{\vdash \texttt{xy:Rxy} \qquad \texttt{Rxy <: Rx}}{\vdash \texttt{xy:Rx}}\ \text{T-Sub}}{\vdash \texttt{f xy:Nat}}\ \text{T-App}}$$

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.

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

`Top` type:

- Can be removed without damaging the properties of the system
- Corresponds to the type `Object` found in most object-oriented languages
- Is a convenient technical device in more sophisticated systems ning subtyping and parametric polymorphism

Bottom type

A *minimal type* `Bot` that is a subtype of every type:

```
T::= ...
     Bot
Bot<:T
```

`Bot` is empty, there are no closed values of type `Bot`.

If there were one, say $\mathtt{v} \in \mathtt{Bot}$, then the subsumption rule plus S-Bot would allow us to derive $\vdash \mathtt{v}:\mathtt{Top}\to\mathtt{Top}$ and $\vdash \mathtt{v}:\{\}$. This led us to a contradiction!!

Bottom type

Giving an expression the type `Bot` has two good effects:

1. It signals to the programmer that no result is expected
2. It signals to the typechecker that such an expression can safely be used in a context expecting any type of value

For example, the exception-raising term `error` can be given type `Bot`:

```
λx:T.
    if <check that x is reasonable> then
       <compute result>
    else error
```

The presence of `Bot` significantly complicates the problem of building a typechecker

Ascription and casting

`t as T`:

- Checked documentation. Debugging
- Sums and variants

In languages with subtyping such as Java and C++, ascription becomes quite a bit more interesting:

- It is often called casting in these languages, and is written `(T) t`.
- two quite different forms of casting: up-casts and down-casts

Up-casts

Up-casts, in which a term is ascribed a supertype of the type that the type- checker would naturally assign it, are instances of the standard ascription operator:

- We give a term `t` and a type `T` at which we intend to "view" `t`.
- The typechecker verifies that `T` is indeed one of the types of `t` by attempting to build a derivation

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \texttt{t} : \texttt{S}} \qquad \dfrac{\dots}{\texttt{S} <: \texttt{T}}}{\dfrac{\Gamma \vdash \texttt{t} : \texttt{T}}{\Gamma \vdash \texttt{t as T} : \texttt{T}}\ \textsc{T-Ascribe}}\ \textsc{T-Sub}$$

Up-casts can be viewed as a form of abstraction, a way of hiding the existence of some parts of a value so that they cannot be used in some surrounding context

Down-casts

A down-cast allows us to assign types to terms that the typechecker cannot derive statically

To allow down-casts, we make a change to the typing rule for ascription:

$$\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \quad \text{(T-Downcast)}$$

where we check that `t1` is well typed (i.e., that it has some type `S`) and then assign it type `T`, without making any demand about the relation between `S` and `T`

Using down-casting we can write::

$$\texttt{f = } \lambda \texttt{x:Top. (x as \{a:Nat\}).a}$$

where the programmer is saying to the typechecker "I know (for reasons that are too complex to explain in terms of the typing rules) that `f` will always be applied to record arguments with numeric a fields; I want you to trust me on this one"

Down-casts: caution

Blindly trusting down-cast assertions will have a disastrous effect on the safety of our language:

- If `f` is applied to a record that does not contain an `a` field, the results might be completely arbitrary!
- Our motto should be trust, but verify
- At compile time, the typechecker accepts the type given in the down-cast but it inserts a run-time check

Down-casts: evaluation rule

The evaluation rule for ascriptions should not just discard the annotation, as our original evaluation rule for ascriptions did, but should first compare the actual (run-time) type of the value with the de- clared type:

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \to v_1} \quad \text{(E-Downcast)}$$

- If we apply the function `f` to the argument `{a=5,b=true}`, `E-Downcast` then this rule will check (successfully) that `{a=5,b=true}:{a:Nat}`
- if we apply `f` to `{b=true}` then the rule will not apply and evaluation will get stuck at this point. This run-time check recovers the type preservation property.

For some languages, down-casts support a kind of "poor-man's polymorphism."

