# *Phantom Types and Subtyping*

Matthew Fluet
*Cornell University*
*Ithaca, NY 14853 USA*
(*e-mail:* fluet@cs.cornell.edu)

Riccardo Pucella
*Northeastern University*
*Boston, MA 02115 USA*
(*e-mail:* riccardo@ccs.neu.edu)

**Abstract**

We investigate a technique from the literature, called the phantom-types technique, that uses parametric polymorphism, type constraints, and unification of polymorphic types to model a subtyping hierarchy. Hindley-Milner type systems, such as the one found in Standard ML, can be used to enforce the subtyping relation, at least for first-order values. We show that this technique can be used to encode any finite subtyping hierarchy (including hierarchies arising from multiple interface inheritance). We formally demonstrate the suitability of the phantom-types technique for capturing first-order subtyping by exhibiting a type-preserving translation from a simple calculus with bounded polymorphism to a calculus embodying the type system of SML.

## 1 Introduction

It is well known that traditional type systems, such as the one found in Standard ML (Milner *et al.*, 1997), with parametric polymorphism and type constructors, can be used to capture program properties beyond those naturally associated with a Hindley-Milner type system (Milner, 1978). For concreteness, let us review a simple example, due to Leijen and Meijer (1999). Consider a type of atoms, either booleans or integers, that can be easily represented with an algebraic datatype:

```
datatype atom = I of int | B of bool.
```

There are a number of operations that we may perform on such atoms (see Figure 1(a)). When the domain of an operation is restricted to only one kind of atom, as with `conj` and `double`, a run-time check must be made and an error or exception reported if the check fails.

One aim of static type checking is to reduce the number of run-time checks by catching type errors at compile time. Of course, in the example above, the SML type system does not consider `conj (mkInt 3, mkBool true)` to be ill-typed; evaluating this expression will simply raise a run-time exception.

If we were working in a language with subtyping, we would like to consider integer atoms and boolean atoms as distinct subtypes of the general type of atoms and use

```
datatype atom = I of int | B of bool


fun mkInt (i:int):atom = I (i)
fun mkBool (b:bool):atom = B (b)
fun toString (v:atom):string =
  (case v
    of I (i) => Int.toString (i)
     | B (b) => Bool.toString (b))
fun double (v:atom):atom =
  (case v
    of I (i) => I (i * 2)
     | _ => raise Fail "type mismatch")
fun conj (v1:atom,
          v2:atom):atom =
  (case (v1,v2)
    of (B (b1), B (b2)) => B (b1 andalso b2)
     | _ => raise Fail "type mismatch")
```

(a) Unsafe operations

```
datatype atom = I of int | B of bool
datatype 'a safe_atom = W of atom

fun mkInt (i:int):int safe_atom = W (I (i))
fun mkBool (b:bool):bool safe_atom = W (B (b))
fun toString (v:'a safe_atom):string =
  (case v
    of W (I (i)) => Int.toString (i)
     | W (B (b)) => Bool.toString (b))
fun double (v:int safe_atom):int safe_atom =
  (case v
    of W (I (i)) => W (I (i * 2))
     | _ => raise Fail "type mismatch")
fun conj (v1:bool atom,
          v2:bool atom):bool atom =
  (case (v1,v2)
    of (W (B (b1)), W (B (b2))) =>
            W (B (b1 andalso b2))
     | _ => raise Fail "type mismatch")
```

(b) Safe operations

Fig. 1. Atom operations

these subtypes to refine the types of the operations. Then the type system would report, at compile time, a type error in the expression `double (mkBool false)`. Fortunately, we can write the operations in a way that utilizes the SML type system to do just this. We introduce a new datatype that represents "safe" atoms, which is a simple wrapper around the datatype for atoms:

```
datatype 'a safe_atom = W of atom
```

and constrain the types of the operations (see Figure 1(b)). We use the superfluous type variable `'a` in the datatype definition to encode information about the kind of atom. (Because instantiations of this type variable do not contribute to the run-time representation of atoms, it is called a *phantom type*.) The type `int safe_atom` is used to represent integer atoms and `bool safe_atom` is used to represent boolean atoms. Now, the expression `conj (mkInt 3, mkBool true)` results in a compile-time type error, because the types `int safe_atom` and `bool safe_atom` do not unify. (Observe that our use of `int` and `bool` as phantom types is arbitrary; we could have used any two types that do not unify to make the integer versus boolean distinction.) On the other hand, both `toString (mkInt 3)` and `toString (mkBool true)` are well-typed; the `toString` operation can be applied to any atom. Note that we had to wrap the `atom` type in another datatype; the next section will explain why.[1]

The example above used a datatype as the representation of values manipulated by "unsafe" operations, and a wrapped version of the datatype to enforce safety. However, the underlying representation need not be a datatype. Consider a common instance of the problem, where we wish is to manipulate operating-system values,

[1] We could have simply defined `safe_atom` as:

```
datatype 'a safe_atom = I of int | B of bool
```

but for the sake of uniformity with the techniques presented in the next section, we use the slightly more verbose wrapping using a `W` constructor.

```
type sock = Word32.word



fun makeUDP (addr:string):sock =
  ffiMakeUDP (addr)
fun makeTCP (addr:string):sock =
  ffiMakeTCP (addr)

fun sendUDP (s:sock,text:string):unit =
  ffiSendUDP (s,text)
fun sendTCP (s:sock,text:string):unit =
  ffiSendTCP (s,text)

fun close (s:sock):unit =
  ffiClose (s)
```

(a) Unsafe operations

```
datatype udp = UDP
datatype tcp = TCP
datatype 'a safe_sock = W of Word32.word

fun makeUDP (addr:string):udp safe_sock =
  W (ffiMakeUDP (addr))
fun makeTCP (addr:string):tcp safe_sock =
  W (ffiMakeTCP (addr))

fun sendUDP (s:udp safe_sock,text:string):unit =
  (case s of W (w) => ffiSendUDP (w,text))
fun sendTCP (s:tcp safe_sock,text:string):unit =
  (case s of W (w) => ffiSendTCP (w,text))

fun close (s:'a safe_sock):unit =
  (case s of W (w) => ffiClose (w))
```

(b) Safe operations

Fig. 2. Socket operations

such as sockets. These are typically accessed via a foreign-function interface and they are typically represented by a 32-bit integer value (either representing a pointer or a handle in a table kept by the operating system). A number of primitive operations are provided through the foreign-function interface for handling those sockets (see Figure 2(a)). However, while the SML representation of a socket is just a 32-bit integer, the operating system often distinguishes internally between different kinds of sockets, for instance, between UDP sockets and TCP sockets, and operations specific to UDP sockets cause run-time exceptions (at the operating-system level) when supplied with a TCP socket. For instance, the operation `sendUDP` expects a UDP socket and a string to send on the socket. This is exactly the kind of check that occurs in the atom example above, except it is performed automatically by the operating system rather than the code. Other operations, such as `close`, work with all sockets, and therefore, there is an implicit subtyping relation among sockets, UDP sockets, and TCP sockets. We can enforce the appropriate use of sockets statically by defining new types:

```
datatype udp = UDP
datatype tcp = TCP
datatype 'a safe_sock = W of Word32.word
```

and constraining the types of the operations appropriately (see Figure 2(b)). Note that we again use a superfluous type variable in the definition of the type `safe_sock` to allow us to constrain the type of the operations. We can now supply appropriate types to versions of safe operations on sockets. (Note once again that we had to wrap the `Word32.word` type in a datatype.)

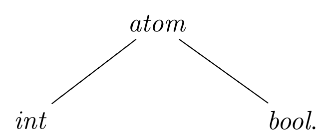
This is the essence of the technique explored in this paper: using a free type variable to encode subtyping information for first-order values, and using an SML-like type system to enforce the subtyping on those values. (We focus on first-order subtyping in this paper; Section 4 explains why.) This "phantom types" technique, where user-defined restrictions are reflected in the constrained types of values and functions, underlies many interesting uses of type systems. It has been used to derive early implementations of extensible records (Wand, 1987; Rémy, 1989; Burton, 1990), to provide a safe and flexible interface to the Network Socket API (Reppy, 1996), to interface to COM components (Finne *et al.*, 1999), to type embedded compiler expressions (Leijen & Meijer, 1999; Elliott *et al.*, 2000), to record sets of effects in type-and-effect type systems (Pessaux & Leroy, 1999), to embed a representation of the C type system in SML (Blume, 2001), and to encode data structure invariants (Fluet & Pucella, 2005).

