Part XI

Subtyping

Chapter 32

Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) on types that validates the *subsumption principle*:

> if σ is a subtype of τ, then a value of type σ may be provided whenever a value of type τ is required.

The subsumption principle relaxes the strictures of a type system to permit values of one type to be treated as values of another.

Experience shows that the subsumption principle, while useful as a general guide, can be tricky to apply correctly in practice. The key to getting it right is the principle of introduction and elimination. To determine whether a candidate subtyping relationship is sensible, it suffices to consider whether every *introductory* form of the subtype can be safely manipulated by every *eliminatory* form of the supertype. A subtyping principle makes sense only if it passes this test; the proof of the type safety theorem for a given subtyping relation ensures that this is the case.

A good way to get a subtyping principle wrong is to think of a type merely as a set of values (generated by introductory forms), and to consider whether every value of the subtype can also be considered to be a value of the supertype. The intuition behind this approach is to think of subtyping as akin to the subset relation in ordinary mathematics. But this can lead to serious errors, because it fails to take account of the operations (eliminatory forms) that one can perform on values of the supertype. It is not enough to think only of the introductory forms; one must also think of the eliminatory forms. Subtyping is a matter of *behavior*, rather than *containment*.

32.1 Subsumption

A *subtyping judgement* has the form $\sigma <: \tau$, and states that σ is a subtype of τ. At a minimum we demand that the following *structural rules* of subtyping be admissible:

$$\frac{}{\tau <: \tau} \tag{32.1a}$$

$$\frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau} \tag{32.1b}$$

In practice we either tacitly include these rules as primitive, or prove that they are admissible for a given set of subtyping rules.

The point of a subtyping relation is to enlarge the set of well-typed programs, which is achieved by the *subsumption rule*:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{32.2}$$

In contrast to most other typing rules, the rule of subsumption is *not* syntax-directed, because it does not constrain the form of e. That is, the subsumption rule may be applied to *any* form of expression. In particular, to show that $e : \tau$, we have two choices: either apply the rule appropriate to the particular form of e, or apply the subsumption rule, checking that $e : \sigma$ and $\sigma <: \tau$.

32.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightharpoonup\}$. In Section 32.4 on page 298 we will examine some of these in more detail from the point of view of type safety.

32.2.1 Numeric Types

For languages with numeric types, our mathematical experience suggests subtyping relationships among them. For example, in a language with types `int`, `rat`, and `real`, representing, respectively, the integers, the rationals, and the reals, it is tempting to postulate the subtyping relationships

$$\texttt{int} <: \texttt{rat} <: \texttt{real}$$

by analogy with the set containments

$$\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$$

familiar from mathematical experience.

But are these subtyping relationships sensible? The answer depends on the representations and interpretations of these types! Even in mathematics, the containments just mentioned are usually not quite true—or are true only in a somewhat generalized sense. For example, the set of rational numbers may be considered to consist of ordered pairs (m, n), with $n \neq 0$ and $\gcd(m, n) = 1$, representing the ratio m/n. The set $\mathbb{Z}$ of integers may be isomorphically embedded within $\mathbb{Q}$ by identifying $n \in \mathbb{Z}$ with the ratio $n/1$. Similarly, the real numbers are often represented as convergent sequences of rationals, so that strictly speaking the rationals are not a subset of the reals, but rather may be embedded in them by choosing a canonical representative (a particular convergent sequence) of each rational.

For mathematical purposes it is entirely reasonable to overlook fine distinctions such as that between $\mathbb{Z}$ and its embedding within $\mathbb{Q}$. This is justified because the operations on rationals restrict to the embedding in the expected manner: if we add two integers thought of as rationals in the canonical way, then the result is the rational associated with their sum. And similarly for the other operations, provided that we take some care in defining them to ensure that it all works out properly. For the purposes of computing, however, one cannot be quite so cavalier, because we must also take account of algorithmic efficiency and the finiteness of machine representations. Often what are called "real numbers" in a programming language are, in fact, finite precision floating point numbers, a small subset of the rational numbers. Not every rational can be exactly represented as a floating point number, nor does floating point arithmetic restrict to rational arithmetic, even when its arguments are exactly represented as floating point numbers.