Variants

Rules are nearly identical to the ones for records

But width rule S-VariantWidth allows **new variants to be added, not dropped, when moving from a subtype to a supertype**:

$$\texttt{<}\texttt{l}_i\texttt{:}\texttt{T}_i^{\,i=1..n}\texttt{>} \quad \texttt{<:} \quad \texttt{<}\texttt{l}_i\texttt{:}\texttt{T}_i^{\,i=1..n+k}\texttt{>} \qquad \text{(S-VariantWidth)}$$

- A singleton variant type <l=t> tells us precisely what label its elements are tagged with
- A two-variant type <$\texttt{l}_1$:$\texttt{T}_1$,$\texttt{l}_2$:$\texttt{T}_2$ > tells us that its elements have either label $\texttt{l}_1$ or label $\texttt{l}_2$, etc.

Adscription is not longer needed to define variants

Variants without adscription

Syntatic forms:

```
t::= ...
       <l=t>
```

Typing rules:

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash <l_1 = t_1> : <l_1 : T_1>} \quad \text{(T-Variant)}$$

Subtyping rules::

$$< l_i : T_i^{i=1..n} > <: < l_i : T_i^{i=1..n+k} > \quad \text{(S-VariantWidth)}$$

$$\frac{\forall i \quad S_i <: T_i}{<l_i : S_i^{i=1..n}> <: <l_i : T_i^{i=1..n}>} \quad \text{(S-VariantDepth)}$$

$$\frac{<k_j : S_j^{j=1..n}> \text{ is a permutation of } <l_i : T_i^{i=1..n}>}{<k_j : S_j^{j=1..n}> <: <l_i : T_i^{i=1..n}>} \quad \text{(S-VariantPerm)}$$

Lists

The List constructor is covariant

If we have a list whose elements have type `S` , and `S<:T`, then we can safely regard our list as having elements of type `T` .

$$\frac{\texttt{S} <: \texttt{T}}{\texttt{List S} <: \texttt{List T}} \quad \text{(S-List)}$$

References

Subtyping rule for references:

$$\frac{\texttt{S} <: \texttt{T} \qquad \texttt{T} <: \texttt{S}}{\texttt{Ref S} <: \texttt{Ref T}} \quad (S - Ref)$$

- For `Ref S <: Ref T` we demand that `S` and `T` be equivalent
- We can only reorder the fields of records under a Ref constructor: `Ref {a:Bool,b:Nat} <: Ref {b:Nat,a:Bool}`

References

Why a so restrictive sybtyping relation?

Values of type Ref can be read (`!`) and writen (`:=`)

- Case `!`t_1: the context expects to obtain a value of type `T`; if `!t` yields a value of type `S` then we need `S<:T`.
- Case: t_1`:=`t_2: value provided by the context will have type `T`, If the actual type of the reference is `Ref S` then someone else may later read this value and use it as an `S` and this will be safe only if `T<:S`.

Source and Sink

Two new constructors:, `Source` and `Sink`.

- `Source T` allows us to read values of type `T` from a memory cell
- `Sink T` allows us to read values of type `T` from a memory cell
- `Ref T` can be understood as a combination of both constructors

Typing and subtyping Source and Sink

Typing:

$$\frac{\Gamma|\Sigma \vdash t_1 : \texttt{Source } T_1}{\Gamma|\Sigma \vdash !t_1 : T_1} \quad \text{(T-Deref)}$$

$$\frac{\Gamma|\Sigma \vdash t_1 : \texttt{Sink } T_1 \qquad \Gamma|\Sigma \vdash t_2 : T_1}{\Gamma|\Sigma \vdash !t_1 := t_2 : \texttt{Unit}} \quad \text{(T-Assign)}$$

Subtyping: `Source` is covariant, `Sink` es contravariant

$$\frac{\texttt{S} <: \texttt{T}}{\texttt{Source S} <: \texttt{Source T}} \quad \text{(S-Source)}$$

$$\frac{\texttt{T} <: \texttt{S}}{\texttt{Sink S} <: \texttt{Sink T}} \quad \text{(S-Sink)}$$

$$\texttt{Ref T} <: \texttt{Source T} \quad \text{(S-RefSource)}$$

$$\texttt{Ref T} <: \texttt{Sink T} \quad \text{(S-RefSink)}$$

End