This paper makes a number of contributions to the extant literature on phantom types. The first contribution is to describe a general procedure for applying the phantom-types technique to subtyping, generalizing all the known uses of phantom types of which we are aware. This procedure relies on an appropriate encoding of the subtyping hierarchy. We study different classes of encodings for different kinds of hierarchies. Next, we formalize this use of phantom types and prove its correctness. We present a type-preserving translation from a calculus with subtyping to a calculus with let-bounded polymorphism, using the procedure described earlier. The kind of subtyping that can be captured turns out to be an interesting variant of bounded polymorphism (Cardelli *et al.*, 1994), with a very restricted subsumption rule.

This paper is structured as follows. In the next section, we describe a simple recipe for deriving an interface enforcing a given subtyping hierarchy. The interface is parameterized by an encoding, via phantom types, of the subtyping hierarchy. In Section 3, we examine different encodings for hierarchies. We also address the issue of extensibility of the encodings. In Section 4, we extend the recipe to capture a limited form of bounded polymorphism. In Section 5, we formally define the kind of subtyping captured by our encodings by giving a simple calculus with subtyping and showing that our encodings provide a type-preserving translation to a variant of the Damas-Milner calculus, embodying the essence of the SML type system. We conclude with some problems inherent to the approach and a consideration of future work. The formal details of the calculi we introduce in Section 5 as well as the proofs of our results can be found in the appendices.

## 2 From Subtyping to Polymorphism

The examples in the introduction has the following features: an underlying *base type* of values (the original type `atom` and the type `Word32.word` for sockets), a set of primitive operations on values of the base type, and *sorts* of this base type that correspond to the sensible domains of the operations. The sorts of the base type form a hierarchy capturing the subtyping inherent in the sorts. The subtyping hierarchy corresponding to the atom example is as follows:




(We assume there is a sort corresponding to the base type as a whole, always the top of the hierarchy, capturing the intuition that every sort is a subtype of the base type.) The subtyping hierarchy is modeled by assigning a type to every sort in the hierarchy. For instance, integer atoms with sort *int* are encoded by the SML type `int safe_atom`. The appropriate use of polymorphic type variables in the type of an operation indicates the maximal type in the domain of the operation. For instance, the operation `toString` has the conceptual type *atom* `-> string` which is encoded by the SML type `'a safe_atom -> string`. The key observation is the use of type unification to enforce the subtyping hierarchy: an `int safe_atom` can be passed to a function expecting an `'a safe_atom`, because these types unify.

In this section, we show by means of an example, given a base type $\tau_b$, a set of sorts of $\tau_b$ (forming a hierarchy), and operations expressed in terms of the sorts of $\tau_b$, how to derive:

- a safe SML signature which uses phantom types to encode the subtyping between sorts, and
- a safe implementation from the unsafe implementation.

By safety here, we mean that the interface guarantees that no primitive operation is ever supplied a value outside its domain; we return to this point in Section 2.2, and make this guarantee precise in Section 5.

All values share the same underlying representation (the base type $\tau_b$) and each operation has a single implementation that acts on this underlying representation. The imposed subtyping captures restrictions that arise because of some external knowledge about the semantics of the operations; intuitively, it captures a "real" subtyping relationship that is not exposed by the representation of the abstract type.

We must emphasize at this point that we are concerned only with subtyping of values passed to primitive operations, where the values are base values as opposed to higher-order values such as functions. Therefore, we are interesting in first-order subtyping only. Of course, since SML is higher-order, we must say something about the subtyping on higher-order values induced by the subtyping on the base values. The subtyping relation on higher-order values will turn out to be severely restricted. We return to this point in Section 4.

### *2.1 The Safe Interface*

We first consider deriving the safe interface. The new interface defines a type `'a` $\tau$ corresponding to the base type $\tau_b$. The instantiations of the type variable `'a` will be used to encode sort information. We require an encoding $\langle\sigma\rangle$ of each sort $\sigma$ in the hierarchy; this encoding should yield a type in the underlying SML type system, with the property that $\langle\sigma_1\rangle$ unifies with $\langle\sigma_2\rangle$ if and only if $\sigma_1$ is a subtype of $\sigma_2$ in the hierarchy. An obvious issue is that we want to use unification (a symmetric relation) to capture subtyping (an asymmetric relation). The simplest approach is to use two encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ defined over all the sorts in the hierarchy. A *value* of sort $\sigma$ will be assigned a type $\langle\sigma\rangle_C\ \tau$. We call $\langle\sigma\rangle_C$ the *concrete* encoding of $\sigma$, and we assume that it uses only ground types (i.e., no type variables). In order to restrict the domain of an operation to the set of values that are subtypes of a sort $\sigma$, we use $\langle\sigma\rangle_A$, the *abstract* encoding of $\sigma$. In order for the underlying type system to enforce the subtyping hierarchy, we require the encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ to be *respectful* by satisfying the following property:

$$\text{for all } \sigma_1 \text{ and } \sigma_2,\ \langle\sigma_1\rangle_C \text{ matches } \langle\sigma_2\rangle_A \text{ iff } \sigma_1 \leq \sigma_2\ .$$

For example, the encodings used in the introduction are respectful:

$$\begin{array}{rclrcl} \langle atom\rangle_A & \triangleq & \texttt{'a} & \langle atom\rangle_C & \triangleq & \texttt{unit} \\ \langle int\rangle_A & \triangleq & \texttt{int} & \langle int\rangle_C & \triangleq & \texttt{int} \\ \langle bool\rangle_A & \triangleq & \texttt{bool} & \langle bool\rangle_C & \triangleq & \texttt{bool}. \end{array}$$

The utility of the phantom-types technique relies on being able to find respectful encodings for subtyping hierarchies of interest.

To allow for matching, the abstract encoding will introduce free type variables. Since, in a Hindley-Milner type system, a type cannot contain free type variables, the abstract encoding will be part of the larger type scheme of some polymorphic function operating on values of appropriate sorts. This leads to some restrictions on when we should constrain values by concrete or abstract encodings. For the time being, we will restrict ourselves to using concrete encodings in all covariant type positions and using abstract encodings in most contravariant type positions. It is fairly easy to see that if we do not impose this restriction, then we can assign type to functions that break the desired subtyping invariants. For example, suppose we added the following function to our collection of "safe" atom operations:

```
fun randAtom ():'a safe_atom =
  (case rand()
    of 0 => W (B (false))
     | 1 => W (B (true))
     | i => W (I (i))).
```

Note that `randAtom` is assigned the type `unit -> 'a safe_atom`, which appears to be consistent with the fact that `randAtom` returns some subtype of the sort *atom*. However, the expression `conj (randAtom (), randAtom ())` is considered well-typed, but its evaluation may raise a run-time exception. Intuitively, the issue with returning a value constrained by an abstract encoding is that we are trying to impose a restriction on the behavior of the function based on the types of future uses of the returned value; such type-directed behavior is not supported in a language like SML. We will return to this issue in Section 4. Another consequence of having the abstract encoding be part of a larger type scheme that binds the free variables in prenex position is that the subtyping is resolved not at the point of function application, but rather at the point of *type application*, when the type variables are instantiated. We postpone a discussion of this important point to Section 4, where we extend our recipe to account for a form of bounded polymorphism.