32.2.2 Product Types

Product types give rise to a form of subtyping based on the subsumption principle. The only elimination form applicable to a value of product type is a projection. Under mild assumptions about the dynamic semantics of projections, we may consider one product type to be a subtype of another by considering whether the projections applicable to the supertype may be validly applied to values of the subtype.

Consider a context in which a value of type $\tau = \prod_{j \in J} \tau_j$ is required. The static semantics of finite products (Rules (16.3)) ensures that the only operation we may perform on a value of type τ, other than to bind it to a variable, is to take the jth projection from it for some $j \in J$ to obtain a

value of type τ_j. Now suppose that e is of type σ. If the projection $e \cdot j$ is to be well-formed, then σ must be a finite product type $\prod_{i \in I} \sigma_i$ such that $j \in I$. Moreover, for this to be of type τ_j, it is enough to require that $\sigma_j = \tau_j$. Since $j \in J$ is arbitrary, we arrive at the following subtyping rule for finite product types:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j} \ . \tag{32.3}$$

It is sufficient, but not necessary, to require that $\sigma_j = \tau_j$ for each $j \in J$; we will consider a more liberal form of this rule in Section 32.3 on page 294.

The argument for Rule (32.3) is based on a dynamic semantics in which we may evaluate $e \cdot j$ regardless of the actual form of e, provided only that it has a field indexed by $j \in J$. Is this a reasonable assumption?

One common case is that I and J are initial segments of the natural numbers, say $I = [0..m-1]$ and $J = [0..n-1]$, so that the product types may be thought of as m- and n-tuples, respectively. The containment $I \subseteq J$ amounts to requiring that $m \geq n$, which is to say that a tuple type is regarded as a subtype of all of its prefixes. When specialized to this case, Rule (32.3) may be stated in the form

$$\frac{m \geq n}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \ . \tag{32.4}$$

One way to justify this rule is to consider elements of the subtype to be consecutive sequences of values of type $\tau_0, \ldots, \tau_{m-1}$ from which we may calculate the jth projection for any $0 \leq j < n \leq m$, regardless of whether or not m is strictly bigger than n.

Another common case is when I and J are finite sets of symbols, so that projections are based on the field name, rather than its position. When specialized to this case, Rule (32.3) takes the following form:

$$\frac{m \geq n}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \ . \tag{32.5}$$

Here we are taking advantage of the implicit identification of labeled tuple types up to reordering of fields, so that the rule states that any field of the supertype must be present in the subtype with the same type.

When using symbolic labels for the components of a tuple, it is perhaps slightly less clear that Rule (32.5) is well-justified. After all, how are we to find field l_i, where $0 \leq i < n$, in a labeled tuple that may have additional fields anywhere within it? The trouble is that the label does not reveal the position of the field within the tuple, precisely because of subtyping. One

way to achieve this is to associate with a labeled tuple a *dictionary* mapping labels to positions within the tuple, which the projection operation uses to find the appropriate component of the record. Since the labels are fixed statically, this may be done in constant time using a perfect hashing function mapping labels to natural numbers, so that the cost of a projection remains constant. Another method is to use *coercions* that a value of the subtype to a value of the supertype whenever subsumption is used. In the case of labeled tuples this means creating a new labeled tuple containing only the fields of the supertype, copied from those of the subtype, so that the type specifies exactly the fields present in the value. This allows for more efficient implementation (for example, by a simple offset calculation), but is not compatible with languages that permit mutation (in-place modification) of fields because it destroys sharing.

32.2.3 Sum Types

By an argument dual to the one given for finite product types we may derive a related subtyping rule for finite sum types. If a value of type $\sum_{j \in J} \tau_j$ is required, the static semantics of sums (Rules (17.3)) ensures that the only non-trivial operation that we may perform on that value is a J-indexed case analysis. If we provide a value of type $\sum_{i \in I} \sigma_i$ instead, no difficulty will arise so long as $I \subseteq J$ and each σ_i is equal to τ_i. If the containment is strict, some cases cannot arise, but this does not disrupt safety. This leads to the following subtyping rule for finite sums:

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i <: \sum_{j \in J} \tau_j} \ . \tag{32.6}$$

Note well the reversal of the containment as compared to Rule (32.3).

