Semantic subtyping for non-strict languages

Davide Ancona[1], Giuseppe Castagna[2], Tommaso Petrucciani[1,2], and Elena Zucca[1]

[1] DIBRIS, Università di Genova, Italy
[2] IRIF, CNRS and Université Paris Diderot, France

Semantic subtyping is an approach to define type systems featuring union and intersection types and a precise subtyping relation. It has been developed for strict languages, and it is unsound for non-strict semantics. We describe how to adapt it to languages with lazy evaluation.

Semantic subtyping. Union and intersection types can be used to type several language constructs – from branching and pattern matching to overloading – very precisely. However, they make it challenging to define a subtyping relation that behaves precisely and intuitively.

Semantic subtyping is a technique to do so, studied by Frisch et al. [1] for types given by:

$$t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathsf{Empty} \mid \mathsf{Any} \qquad \text{where } b ::= \mathsf{Int} \mid \mathsf{Bool} \mid \cdots$$

Types include constructors – basic types b, arrows, and products – plus union $t \vee t$, intersection $t \wedge t$, negation (or complementation) $\neg t$, and the bottom and top types Empty and Any (actually, $t_1 \wedge t_2$ and Any can be defined as $\neg(\neg t_1 \vee \neg t_2)$ and $\neg\mathsf{Empty}$). Types can also be recursive (simply, by considering the types that are *coinductively* generated by the productions above).

Subtyping is defined by giving an interpretation $[\![\,\cdot\,]\!]$ of types as sets and defining $t_1 \leq t_2$ as $[\![t_1]\!] \subseteq [\![t_2]\!]$. Intuitively, we can see $[\![t]\!]$ as the set of values which inhabit t in the language. By interpreting union, intersection, and negation as the corresponding operations on sets, we ensure that subtyping will satisfy all commutative and distributive laws we expect (e.g., $(t_1 \times t_2) \vee (t'_1 \times t'_2) \leq (t_1 \vee t'_1) \times (t_2 \vee t'_2)$ or $(t \to t_1) \wedge (t \to t_2) \leq t \to (t_1 \wedge t_2)$).

This relation is used to type a call-by-value language featuring higher-order functions, data constructors and destructors (pairs), and a typecase construct which models runtime type dispatch and acts as a form of pattern matching. Functions can be recursive and are explicitly typed: their type can be an intersection of arrow types, describing overloaded behaviour.

Semantic subtyping in lazy languages. Current semantic subtyping systems are unsound for non-strict semantics because of the way they deal with the bottom type Empty, which corresponds to the empty set of values ($[\![\mathsf{Empty}]\!] = \varnothing$). The intuition is that a (reducible) expression e can be safely given a type t only if, whenever e returns a result, this result is a value in t. Accordingly, Empty can only be assigned to expressions that are statically known to diverge (i.e., that never return a result). For example, the ML expression $\mathsf{let\ rec\ f\ x = f\ x\ in\ f\ ()}$ has type Empty. Let $\bar{e}$ be this expression and consider the following typing derivations, which are valid in semantic subtyping systems (π_2 projects the second component of a pair).

$$\dfrac{[\simeq]\ \dfrac{\vdash (\bar{e}, 3) \colon \mathsf{Empty} \times \mathsf{Int}}{\vdash (\bar{e}, 3) \colon \mathsf{Empty} \times \mathsf{Bool}}}{\vdash \pi_2\ (\bar{e}, 3) \colon \mathsf{Bool}} \qquad \dfrac{[\simeq]\ \dfrac{\vdash \lambda x.\, 3 \colon \mathsf{Empty} \to \mathsf{Int}}{\vdash \lambda x.\, 3 \colon \mathsf{Empty} \to \mathsf{Bool}} \qquad \vdash \bar{e} \colon \mathsf{Empty}}{\vdash (\lambda x.\, 3)\ \bar{e} \colon \mathsf{Bool}}$$

Note that both $\pi_2\ (\bar{e}, 3)$ and $(\lambda x.\, 3)\ \bar{e}$ diverge in call-by-value semantics (since $\bar{e}$ must be evaluated first), while they both reduce to 3 in call-by-name or call-by-need. The derivations are therefore sound for call-by-value, while they are clearly unsound with non-strict evaluation.