Consider again the atom example from the introduction. Assume we have encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ for the hierarchy and a structure `Atom` implementing the "unsafe"

```
signature ATOM = sig
  type atom
  val mkInt: int -> atom
  val mkBool: bool -> atom
  val toString: atom -> string
  val double: atom -> atom
  val conj: atom * atom -> atom
end
```

(a) Unsafe interface

```
signature SAFE_ATOM = sig
  type 'a safe_atom
  val mkInt: int -> ⟨int⟩C safe_atom
  val mkBool: bool -> ⟨bool⟩C safe_atom
  val toString: ⟨atom⟩A safe_atom -> string
  val double: ⟨int⟩A safe_atom -> ⟨int⟩C safe_atom
  val conj: ⟨bool⟩A safe_atom * ⟨bool⟩A safe_atom
          -> ⟨bool⟩C safe_atom
end
```

(b) Safe interface

Fig. 3. Interfaces for atoms

```
structure SafeAtom1 :> SAFE_ATOM = struct
  type 'a safe_atom = Atom.atom
  val mkInt = Atom.mkInt
  val mkBool = Atom.mkBool
  val toString = Atom.toString
  val double = Atom.double
  val conj = Atom.conj
end
```

(a) Opaque signature

```
structure SafeAtom2 : SAFE_ATOM = struct
  datatype 'a safe_atom = W of Atom.atom
  fun int (i) = W (Atom.mkInt (i))
  fun bool (b) = W (Atom.mkBool (b))
  fun toString (W v) = Atom.toString (v)
  fun double (W v) = W (Atom.double (v))
  fun conj (W b1, W b2) = W (Atom.conj (b1,b2))
end
```

(b) Datatype declaration

Fig. 4. Two implementations of the safe interface for atoms

operations, with the signature `ATOM` given in Figure 3(a). Deriving an interface using the recipe above, we get the safe signature given in Figure 3(b).[2]

### 2.2 The Safe Implementation

We must now derive an implementation corresponding to the safe signature. We need a type `'a` $\tau$ isomorphic to $\tau_b$ such that the type system considers $\tau_1$ and $\tau_2$ equivalent when $\tau_1\ \tau$ and $\tau_2\ \tau$ are equivalent. We can then constrain the types of values and operations using $\langle\sigma\rangle_C\ \tau$ and $\langle\sigma\rangle_A\ \tau$, as indicated above. There are two ways of enforcing this equivalence in SML:

1. We can use an abstract type at the module system level, as shown in Figure 4(a). The use of an opaque signature is critical to get the required behavior in terms of type equivalence. The advantage of this method is that there is no overhead.
2. We can wrap the base type $\tau_b$ using a datatype declaration

   `datatype 'a` $\tau$ `= W of` $\tau_b$`.`

[2] The signature we use is fairly minimal. There are other functions that would be useful, and that are still safe with respect to our definition of safety that we use in Section 5. For instance, consider the following function:

```
fun f (b) = if b then SafeAtom.mkInt (3) else SafeAtom.mkBool (false).
```

This function does not type-check, since `SafeAtom.mkInt (3)` has type `int safe_atom` while `SafeAtom.mkBool (false)` has type `bool safe_atom` (assuming the concrete and abstract encodings defined earlier). What one wants here are *coercion functions*, that take values of sort *bool* or sort *int* and coerce them to the sort *atom*. This corresponding to adding a function `coerceToAtom` of type `'a safe_atom -> unit safe_atom` to the safe interface for atoms; the implementation of this function is simply the identity function. We will not use coercion functions in this paper, but they can be added without difficulty.

The type `'a` $\tau$ behaves as required, because the datatype declaration defines a generative type operator. However, we must explicitly convert values of the base type to and from `'a` $\tau$ to witness the isomorphism. This yields the implementation given in Figure 4(b).

Note that the equivalence requirement precludes the use of type abbreviations of the form `type 'a` $\tau$ `=` $\tau_b$ not restricted by an opaque signature, which generally define constant type functions. Moreover, in a language such as Haskell which does not provide abstract types at the module level, the second approach is the only one available. Therefore, for the sake of generality, we use the second approach throughout this paper, with the understanding that our discussion can be straightforwardly adapted to the first approach.

We should stress that the safe interface must ensure that the type `'a` $\tau$ is abstract—either through the use of opaque signature matching, or by hiding the value constructors of the type. Otherwise, it may be possible to create values that do not respect the subtyping invariants enforced by the encodings. For example, exposing the wrapper constructor allows a client to write the following, which type-checks, but violates the implicit subtyping:

```
val bogus = (W (Atom.mkInt 5)) : bool safe_atom
val bad = conj (bogus, bogus).
```

The evaluation of `conj (bogus, bogus)` will raise a run-time exception. This example demonstrates the subtle difference between the guarantee made by the SML type-system and the guarantee made by a library employing the phantom-types technique. While the former ensures that "a well-typed program won't go wrong," the latter ensures that "a well-typed client won't go wrong, provided the library is correctly implemented." The purpose of this section has been to better characterize what it means for a library to be "correctly implemented," while Section 5 will make this characterization precise.

We now have a way to derive a safe interface and implementation, by adding type information to a generic, unsafe implementation. In the next section, we show how to construct respectful encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ by taking advantage of the structure of the subtyping hierarchy.

## 3 Encoding Hierarchies

The framework presented in the previous section relies on having concrete and abstract encodings of the sorts in the subtyping hierarchy with the property that unification of the results of the encoding respects the subtyping relation. In this section, we describe how such encodings can be obtained. Different encodings are appropriate, depending on the characteristics of the subtyping hierarchy being encoded. We assume, for the purpose of this paper, that subtyping hierarchies are at least join-semilattices. These encodings assume that the subtyping relation is completely known *a priori*. We address the question of extensibility in Section 3.5.

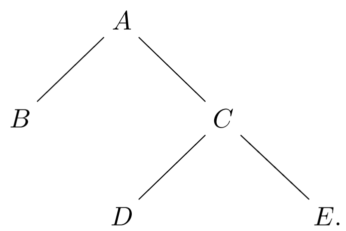
### *3.1 Tree Hierarchies*

For the time being, we restrict ourselves to finite subtyping hierarchies. The simplest case to handle is a tree hierarchy. Since this is the type of hierarchy that occurs in both examples discussed in the introduction (and, in fact, in all the examples we found in the literature on encoding subtyping hierarchies in a polymorphic type system), this encoding should be clear. The idea is to assign a type constructor to every subtype of a subtyping hierarchy. Assume we have an encoding $\langle\cdot\rangle_N$ assigning a distinct (syntactic) name to each entry in a subtyping hierarchy $(\Sigma, \leq)$. Hence, for each $\sigma \in \Sigma$, we define:

$$\texttt{datatype 'a } \langle\sigma\rangle_N \texttt{ = Irrelevant\_}\langle\sigma\rangle_N\texttt{.}$$

(The name of the data constructor is completely irrelevant, as we will never construct values of this type. It is required because SML does not allow the definition of a datatype with no constructors.)

For example, consider the following subtyping hierarchy (which is essentially the one used in the Sockets API described by Reppy (1996)):




We first define type constructors for every element of the hierarchy. We assume a reasonable name encoding $\langle\cdot\rangle_N$, such as $\langle A\rangle_N = \texttt{A}$, $\langle B\rangle_N = \texttt{B}$, etc. Hence, we have

```
datatype 'a A = Irrelevant_A
datatype 'a B = Irrelevant_B
```

and likewise for the other elements. The concrete encoding for an element of the hierarchy represents the path from the top of the hierarchy to the element itself. Hence,

$$\begin{aligned}
\langle A\rangle_C &\triangleq \texttt{unit A} \\
\langle B\rangle_C &\triangleq \texttt{(unit B) A} \\
\langle C\rangle_C &\triangleq \texttt{(unit C) A} \\
\langle D\rangle_C &\triangleq \texttt{((unit D) C) A} \\
\langle E\rangle_C &\triangleq \texttt{((unit E) C) A.}
\end{aligned}$$

For the corresponding abstract encoding to be respectful, we require the abstract encoding of $\sigma$ to unify with the concrete encoding of all the subtypes of $\sigma$. In other words, we require the abstract encoding to represent the prefix of the path leading to the element $\sigma$ in the hierarchy. We use a type variable to unify with any part of the path after the prefix we want to represent. Hence,

$$
\begin{array}{rcl}
\langle A\rangle_A & \triangleq & \texttt{'a}_1\ \texttt{A} \\
\langle B\rangle_A & \triangleq & \texttt{('a}_2\ \texttt{B) A} \\
\langle C\rangle_A & \triangleq & \texttt{('a}_3\ \texttt{C) A} \\
\langle D\rangle_A & \triangleq & \texttt{(('a}_4\ \texttt{D) C) A} \\
\langle E\rangle_A & \triangleq & \texttt{(('a}_5\ \texttt{E) C) A}.
\end{array}
$$

We can then verify, for example, that the concrete encoding of $D$ unifies with the abstract encoding of $C$, as required.