When I and J are initial segments of the natural numbers, we obtain the following special case of Rule (32.6):

$$\frac{m \leq n}{[l_1 : \tau_1, \ldots, l_m : \tau_m] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} \tag{32.7}$$

One may also consider a form of width subtyping for unlabeled n-ary sums, by considering any prefix of an n-ary sum to be a subtype of that sum. Here again the elimination form for the supertype, namely an n-ary case analysis, is prepared to handle any value of the subtype, which is enough to ensure type safety.

32.3 Variance

In addition to basic subtyping principles such as those considered in Section 32.2 on page 290, it is also important to consider the effect of subtyping on type constructors. A type constructor is said to be *covariant* in an argument if subtyping in that argument is preserved by the constructor. It is said to be *contravariant* if subtyping in that argument is reversed by the constructor. It is said to be *invariant* in an argument if subtyping for the constructed type is not affected by subtyping in that argument.

32.3.1 Product Types

Finite product types are *covariant* in each field. For if e is of type $\prod_{i\in I}\sigma_i$, and the projection $e\cdot j$ is expected to be of type τ_j, then it is sufficient to require that $j\in I$ and $\sigma_j <: \tau_j$. This is summarized by the following rule:

$$\frac{(\forall i\in I)\ \sigma_i <: \tau_i}{\prod_{i\in I}\sigma_i <: \prod_{i\in I}\tau_i} \tag{32.8}$$

It is implicit in this rule that the dynamic semantics of projection must not be sensitive to the precise type of any of the fields of a value of finite product type.

When specialized to n-tuples, Rule (32.8) reads as follows:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\langle\sigma_1,\ldots,\sigma_n\rangle <: \langle\tau_1,\ldots,\tau_n\rangle}\ . \tag{32.9}$$

When specialized to symbolic labels, the covariance principle for finite products may be re-stated as follows:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\langle l_1:\sigma_1,\ldots,l_n:\sigma_n\rangle <: \langle l_1:\tau_1,\ldots,l_n:\tau_n\rangle}\ . \tag{32.10}$$

32.3.2 Sum Types

Finite sum types are also covariant, because each branch of a case analysis on a value of the supertype expects a value of the corresponding summand, for which it is sufficient to provide a value of the corresponding subtype summand:

$$\frac{(\forall i\in I)\ \sigma_i <: \tau_i}{\sum_{i\in I}\sigma_i <: \sum_{i\in I}\tau_i} \tag{32.11}$$

When specialized to symbolic labels as index sets, we obtain the following formulation of the covariance principle for sum types:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{[l_1 : \sigma_1, \ldots, l_n : \sigma_n] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]}\ . \tag{32.12}$$

A case analysis on a value of the supertype is prepared, in the ith branch, to accept a value of type τ_i. By the premises of the rule, it is sufficient to provide a value of type σ_i instead.

32.3.3 Function Types

The variance of the function type constructor is a bit more subtle. Let us consider first the variance of the function type in its range. Suppose that $e : \sigma \to \tau$. This means that if $e_1 : \sigma$, then $e(e_1) : \tau$. If $\tau <: \tau'$, then $e(e_1) : \tau'$ as well. This suggests the following covariance principle for function types:

$$\frac{\tau <: \tau'}{\sigma \to \tau <: \sigma \to \tau'} \tag{32.13}$$

Every function that delivers a value of type τ must also deliver a value of type τ', provided that $\tau <: \tau'$. Thus the function type constructor is covariant in its range.

Now let us consider the variance of the function type in its domain. Suppose again that $e : \sigma \to \tau$. This means that e may be applied to any value of type σ, and hence, by the subsumption principle, it may be applied to any value of any subtype, σ', of σ. In either case it will deliver a value of type τ. Consequently, we may just as well think of e as having type $\sigma' \to \tau$.

$$\frac{\sigma' <: \sigma}{\sigma \to \tau <: \sigma' \to \tau} \tag{32.14}$$

The function type is contravariant in its domain position. Note well the reversal of the subtyping relation in the premise as compared to the conclusion of the rule!

Combining these rules we obtain the following general principle of contra- and co-variance for function types:

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \tag{32.15}$$

Beware of the reversal of the ordering in the domain!