Why are these derivations valid? The crucial steps are those marked with $[\simeq]$, which convert between types that have the same interpretation; $\simeq$ denotes this equivalence relation. With semantic subtyping, $\mathsf{Empty} \times \mathsf{Int} \simeq \mathsf{Empty} \times \mathsf{Bool}$ holds because all types of the form $\mathsf{Empty} \times t$ are equivalent to Empty itself: none of these types contains any value (indeed, product types are interpreted as Cartesian products). It can appear more surprising that $\mathsf{Empty} \to \mathsf{Int} \simeq \mathsf{Empty} \to \mathsf{Bool}$ holds. We interpret a type $t_1 \to t_2$ as the set of functions which, on arguments of type t_1, return results in type t_2. Since there is no argument of type Empty (because, in call-by-value, arguments are always values), all types of the form $\mathsf{Empty} \to t$ are equivalent.

In passing, note that somewhat similar problems occur when using refinement types for non-strict semantics, as studied by Vazou et al. [2] (the false refinement is analogous to Empty).

Our approach. The intuition behind our solution is that, with non-strict semantics, it is not appropriate to see a type as the set of the values that have that type. In a call-by-value language, operations like application or projection must occur on values: thus, we can identify two types if they contain the same values. In non-strict languages, instead, operations also occur on partially evaluated results which might contain diverging sub-expressions.

The Empty type is important for the internal machinery of subtyping. Notably, the decision procedure for subtyping relies on the existence of types with empty interpretation (e.g., $t_1 \leq t_2$ if and only if $t_1 \wedge \neg t_2$ is empty). In a strict setting, it is sound to assign Empty to diverging computations. In a non-strict one, though, Empty should be completely empty: no expression at all should inhabit it. Diverging expressions should have a different type, with non-empty interpretation. We add this new type, written $\perp$, and have it be non-empty but disjoint from the types of constants, functions, and pairs: $[\![\perp]\!]$ is a singleton whose element represents divergence.

Introducing the $\perp$ type means that we track termination in types, albeit very approximately. We allow the derivation of types like Int, $\mathsf{Int} \to \mathsf{Bool}$, or $\mathsf{Int} \times \mathsf{Bool}$ (which are disjoint from $\perp$) only for expressions that are statically guaranteed to terminate. In particular, our rules can only derive them for constants, functions, and pairs. Application and projection, instead, always have types of the form $t \vee \perp$, meaning that they could diverge. We allow the typing rules to propagate the $\perp$ type. For example, to type the application $e_1\, e_2$, if $\Gamma \vdash e_2 \colon t$, we require $\Gamma \vdash e_1 \colon (t \to t') \vee \perp$ instead of $\Gamma \vdash e_1 \colon t \to t'$, so that e_1 is allowed to be possibly diverging. Then, $e_1\, e_2$ has type $t' \vee \perp$. Subtyping verifies $t \leq t \vee \perp$, so a terminating expression can be used where a possibly diverging one is expected. Because $[\![\perp]\!]$ is non-empty, the problematic type equivalences we have seen do not hold for it. Indeed, $\perp \times \mathsf{Int}$ is now the type of pairs whose first component diverges and the second evaluates to an Int: it is not equivalent to $\perp \times \mathsf{Bool}$.

Typing all applications as possibly diverging is a very coarse approximation, but it achieves our goal of giving a sound type system that still enjoys most benefits of semantic subtyping. Indeed, $\perp$ can be seen as an “internal” type that does not need to be written explicitly by programmers. As future work we intend to explore whether we can give a more expressive system, while maintaining soundness, by tracking termination somewhat more precisely, and to give a semantic interpretation of types in terms of sets of “results” rather than of “values”.

We have defined this type system for a call-by-need variant of the language studied by Frisch et al. [1], and we have proved its soundness. The choice of call-by-need stems from the presence of union and intersection types: indeed, for quite technical reasons, our system is not sound for call-by-name if we assume that reduction might be non-deterministic.

References

[1] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–67, 2008.

[2] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *ICFP '14*, 2014.