Note that $\langle\cdot\rangle_A$ requires every type variable $\texttt{'a}_i$ to be a fresh variable, unique in its context. This ensures that we do not inadvertently refer to any type variable bound in the context where we are introducing the abstractly encoded type. The following example illustrates the potential problem. Let $(\Sigma, \leq)$ be the subtyping hierarchy given above, over some underlying base type $\tau_b$. Suppose we wish to encode an operation of type $A$ * $A$ -> int with the understanding that a (different) subtype of $A$ may be passed for each of the arguments. The encoded type of the operation becomes $\langle A\rangle_A\ \tau$ * $\langle A\rangle_A\ \tau$ -> int (where 'a $\tau$ is the wrapped type of $\tau_b$ values) which should translate to ('a A) $\tau$ * ('b A) $\tau$ -> int. If we are not careful in choosing fresh type variables, we could generate the following type ('a A) $\tau$ * ('a A) $\tau$ -> int, corresponding to a function that requires two arguments of the same type, which is not the intended meaning. (The handling of introduced type variables is somewhat delicate; we address the issue in more detail in Section 4.)

It should be clear how to generalize the above discussion to concrete and abstract encodings for arbitrary finite tree hierarchies. Let $\top_\Sigma$ correspond to the root of the finite tree hierarchy. Define an auxiliary encoding $\langle\cdot\rangle_X$ which can be used to construct chains of type constructors. The encoding $\langle\sigma\rangle_X$ returns a function expecting the type to "attach" at the end of the chain

$$
\begin{array}{rcll}
\langle\top_\Sigma\rangle_X\ (t) & \triangleq & t\ \langle\top_\Sigma\rangle_N & \\
\langle\sigma\rangle_X\ (t) & \triangleq & \langle\sigma_{\text{parent}}\rangle_X\ (t\ \langle\sigma\rangle_N) & \text{where } \sigma_{\text{parent}} \text{ is the parent of } \sigma.
\end{array}
$$

Thus, in the example above, we have:

$$
\begin{array}{rcl}
\langle A\rangle_X\ (t) & \triangleq & t\ \texttt{A} \\
\langle B\rangle_X\ (t) & \triangleq & \texttt{(}t\ \texttt{B) A} \\
\langle C\rangle_X\ (t) & \triangleq & \texttt{(}t\ \texttt{C) A} \\
\langle D\rangle_X\ (t) & \triangleq & \texttt{((}t\ \texttt{D) C) A} \\
\langle E\rangle_X\ (t) & \triangleq & \texttt{((}t\ \texttt{E) C) A}.
\end{array}
$$

Using this auxiliary encoding, we can define the concrete and abstract encodings by supplying the appropriate type:

$$
\begin{array}{rcll}
\langle\sigma\rangle_C & \triangleq & \langle\sigma\rangle_X\ (\texttt{unit}) & \\
\langle\sigma\rangle_A & \triangleq & \langle\sigma\rangle_X\ (\texttt{'a}) & \text{where } \texttt{'a} \text{ is fresh.}
\end{array}
$$

### *3.2 Finite Powerset Lattices*

Not every subtyping hierarchy of interest is a tree. More general hierarchies can be used to model multiple interface inheritance in an object-oriented setting. Let us now examine more general subtyping hierarchies. We first consider a particular lattice that will be useful in our development. Recall that a lattice is a hierarchy where every set of elements has both a least upper bound and a greatest lower bound. Given a finite set $S$, we let the *powerset lattice* of $S$ be the lattice of all subsets of $S$, ordered by inclusion, written $(\wp(S), \subseteq)$. We now exhibit an encoding of powerset lattices.

Let $n$ be the cardinality of $S$ and assume an ordering $s_1, \ldots, s_n$ on the elements of $S$. We encode subset $X$ of $S$ as an $n$-tuple type, where the $i^{\text{th}}$ entry expresses that $s_i \in X$ or $s_i \notin X$. First, we introduce a datatype that roughly acts as a Boolean value at the level of types:

```
datatype 'a z = Irrelevant_z.
```

The encoding of an arbitrary subset of $S$ is given by:

$$\langle X \rangle_C \quad \triangleq \quad t_1 \texttt{*} \ldots \texttt{*} t_n \quad \text{where } t_i \triangleq \begin{cases} \texttt{unit} & \text{if } s_i \in X \\ \texttt{unit z} & \text{otherwise} \end{cases}$$

$$\langle X \rangle_A \quad \triangleq \quad t_1 \texttt{*} \ldots \texttt{*} t_n \quad \text{where } t_i \triangleq \begin{cases} \texttt{'a}_i & \text{if } s_i \in X \\ \texttt{'a}_i\ \texttt{z} & \text{otherwise.} \end{cases}$$

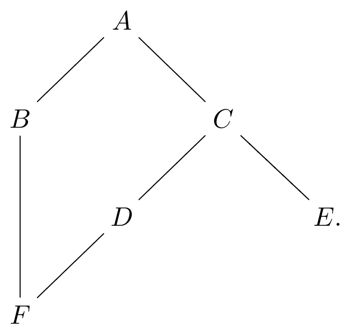
Note that $\langle \cdot \rangle_A$ requires every type variable $\texttt{'a}_i$ to be a fresh type variable, unique in its context. This ensures that we do not inadvertently refer to any type variable bound in the context where we are introducing the abstractly encoded type.

As an example, consider the powerset lattice of $\{1, 2, 3, 4\}$, which encodes into a four-tuple. We can verify, for example, that the concrete encoding for $\{2\}$, namely `(unit z * unit * unit z * unit z)`, unifies with the abstract encoding for $\{1, 2\}$, namely $(\texttt{'a}_1\ \texttt{*}\ \texttt{'a}_2\ \texttt{*}\ \texttt{'a}_3\ \texttt{z}\ \texttt{*}\ \texttt{'a}_4\ \texttt{z})$. On the other hand, the concrete encoding of $\{1, 2\}$, namely `(unit * unit * unit z * unit z)`, does not unify with the abstract encoding of $\{2, 3\}$, namely $(\texttt{'a}_1\ \texttt{z}\ \texttt{*}\ \texttt{'a}_2\ \texttt{*}\ \texttt{'a}_3\ \texttt{*}\ \texttt{'a}_4\ \texttt{z})$.

### *3.3 Embeddings*

The main reason we introduced powerset lattices is the fact that any finite hierarchy can be embedded in the powerset lattice of a set $S$. It is a simple matter, given a hierarchy $\Sigma'$ embedded in a hierarchy $\Sigma$, to derive an encoding for $\Sigma'$ given an encoding for $\Sigma$. Let $inj(\cdot)$ be the injection from $\Sigma'$ to $\Sigma$ witnessing the embedding and let $\langle \cdot \rangle_{C_\Sigma}$ and $\langle \cdot \rangle_{A_\Sigma}$ be the encodings for the hierarchy $\Sigma$. Deriving an encoding for $\Sigma'$ simply involves defining $\langle \sigma \rangle_{C_{\Sigma'}} \triangleq \langle inj(\sigma) \rangle_{C_\Sigma}$ and $\langle \sigma \rangle_{A_{\Sigma'}} \triangleq \langle inj(\sigma) \rangle_{A_\Sigma}$. It is straightforward to verify that if $\langle \cdot \rangle_{C_\Sigma}$ and $\langle \cdot \rangle_{A_\Sigma}$ are respectful encodings, so are $\langle \cdot \rangle_{C_{\Sigma'}}$ and $\langle \cdot \rangle_{A_{\Sigma'}}$. By the result above, this allows us to derive an encoding for an arbitrary finite hierarchy.