32.3.4 Recursive Types

The variance principle for recursive types is rather subtle, and has been the source of errors in language design. To gain some intuition, consider the type of labeled binary trees with natural numbers at each node,

$$\mu t.\,[\mathtt{empty}:\mathtt{unit},\mathtt{binode}:\langle\mathtt{data}:\mathtt{nat},\mathtt{lft}:t,\mathtt{rht}:t\rangle],$$

and the type of "bare" binary trees, without labels on the nodes,

$$\mu t.\,[\mathtt{empty}:\mathtt{unit},\mathtt{binode}:\langle\mathtt{lft}:t,\mathtt{rht}:t\rangle].$$

Is either a subtype of the other? Intuitively, one might expect the type of labeled binary trees to be a *subtype* of the type of bare binary trees, since any use of a bare binary tree can simply ignore the presence of the label.

Now consider the type of bare "two-three" trees with two sorts of nodes, those with two children, and those with three:

$$\mu t.\,[\mathtt{empty}:\mathtt{unit},\mathtt{binode}:\langle\mathtt{lft}:t,\mathtt{rht}:t\rangle,\mathtt{trinode}:\langle\mathtt{lft}:t,\mathtt{mid}:t,\mathtt{rht}:t\rangle].$$

What subtype relationships should hold between this type and the preceding two tree types? Intuitively the type of bare two-three trees should be a *supertype* of the type of bare binary trees, since any use of a two-three tree must proceed by three-way case analysis, which covers both forms of binary tree.

To capture the pattern illustrated by these examples, we must formulate a subtyping rule for recursive types. It is tempting to consider the following rule:

$$\frac{t\ \mathsf{type} \vdash \sigma <: \tau}{\mu t.\,\sigma <: \mu t.\,\tau}\ \textbf{??} \tag{32.16}$$

That is, to determine whether one recursive type is a subtype of the other, we simply compare their bodies, with the bound variable treated as a parameter. Notice that by reflexivity of subtyping, we have $t <: t$, and hence we may use this fact in the derivation of $\sigma <: \tau$.

Rule (32.16) validates the intuitively plausible subtyping between labeled binary tree and bare binary trees just described. To derive this reduces to checking the subtyping relationship

$$\langle\mathtt{data}:\mathtt{nat},\mathtt{lft}:t,\mathtt{rht}:t\rangle <: \langle\mathtt{lft}:t,\mathtt{rht}:t\rangle,$$

generically in t, which is evidently the case.

Unfortunately, Rule (32.16) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\sigma = \mu t.\langle \mathtt{a}:t \to \mathtt{nat}, \mathtt{b}:t \to \mathtt{int}\rangle$$

and

$$\tau = \mu t.\langle \mathtt{a}:t \to \mathtt{int}, \mathtt{b}:t \to \mathtt{int}\rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using Rule (32.16) we may derive $\sigma <: \tau$, which we will show to be incorrect. Let $e : \sigma$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} = \lambda(x{:}\sigma.\,4), \mathtt{b} = \lambda(x{:}\sigma.\,q((\mathtt{unfold}(x)\cdot\mathtt{a})(x)))\rangle),$$

where $q : \mathtt{nat} \to \mathtt{nat}$ is the discrete square root function. Since $\sigma <: \tau$, it follows that $e : \tau$ as well, and hence

$$\mathtt{unfold}(e) : \langle \mathtt{a}:\tau \to \mathtt{int}, \mathtt{b}:\tau \to \mathtt{int}\rangle.$$

Now let $e' : \tau$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} = \lambda(x{:}\tau.\,\mathtt{-4}), \mathtt{b} = \lambda(x{:}\tau.\,\mathtt{0})\rangle).$$

(The important point about e' is that the `a` method returns a negative number; the `b` method is of no significance.) To finish the proof, observe that

$$(\mathtt{unfold}(e)\cdot\mathtt{b})(e') \mapsto^* q(\mathtt{-4}),$$

which is a stuck state. We have derived a well-typed program that "gets stuck", refuting type safety!

Rule (32.16) is therefore incorrect. But what has gone wrong? The error lies in the choice of a single parameter to stand for both recursive types, which does not correctly model self-reference. In effect we are regarding two distinct recursive types as equal while checking their bodies for a subtyping relationship. But this is clearly wrong! It fails to take account of the self-referential nature of recursive types. On the left side the bound variable stands for the subtype, whereas on the right the bound variable stands for the super-type. Confusing them leads to the unsoundness just illustrated.