To give an example of embedding, consider the following subtyping hierarchy to be encoded:




Notice that this hierarchy can be embedded into the powerset lattice of $\{1,2,3,4\}$, via the injection function sending $A$ to $\{1,2,3,4\}$, $B$ to $\{1,2,3\}$, $C$ to $\{2,3,4\}$, $D$ to $\{2,3\}$, $E$ to $\{3,4\}$, and $F$ to $\{2\}$.

### *3.4 Other Encodings*

We have presented recipes for obtaining respectful encodings, depending on the characteristics of the subtyping hierarchy at hand. It should be clear that there are more general hierarchies than the ones presented here that can still be encoded, although the encodings quickly become complicated and *ad hoc*. It would be an interesting project to study in depth the theory of hierarchies encoding that seems to be lurking here. As an example, let us examine an encoding that generalizes the finite powerset lattice encoding to the (countably) infinite case, but where only the finite subsets of a countably infinite set are encoded. Therefore, this encoding is only useful when a program manipulates only values with sorts corresponding to finite subsets in the hierarchy. While *ad hoc*, this example is interesting enough to warrant a discussion.

Technically, the encoding is in the spirit of the finite powerset encoding. Let $S$ be a countably infinite set, and assume an ordering $s_1, s_2, \ldots$ of the elements of $S$. As in the finite case, we define a datatype

```
datatype 'a z = Irrelevant_z.
```

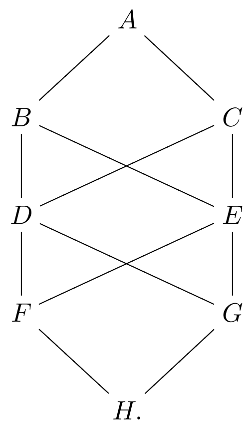
The encoding is given for *finite* subsets of $S$ by the following pair of encodings:

$$\begin{aligned}
\langle X\rangle_C &\triangleq (t_1 \texttt{ * } (t_2 \texttt{ * } (t_3 \texttt{ * } \ldots \texttt{ * } (t_n \texttt{ * 'a}) \ldots))) \\
&\quad \text{where } t_i \triangleq \begin{cases} \texttt{unit} & \text{if } s_i \in X \\ \texttt{unit z} & \text{otherwise} \end{cases} \\
&\quad\quad \text{and } n \text{ is the least index such that } X \subseteq \{s_1, \ldots, s_n\} \\
\langle X\rangle_A &\triangleq (t_1 \texttt{ * } (t_2 \texttt{ * } (t_3 \texttt{ * } \ldots \texttt{ * } (t_n \texttt{ * unit}), \ldots))) \\
&\quad \text{where } t_i \triangleq \begin{cases} \texttt{'a}_i & \text{if } s_i \in X \\ \texttt{'a}_i\texttt{ z} & \text{otherwise} \end{cases} \\
&\quad\quad \text{and } n \text{ is the least index such that } X \subseteq \{s_1, \ldots, s_n\}.
\end{aligned}$$

(As usual, with the restriction that the type variables $\texttt{'a}_1, \ldots, \texttt{'a}_n$ are fresh.) One can verify that this is indeed a respectful encoding of the finite elements of the infinite lattice.

Note that this use of a free type variable to be "polymorphic in the rest of the encoded value" is strongly reminiscent of the notion of a *row variable*, as originally used by Wand (1987) to type extensible records, and further developed by Rémy (1989). The technique was used by Burton (1990) to encode extensible records directly in a polymorphic type system. Recently, the same technique was used to represents sets of effects in type-and-effect systems (Pessaux & Leroy, 1999).

We have not focussed on the complexity or space-efficiency of the encodings. We have emphasized simplicity and uniformity of the encodings, at the expense of succinctness. For instance, deriving an encoding for a finite hierarchy by embedding it in a powerset lattice can lead to large encodings even when simpler encodings exist. Consider the following subtyping hierarchy:




The smallest powerset lattice in which this hierarchy can be embedded is the powerset lattice of a 6-element set; therefore, the encoding will require 6-tuples. On the other hand, it is not hard to verify that the following encoding respects the subtyping induced by this hierarchy. As before, we define a datatype

```
datatype 'a z = Irrelevant_z.
```

Consider the following encoding:

$\langle A\rangle_C \triangleq$ `(unit * unit)`
$\langle B\rangle_C \triangleq$ `(unit z * unit)`
$\langle C\rangle_C \triangleq$ `(unit * unit z)`
$\langle D\rangle_C \triangleq$ `((unit z) z * unit z)`
$\langle E\rangle_C \triangleq$ `(unit z * (unit z) z)`
$\langle F\rangle_C \triangleq$ `(((unit z) z) z * (unit z) z)`
$\langle G\rangle_C \triangleq$ `((unit z) z * ((unit z) z) z)`
$\langle H\rangle_C \triangleq$ `(((unit z) z) z * ((unit z) z) z).`

The abstract encoding is obtained by replacing every `unit` by a type variable `'a`, taken fresh, as usual.

It is possible to generate encodings for finite hierarchies that are in general more efficient than the encodings derived from the powerset lattice embeddings. One such encoding, which we now describe, uses a tuple approach just like the powerset lattice encoding. This encoding yields tuples whose size correspond to the width of the subtyping hierarchy being encoded, rather than the typically larger size of the smallest set in whose powerset lattice the hierarchy can be embedded. (The efficient encoding for the previous subtyping hierarchy is an instance of such a width encoding.)

Let $(\Sigma, \leq)$ be a hierarchy we wish to encode. The *width* of $\Sigma$ is the maximal size of sets of incomparable elements. Formally,

$$w(\Sigma) \triangleq \max\{|X| \mid X \subseteq \Sigma, \forall x, y \in \Sigma, (x \not\leq y \ \wedge \ y \not\leq x)\}.$$

The following proposition allows us to derive an encoding based on the width of the hierarchy.

*Proposition 3.1*
Let $\Sigma$ be a finite hierarchy, and $w$ be the width of $\Sigma$. There exists a function $l : \Sigma \to \mathbb{N}^w$ such that $x \leq y$ if and only if for $i = 1, \ldots, w$, $l(x)(i) \geq l(y)(i)$.

*Proof*
Choose $S = \{s_1, \ldots, s_w\}$ a subset of $\Sigma$ such that $S$ is a set of mutually incomparable elements of size $w$. We iteratively define a function $l' : \Sigma \to \mathbb{Q}$. We initially set $l'(\top_\Sigma) \triangleq (0, \ldots, 0)$, and for every $s_i$ in the set $S$,

$$l'(s_i) \triangleq (a_1, \ldots, a_w) \quad \text{where } a_k \triangleq \begin{cases} \frac{1}{2} & \text{if } i \neq k \\ 0 & \text{otherwise.} \end{cases}$$

Iteratively, for all elements $x \in \Sigma$ not assigned a value by $l'$, define the sets

$$x^{>} \triangleq \{y \in \Sigma \mid y \text{ is assigned a value by } l' \text{ and } y > x\}$$

and

$$x^{<} \triangleq \{y \in \Sigma \mid y \text{ is assigned a value by } l' \text{ and } y < x\}.$$

It is easy to verify that either $x^{>} \cap S \neq \varnothing$ or $x^{<} \cap S \neq \varnothing$, but not both (otherwise, there exists $y^{<} \in S$ and $y^{>} \in S$ such that $y^{>} > x > y^{<}$, and hence $y^{>} > y^{<}$, contradicting the mutual incomparability of elements of $S$). Define $l'(x) \triangleq (x_1, \ldots, x_n)$, where

$$x_i \triangleq \frac{\min_{y \in x^{<}}\{l'(y)(i)\} + \max_{y \in x^{>}}\{l'(y)(i)\}}{2}.$$

We can now define the function $l : \Sigma \to \mathbb{N}^w$ by simply rescaling the result of the function $l'$. Let $R$ be the sequence of all the rational numbers that appear in some tuple position in the result $l'(x)$ for some $x \in \Sigma$, ordered by the standard order on $\mathbb{Q}$. For $r \in R$, let $i(r)$ be the index of the rational number $r$ in $R$. Define the function $l$ by $l(x) \triangleq (i(l'(x)(1)), \ldots, i(l'(x)(w)))$. It is straightforward to verify that the property in the proposition holds for this function. □

We can use Proposition 3.1 to encode elements of a finite hierarchy $\Sigma$. Define a datatype

```
datatype 'a z = Irrelevant_z.
```

(As usual, the data constructor name is irrelevant). We encode an element into a tuple of size $w$, the width of $\Sigma$. Assume we have a labeling of the elements of $\Sigma$ by a function $l$ as given by Proposition 3.1. Essentially, $l$ will indicate the nesting of the above type constructor in the encoding. Formally,

$$\begin{aligned}\langle X\rangle_C &\triangleq \underbrace{\texttt{(}\ldots\texttt{(unit z)}\ldots\texttt{z)}}_{l(x)(1)} \texttt{*}\ldots\texttt{*} \underbrace{\texttt{(}\ldots\texttt{(unit z)}\ldots\texttt{z)}}_{l(x)(w)}\\ \langle X\rangle_A &\triangleq \underbrace{\texttt{(}\ldots\texttt{('a}_1\texttt{ z)}\ldots\texttt{z)}}_{l(x)(1)} \texttt{*}\ldots\texttt{*} \underbrace{\texttt{(}\ldots\texttt{('a}_w\texttt{ z)}\ldots\texttt{z)}}_{l(x)(w)}.\end{aligned}$$

(As usual, each $\texttt{'a}_i$ in $\langle\cdot\rangle_A$ is fresh.)

The fact that there are different encodings for the same hierarchy raises an obvious question: how do we determine the best encoding to use for a given hierarchy in a given situation? There are interesting problems here, for instance, lower and upper bounds on optimal encodings for hierarchies, as well as measurement metrics for comparing different encodings. We know of no work directly addressing these issues.

### *3.5 Extensibility*

One aspect of encodings we have not yet discussed is that of extensibility. Roughly speaking, extensibility refers to the possibility of adding new elements to the subtyping hierarchy after a program has already been written. One would like to avoid having to rewrite the whole program taking the new subtyping hierarchy into account. This is especially important in the design of libraries, where the user may need to extend the kind of data that the library handles, without changing the provided interface. For example, we can easily adapt the subtyping hierarchy of the atom example to accommodate strings by extending the `SAFE_ATOM` signature with

```
val mkString: string -> ⟨str⟩_C safe_atom
val concat: ⟨str⟩_A safe_atom * ⟨str⟩_A safe_atom -> ⟨str⟩_C safe_atom
```

and taking

$$\langle str\rangle_A \triangleq \texttt{string} \qquad \langle str\rangle_C \triangleq \texttt{string}.$$

Note that while the implementations of the `Atom` and `SafeAtom` structures require changes, no existing client of the `SAFE_ATOM` signature requires any changes. In this section, we examine the extensibility of the encodings we have presented.

Looking at the encodings of Section 3, it should be clear that the only immediately extensible encodings are the tree encodings in Section 3.1. In such a case, adding a new sort $\sigma_{\text{new}}$ as an immediate subtype of a given sort $\sigma_{\text{parent}}$ in the tree simply requires the definition of a new datatype:

```
datatype 'a ⟨σ_new⟩_N = Irrelevant_⟨σ_new⟩_N.
```

We assume a naming function $\langle\cdot\rangle_N$ extended to include $\sigma_{\text{new}}$. One can check that the abstract and concrete encodings of the original elements of the hierarchy are not changed by the extension—since the encoding relies on the path to the elements. The concrete and abstract encodings of the new subtype $\sigma_{\text{new}}$ is just the path to $\sigma_{\text{new}}$, as expected.

The powerset lattice encodings and their embeddings are not so clearly extensible. Indeed, in general, it does not seem possible to arbitrarily extend an encoding of a subtyping hierarchy that contains "join elements" (that is, a sort which is a subtype of at least two otherwise unrelated sorts, related to *multiple inheritance* in object-oriented programming). However, as long as the extension takes the form of adding new sub-sorts to a single sort in the hierarchy, it is possible to extend any subtyping hierarchy in a way that does not invalidate the original encoding of the hierarchy. As an illustration, consider again the simplest case, where we want to add a new sort $\sigma_{\text{new}}$ to an existing hierarchy, where $\sigma_{\text{new}}$ is an immediate subtype of the sort $\sigma_{\text{parent}}$. Assume that the existing hierarchy is encoded using the finite powerset encoding of Section 3.2. Observe that in the lattice encodings, the encoding of a sort $\sigma$ corresponding to subset $\{s_{i_1}, \ldots, s_{i_n}\}$ contains a `unit` in the tuple positions corresponding to $s_{i_1}$ through $s_{i_n}$, and `unit z` in the other positions. To encode the new sort $\sigma_{\text{new}}$, we can simply create a new type

$$\texttt{datatype 'a } \langle\sigma_{\text{new}}\rangle_N \texttt{ = Irrelevant\_}\langle\sigma_{\text{new}}\rangle_N$$

as in the case of tree encodings, and, for the concrete encoding, replace every `unit` in the concrete encoding of $\sigma_{\text{parent}}$ by `unit` $\langle\sigma_{\text{new}}\rangle_N$. For the abstract encoding of $\sigma_{\text{new}}$, we replace every `unit` in the concrete encoding by a (fresh) type variable. One can verify that indeed the resulting encoding is respectful of the subtyping hierarchy with the additional sort $\sigma_{\text{new}}$.

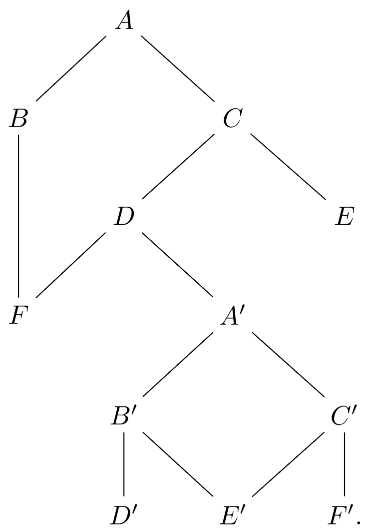
We can easily generalize this procedure of adding a new sort to an existing hierarchy encoded using a powerset lattice encoding to the case where we want to add a whole hierarchy as subtypes of a single sort in an existing hierarchy. Here is an outline of the general approach, of which the above is a special case. Let $\Sigma$ be a powerset lattice over a set $S$ of cardinality $n$, and let $\sigma_{\text{parent}}$ be an element of $\Sigma$ we want to extend by another hierarchy $\Sigma_{\text{new}}$; that is, all elements of $\Sigma_{\text{new}}$ are subtypes of $\sigma_{\text{parent}}$ and incomparable to other elements of $\Sigma$. Assume that $\Sigma$ is encoded via a lattice embedding encoding $\langle\cdot\rangle_C, \langle\cdot\rangle_A$, and that $\Sigma_{\text{new}}$ is encoded via some encoding $\langle\cdot\rangle_{C_{\text{new}}}, \langle\cdot\rangle_{A_{\text{new}}}$. We can extend the encoding for $\Sigma$ over the elements $\sigma' \in \Sigma_{\text{new}}$:

$$\langle\sigma'\rangle_C \triangleq t_1 \texttt{ * } \ldots \texttt{ * } t_n \quad \text{where } t_i \triangleq \begin{cases} \langle\sigma'\rangle_{C_{\text{new}}} & \text{if } s_i \in \sigma_{\text{parent}} \\ \texttt{unit z} & \text{otherwise} \end{cases}$$

$$\langle\sigma'\rangle_A \triangleq t_1 \texttt{ * } \ldots \texttt{ * } t_n \quad \text{where } t_i \triangleq \begin{cases} \langle\sigma'\rangle_{A_{\text{new}}} & \text{if } s_i \in \sigma_{\text{parent}} \\ \texttt{'a}_i\texttt{ z} & \text{otherwise.} \end{cases}$$

(As usual, each $\texttt{'a}_i$ in $\langle\cdot\rangle_A$, including the type variables in $\langle\sigma'\rangle_{A_{\text{new}}}$, is fresh.) Again, such an encoding is easily seen as being respectful of the extended subtyping hierarchy. The above scheme generalizes in a straightforward way to encodings via lattice embeddings and to the countable lattice encoding of Section 3.4.