As is often the case with self-reference, the solution is to *assume* what we are trying to prove, and check that this assumption can be maintained

by examining the bodies of the recursive types. To do so we maintain a finite set, Ψ, of hypotheses of the form

$$s_1 <: t_1, \ldots, s_n <: t_n,$$

which is used to state the rule of subsumption for recursive types:

$$\frac{\Psi, s <: t \vdash \sigma <: \tau}{\Psi \vdash \mu s.\sigma <: \mu t.\tau}. \tag{32.17}$$

That is, to check whether $\mu s.\sigma <: \mu t.\tau$, we assume that $s <: t$, since s and t stand for the respective recursive types, and check that $\sigma <: \tau$ under this assumption.

We tacitly include the rule of reflexivity for subtyping assumptions,

$$\frac{}{\Psi, s <: t \vdash s <: t} \tag{32.18}$$

Using reflexivity in conjunction with Rule (32.17), we may verify the subtypings among the tree types sketched above. Moreover, it is instructive to check that the unsound subtyping is *not* derivable using this rule. The reason is that the assumption of the subtyping relation is at odds with the contravariance of the function type in its domain.

32.4 Safety for Subtyping

Proving safety for a language with subtyping is considerably more delicate than for languages without. The rule of subsumption means that the static type of an expression reveals only partial information about the underlying value. This changes the proof of the preservation and progress theorems, and requires some care in stating and proving the auxiliary lemmas required for the proof.

As a representative case we will sketch the proof of safety for a language with subtyping for product types. The subtyping relation is defined by Rules (32.3) and (32.8). We assume that the static semantics includes subsumption, Rule (32.2).

Lemma 32.1 (Structurality)**.**

1. *The tuple subtyping relation is reflexive and transitive.*
2. *The typing judgement $\Gamma \vdash e : \tau$ is closed under weakening and substitution.*

Proof.

1. Reflexivity is proved by induction on the structure of types. Transitivity is proved by induction on the derivations of the judgements $\rho <: \sigma$ and $\sigma <: \tau$ to obtain a derivation of $\rho <: \tau$.
2. By induction on Rules (16.3), augmented by Rule (32.2).

□

Lemma 32.2 (Inversion)**.**

1. *If* $e \cdot j : \tau$, *then* $e : \prod_{i \in I} \tau_i$, $j \in I$, *and* $\tau_j <: \tau$.
2. *If* $\langle e_i \rangle_{i \in I} : \tau$, *then* $\prod_{i \in I} \sigma_i <: \tau$ *where* $e_i : \sigma_i$ *for each* $i \in I$.
3. *If* $\sigma <: \prod_{j \in J} \tau_j$, *then* $\sigma = \prod_{i \in I} \sigma_i$ *for some* I *and some types* σ_i *for* $i \in I$.
4. *If* $\prod_{i \in I} \sigma_i <: \prod_{j \in J} \tau_j$, *then* $I \subseteq J$ *and* $\sigma_j <: \tau_j$ *for each* $j \in J$.

Proof. By induction on the subtyping and typing rules, paying special attention to Rule (32.2). □

Theorem 32.3 (Preservation)**.** *If* $e : \tau$ *and* $e \mapsto e'$, *then* $e' : \tau$.

Proof. By induction on Rules (16.4). For example, consider Rule (16.4d), so that $e = \langle e_i \rangle_{i \in I} \cdot k$, $e' = e_k$. By Lemma 32.2 we have that $\langle e_i \rangle_{i \in I} : \prod_{j \in J} \tau_j$, $k \in J$, and $\tau_k <: \tau$. By another application of Lemma 32.2 for each $i \in I$ there exists σ_i such that $e_i : \sigma_i$ and $\prod_{i \in I} \sigma_i <: \prod_{j \in J} \tau_j$. By Lemma 32.2 again, we have $J \subseteq I$ and $\sigma_j <: \tau_j$ for each $j \in J$. But then $e_k : \tau_k$, as desired. The remaing cases are similar. □

Lemma 32.4 (Canonical Forms)**.** *If* e val *and* $e : \prod_{j \in J} \tau_j$, *then* e *is of the form* $\langle e_i \rangle_{i \in I}$, *where* $J \subseteq I$, *and* $e_j : \tau_j$ *for each* $j \in J$.

Proof. By induction on Rules (16.3) augmented by Rule (32.2). □

Theorem 32.5 (Progress)**.** *If* $e : \tau$, *then either* e val *or there exists* e' *such that* $e \mapsto e'$.