As an example, we extend the hierarchy of Section 3.3 as follows:




The complete concrete encoding is given by:

$$\langle A' \rangle_{C_{\text{new}}} \triangleq \texttt{(unit * unit)}$$
$$\langle B' \rangle_{C_{\text{new}}} \triangleq \texttt{(unit * unit z)}$$
$$\langle C' \rangle_{C_{\text{new}}} \triangleq \texttt{(unit z * unit)}$$
$$\langle D' \rangle_{C_{\text{new}}} \triangleq \texttt{(unit * (unit z) z)}$$
$$\langle E' \rangle_{C_{\text{new}}} \triangleq \texttt{(unit z * unit z)}$$
$$\langle F' \rangle_{C_{\text{new}}} \triangleq \texttt{((unit z) z * unit)}$$

and

$$\langle A \rangle_C \triangleq \texttt{(unit * unit * unit * unit)}$$
$$\langle B \rangle_C \triangleq \texttt{(unit * unit * unit * unit z)}$$
$$\langle C \rangle_C \triangleq \texttt{(unit z * unit * unit * unit)}$$
$$\langle D \rangle_C \triangleq \texttt{(unit z * unit * unit * unit z)}$$
$$\langle E \rangle_C \triangleq \texttt{(unit z * unit z * unit * unit)}$$
$$\langle F \rangle_C \triangleq \texttt{(unit z * unit * unit z * unit z)}$$

$$\langle A' \rangle_C \triangleq \texttt{(unit z * (unit * unit) * (unit * unit) * unit z)}$$
$$\langle B' \rangle_C \triangleq \texttt{(unit z * (unit * unit z) * (unit * unit z) * unit z)}$$
$$\langle C' \rangle_C \triangleq \texttt{(unit z * (unit z * unit) * (unit z * unit) * unit z)}$$
$$\langle D' \rangle_C \triangleq \texttt{(unit z * (unit * (unit z) z) * (unit * (unit z) z) * unit z)}$$
$$\langle E' \rangle_C \triangleq \texttt{(unit z * (unit z * unit z) * (unit z * unit z) * unit z)}$$

$\langle F' \rangle_C \triangleq$ (unit z * ((unit z) z * unit) * ((unit z) z * unit) * unit z).

The abstract encoding is obtained by replacing every unit by a type variable 'a, taken fresh, as usual.

The interesting thing to notice about the above development is that although extensions are restricted to a single sort (i.e., we can only subtype one given sort), the extension can itself be an arbitrary lattice. As we already pointed out, it does not seem possible to describe a general extensible encoding that supports subtyping two different sorts at the same time (multiple inheritance). In other words, to adopt an object-oriented perspective, we cannot multiply-inherit from multiple sorts but we can single-inherit into an arbitrary lattice, which can use multiple inheritance locally.

## 4 Encoding a More General Form of Subtyping

As mentioned in Section 3, the handling of type variables is somewhat delicate. In this section, we revisit this issue. We show, by approaching the problem from a different perspective, how we can encode using phantom types a more general form of subtyping than simply subtyping at function arguments. We believe that this is the right setting to understand the ad-hoc restrictions given previously.

If we allow common type variables to be used across abstract encodings, then we can capture a form of *bounded polymorphism* as in $F_{<:}$ (Cardelli *et al.*, 1994). Bounded polymorphism is a typing discipline which extends both parametric polymorphism and subtyping. From parametric polymorphism, it borrows type variables and universal quantification; from subtyping, it allows one to set bounds on quantified type variables. For example, one can guarantee that the argument and return types of a function are the same and a subtype of $\sigma$, as in $\forall\alpha{<:}\sigma(\alpha \to \alpha)$.[3] Similarly, one can guarantee that two arguments have the same type that is a subtype of $\sigma$, as in $\forall\alpha{<:}\sigma(\alpha \times \alpha \to \sigma)$. Notice that neither function can be written in a language that supports only subtyping. In short, bounded polymorphism lets us be more precise when specifying subtyping occurring in functions.

The recipe we gave in Section 2 shows that we can capture subtyping using parametric polymorphism and restrictions on type equivalence. It turns out that we can capture a form of bounded polymorphism by adapting this procedure. As an example, consider the type $\forall\beta{<:}\sigma_1(\beta \times \sigma_2 \to \beta)$. Let the "safe" interface use types of the form $\alpha\ \tau$. Since $\beta$ stands for a subtype of $\sigma_1$, we let $\phi_\beta \triangleq \langle\sigma_1\rangle_A$, the abstract encoding of the bound. We then translate the type as we did in Section 2, but replace occurrences of the type variable $\beta$ by $\phi_\beta$ instead of applying $\langle\cdot\rangle_A$ repeatedly. This lets us *share* the type variables introduced by $\langle\sigma_1\rangle_A$. Hence, we get the type $\phi_\beta\ \tau \times \langle\sigma_2\rangle_A\ \tau \to \phi_\beta\ \tau$. This procedure in fact generalizes that of Section 2: we can convert all the subtyping into bounded polymorphism. More precisely, if a function expects an argument of a sort that is a subtype of $\sigma$, we can introduce a fresh type variable for that argument and bind it by $\sigma$. For example, the type above can be

[3] In this section, we freely use a $F_{<:}$-like notation for expressions.

rewritten as $\forall\beta{<:}\sigma_1(\forall\gamma{<:}\sigma_2(\beta \times \gamma \to \beta))$, and encoded as $\phi_\beta\ \tau \times \phi_\gamma\ \tau \to \phi_\beta\ \tau$, where $\phi_\beta \triangleq \langle\sigma_1\rangle_A$ and $\phi_\gamma \triangleq \langle\sigma_2\rangle_A$.

As one might expect, this technique does not generalize to full $F_{<:}$. For example, it is not clear how to encode types using bounded polymorphism where the bound on a type variable uses a type variable, such as a function `f` with type $\forall\alpha{<:}\sigma(\forall\beta{<:}\alpha(\alpha \times \beta \to \alpha))$. Encoding this type as $\phi_\alpha\ \tau \times \phi_\beta\ \tau \to \phi_\alpha\ \tau$, where $\phi_\alpha \triangleq \langle\sigma\rangle_A$ and $\phi_\beta \triangleq \langle\alpha\rangle_A$, fails, because we have no definition of $\langle\alpha\rangle_A$. Essentially, we need a different encoding of $\beta$ for each instantiation of $\alpha$ at each application of `f`, something that cannot be accommodated by a single encoding of the type at the definition of `f`.

However, the most important restriction on the kind of bounded polymorphism that can be handled in a straightforward way is due to the fact that we are capturing this form of subtyping using SML, which uses *prenex* parametric polymorphism. This means, for instance, that we cannot encode first-class polymorphism, such as a function `g` with type $\forall\alpha{<:}\sigma_1(\alpha \to (\forall\beta{<:}\sigma_2(\beta \to \beta)))$. Applying the technique yields a type $\phi_\alpha\ \tau \to \phi_\beta\ \tau \to \phi_\beta\ \tau$ where $\phi_\alpha$ and $\phi_\beta$ contain free type variables. A Hindley-Milner style type system requires quantification over these variables in prenex position, which doesn't match the intuition of the original type. Thus, because we are translating into a language with prenex polymorphism, it seems we can only capture bounded polymorphism that is itself in prenex form.