Proof. By induction on Rules (16.3) augmented by Rule (32.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. Rule (16.4d) follows from Lemma 32.4 on the preceding page. □

To account for recursive subtyping in addition to finite product subtyping, the following inversion lemma is required.

Lemma 32.6.

1. *If* $\Upsilon, s <: t \vdash \sigma' <: \tau'$ *and* $\Upsilon, \sigma <: \tau$, *then* $\Upsilon, [\sigma/s]\sigma' <: [\tau/t]\tau'$.
2. *If* $\Upsilon \vdash \sigma <: \mu t.\tau'$, *then* $\sigma = \mu s.\sigma'$ *and* $\Upsilon, s <: t \vdash \sigma' <: \tau'$.
3. *If* $\Upsilon \vdash \mu s.\sigma <: \mu t.\tau$, *then* $\Upsilon \vdash [\mu s.\sigma/s]\sigma <: [\mu t.\tau/t]\tau$.
4. *The subtyping relation is reflexive and transitive, and closed under weakening.*

Proof.

1. By induction on the derivation of the first premise. Wherever the assumption is used, replace it by $\sigma <: \tau$, and propagate forward.
2. By induction on the derivation of $\sigma <: \mu t.\tau$.
3. Follows immediately from the preceding two properties of subtyping.
4. Reflexivity is proved by construction. Weakening is proved by an easy induction on subtyping derivations. Transitivity is proved by induction on the sizes of the types involved. For example, suppose we have $\Upsilon \vdash \mu r.\rho <: \mu s.\sigma$ because $\Upsilon, r <: s \vdash \rho <: \sigma$, and $\Upsilon \vdash \mu s.\sigma <: \mu t.\tau$ because and $\Upsilon, s <: t \vdash \sigma <: \tau$. We may assume without loss of generality that s does not occur free in either ρ or τ. By weakening we have $\Upsilon, r <: s, s <: t \vdash \rho <: \sigma$ and $\Upsilon, r <: s, s <: t \vdash \sigma <: \tau$. Therefore by induction we have $\Upsilon, r <: s, s <: t \vdash \rho <: \tau$. But since $\Upsilon, r <: t \vdash r <: t$ and $\Upsilon, r <: t \vdash t <: t$, we have by the first property above that $\Upsilon, r <: t \vdash \rho <: \tau$, from which the result follows immediately.

□

The remainder of the proof of type safety in the presence of recursive subtyping proceeds along lines similar to that for product subtyping.

32.5 Exercises

Chapter 33

Singleton and Dependent Kinds

The expression $\texttt{let}\, e_1 : \tau\, \texttt{be}\, x\, \texttt{in}\, e_2$ is a form of abbreviation mechanism by which we may bind e_1 to the variable x for use within e_2. In the presence of function types this expression is definable as the application $\lambda(x:\tau.\, e_2)(e_1)$, which accomplishes the same thing. It is natural to consider an analogous form of `let` expression which permits a *type expression* to be bound to a type variable within a specified scope. The expression $\texttt{let}\, t\, \texttt{be}\, \tau\, \texttt{in}\, e$ binds t to τ within e, so that one may write expressions such as

$$\texttt{let}\, t\, \texttt{be}\, \texttt{nat} \times \texttt{nat}\, \texttt{in}\, \lambda(x:t.\, \texttt{s}(\texttt{pr}_1(x))).$$

For this expression to be type-correct the type variable t must be *synonymous* with the type $\texttt{nat} \times \texttt{nat}$, for otherwise the body of the λ-abstraction is not type correct.

Following the pattern of the expression-level `let`, we might guess that `lettype` is an abbreviation for the polymorphic instantiation $\Lambda(t.e)[\tau]$, which binds t to τ within e. This does, indeed, capture the dynamic semantics of type abbreviation, but it fails to validate the intended static semantics. The difficulty is that, according to this interpretation of `lettype`, the expression e is type-checked in the absence of any knowledge of the binding of t, rather than in the knowledge that t is synomous with τ. Thus, in the above example, the expression $\texttt{s}(\texttt{pr}_1(x))$ fails to type check, unless the binding of t were exposed.

The proposed definition of `lettype` in terms of type abstraction and type application fails. Lacking any other idea, one might argue that type abbreviation ought to be considered as a primitive concept, rather than a

derived notion. The expression $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$ would be taken as a primitive form of expression whose static semantics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e : \tau'} \tag{33.1}$$

This would address the problem of supporting type abbreviations, but it does so in a rather *ad hoc* manner. One might hope for a more principled solution that arises naturally from the type structure of the language.

Our methodology of identifying language constructs with type structure suggests that we ask not how to support type abbreviations, but rather what form of type structure gives rise to type abbreviations? And what else does this type structure suggest? By following this methodology we are led to the concept of *singleton kinds*, which not only account for type abbreviations but also play a crucial role in the design of module systems.

33.1 Informal Overview

The central organizing principle of type theory is *compositionality*. To ensure that a program may be decomposed into separable parts, we ensure that the composition of a program from constituent parts is mediated by the types of those parts. Put in other terms, the only thing that one portion of a program "knows" about another is its type. For example, the formation rule for addition of natural numbers depends only on the type of its arguments (both must have type `nat`), and not on their specific form or value. But in the case of a type abbreviation of the form $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$, the principle of compositionality dictates that the only thing that e "knows" about the type variable t is its kind, namely `Type`, and not its binding, namely τ. This is accurately captured by the proposed representation of type abbreviation as the combination of type abstraction and type application, but, as we have just seen, this is not the intended meaning of the construct!

We could, as suggested in the introduction, abandon the core principles of type theory, and introduce type abbreviations as a primitive notion. But there is no need to do so. Instead we can simply note that what is needed is for the kind of t to capture its identity. This may be achieved through the notion of a *singleton kind*. Informally, the kind $\texttt{Eqv}(\tau)$ is the kind of types that are definitionally equivalent to τ. That is, up to definitional equality, this kind has only one inhabitant, namely τ. Consequently, if $u :: \texttt{Eqv}(\tau)$ is a variable of singleton kind, then within its scope, the variable u is synonymous with τ. Thus we may represent $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$ by

$\Lambda(t::\mathtt{Eqv}(\tau).e)[\tau]$, which correctly propagates the identity of t, namely τ, to e during type checking.

A proper treatment of singleton kinds requires some additional machinery at the constructor and kind level. First, we must capture the idea that a constructor of singleton kind is *a fortiori* a constructor of kind `Type`, and hence is a type. Otherwise, a variable, u, singleton kind cannot be used as a type, even though it is explicitly defined to be one! This may be captured by introducing a *subkinding* relation, $\kappa_1 :<: \kappa_2$, which is analogous to subtyping, exception at the kind level. The fundamental axiom of subkinding is $\mathtt{Eqv}(\tau) :<: \mathtt{Type}$, stating that every constructor of singleton kind is a type.

Second, we must account for the occurrence of a constructor of kind `Type` within the singleton kind $\mathtt{Eqv}(\tau)$. This intermixing of the constructor and kind level means that singletons are a form of *dependent kind* in that a kind may depend on a constructor. Another way to say the same thing is that $\mathtt{Eqv}(\tau)$ represents a *family of kinds* indexed by constructors of kind `Type`. This, in turn, implies that we must generalize the function and product kinds to *dependent functions* and *dependent products*. The dependent function kind, $\Pi\, u::\kappa_1.\kappa_2$ classifies functions that, when applied to a constructor $c_1 :: \kappa_1$, results in a constructor of kind $[c_1/u]\kappa_2$. The important point is that the kind of the result is sensitive to the argument, and not just to its kind.[1] The dependent product kind, $\Sigma\, u::\kappa_1.\kappa_2$, classifies pairs $\langle c_1, c_2\rangle$ such that $c_1 :: \kappa_1$, as might be expected, and $c_2 :: [c_1/u]\kappa_2$, in which the kind of the second component is sensitive to the first component itself, and not just its kind.

Third, it is useful to consider singletons not just of kind `Type`, but also of higher kinds. To support this we introduce *higher-kind singletons*, written $\mathtt{Eqv}(c::\kappa)$, where κ is a kind and c is a constructor of kind k. These are definable in terms of the primitive form of singleton kind by making use of dependent function and product kinds.

This chapter is under construction ....

[1] As we shall see in the development, the propagation of information as sketched here is managed through the use of singleton kinds.