One consequence of being restricted to prenex bounded polymorphism is that we cannot account for the general subsumption rule found in $F_{<:}$. Instead, we require all subtyping to occur at type application. This is why we can convert all subtyping into bounded polymorphism, as we did above. By introducing type variables for each argument, we move the resolution of the subtyping to the point of type application (when we instantiate the type variables). The following example may illustrate this point. In $F_{<:}$ with first-class polymorphism, we can write a function `app1` with type $(\forall\alpha{<:}\sigma_1(\alpha \to \sigma_2)) \to \sigma_2 \times \sigma_2$ that applies a function to two values, `v1` of type $\sigma_1$ and `v2` of type $\sigma_2 \leq \sigma_1$, using an SML-like syntax that should be self-explanatory:

```
local
  val v1 : σ₁ = ...
  val v2 : σ₂ = ...
in
  fun app1 (f : ∀α ≤ σ₁(α → σ₂)) : σ₂ × σ₂ =
    (f [σ₁] v1, f [σ₂] v2)
end.
```

This definition of `app1` type-checks when we apply the argument function to $\sigma_1$ and then to `v1` and we apply the argument function to $\sigma_2$ (using subsumption at type application) and then to `v2`. But, as we argued above, we cannot encode first-class polymorphism. An alternative version, `app2`, can be written in $F_{<:}$ with type $(\sigma_1 \to \sigma_2) \to \sigma_2 \times \sigma_2$:

```
local
  val v1 : σ1 = ...
  val v2 : σ2 = ...
in
  fun app2 (f : (σ1 → σ2)) : σ2 × σ2 =
    (f v1, f (v2 : σ1))
end.
```

This definition of app2 type-checks when we apply the argument function to v1 and we apply the argument function to v2 (using subsumption on v2, coercing from $\sigma_2$ to $\sigma_1$). Yet, we cannot give any reasonable encoding of app2 into SML, because it would require applying the argument function to the encoding of v1, with type $\langle\sigma_1\rangle_C\ \tau$, and to the encoding of v2, with type $\langle\sigma_2\rangle_C\ \tau$; that is, it would require applying an argument function at two different types. As hinted above, this is a consequence of the lack of first-class polymorphism in the SML type system; the argument function cannot be polymorphic.

These two restrictions impose one final restriction on the kind of subtyping we can encode. Consider a higher-order function h with type $\forall\alpha{<:}(\sigma_1 \to \sigma_2)(\alpha \to \sigma_2)$. What are the possible encodings of the bound $\sigma_1 \to \sigma_2$ that allow subtyping? Clearly encoding the bound as $\langle\sigma_1\rangle_C\ \tau \to \langle\sigma_2\rangle_C\ \tau$ does not allow any subtyping. Encoding the bound as $\langle\sigma_1\rangle_A\ \tau \to \langle\sigma_2\rangle_A\ \tau$ or $\langle\sigma_1\rangle_A\ \tau \to \langle\sigma_2\rangle_C\ \tau$ leads to an unsound system. (Consider applying the argument function to a value of type $\sigma_0 \geq \sigma_1$, which would type-check in the encoding, because $\langle\sigma_0\rangle_C$ unifies with $\langle\sigma_1\rangle_A$ by the definition of a respectful encoding.) However, we can soundly encode the bound as $\langle\sigma_1\rangle_C\ \tau \to \langle\sigma_2\rangle_A\ \tau$. This corresponds to a subtyping rule on functional types that asserts $\tau_1 \to \tau_2 \leq \tau_1 \to \tau_2'$ if and only if $\tau_2 \leq \tau_2'$. This is the main reason why we focus on first-order subtyping in this paper.

Despite these restrictions, the phantom-types technique is still a viable method for encoding subtyping in a language like SML. All of the examples of phantom types found in the literature satisfy these restrictions. In practice, one rarely needs first-class polymorphism or complicated dependencies between the subtypes of function arguments, particularly when implementing a safe interface to existing library functions.

## 5 A Formalization

As the previous section illustrates, there are subtle issues regarding the kind of subtyping that can be captured using phantom types. In this section, we clarify the picture by exhibiting a typed calculus with a suitable notion of subtyping that can be faithfully translated into a language such as SML, via a phantom types encoding. The idea is simple: to see if an interface can be implemented using phantom types, first express the interface in this calculus in such a way that the program type-checks. If it is possible to do so, our results show that a translation using phantom types exists. The target of the translation is a calculus embodying the essence of SML, essentially the calculus of Damas and Milner (1982), a predicative polymorphic lambda calculus.

### *5.1 The Source Calculus* $\lambda^{\text{DM}}_{<:}$

Our source calculus, $\lambda^{\text{DM}}_{<:}$, is a variant of the Damas-Milner calculus with a restricted notion of bounded polymorphism, and allowing multiple types for constants. We assume a partially ordered set $(T, \leq)$ of base types, which forms the subtyping hierarchy.

**Types of $\lambda^{\text{DM}}_{<:}$:**

| | |
|---|---|
| $\tau ::=$ | types |
| $\quad t$ | $\quad$ base type ($t \in T$) |
| $\quad \alpha$ | $\quad$ type variable |
| $\quad \tau_1 \to \tau_2$ | $\quad$ function type |
| $\sigma ::=$ | prenex quantified type scheme |
| $\quad \forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau)$ | $\quad$ ($FV(\tau_i) = \langle\,\rangle$, for all $i$) |

Given a type $\tau$ in $\lambda^{\text{DM}}_{<:}$, we define $FV(\tau)$ to be the sequence of type variables appearing in $\tau$, in depth-first, left-to-right order. (Since there is no binder in $\tau$, all the type variables appearing in $\tau$ are necessarily free.) We write sequences using the notation $\langle \alpha_1, \ldots, \alpha_n \rangle$. We make a syntactic restriction that precludes the use of type variables in the bounds of quantified type variables.

An important aspect of our calculus, at least for our purposes, is the set of constants that we allow. We distinguish between two types of constants: base constants and primitive operations. Base constants, taken from a set $C$, are constants representing values of base types $t \in T$. We suppose a function $\pi_C : C \to T$ assigning a base type to every base constant. The primitive operations, taken from a set $F$, are operations acting on constants and returning constants.[4] Rather than giving primitive operations polymorphic types, we assume that the operations can have multiple types, which encode the allowed subtyping. We suppose a function $\pi_F$ assigning to every primitive operation $f \in F$ a set of types $\pi_F(f)$, each type a functional type of the form $t \to t'$ (for $t, t' \in T$).

Our expression language is a typical polymorphic lambda calculus expression language.

**Expression Syntax of $\lambda^{\text{DM}}_{<:}$:**

| | |
|---|---|
| $e ::=$ | monomorphic expressions |
| $\quad c$ | $\quad$ base constant ($c \in C$) |
| $\quad f$ | $\quad$ primitive operation ($f \in F$) |
| $\quad \lambda x{:}\tau(e)$ | $\quad$ functional abstraction |
| $\quad e_1\ e_2$ | $\quad$ function application |
| $\quad x$ | $\quad$ variable |
| $\quad p\ [\tau_1, \ldots, \tau_n]$ | $\quad$ type application |

[4] For simplicity, we will not deal with higher-order primitive operations here—they would simply complicate the formalism without bringing any new insight. Likewise, allowing primitive operations to act on and return tuples of values is a simple extension of the formalism presented here.

| | |
|---|---|
| $\quad$ **let** $x = p$ **in** $e$ | $\quad$ local binding |
| $p ::=$ | polymorphic expressions |
| $\quad x$ | $\quad$ variable |
| $\quad \Lambda\alpha_1{<:}\tau_1,\ldots,\alpha_n{<:}\tau_n(e)$ | $\quad$ type abstraction ($FV(\tau_i) = \langle\,\rangle$, for all $i$) |

The operational semantics is given using a standard contextual reduction semantics, written $e_1 \longrightarrow_{<:} e_2$. While the details can be found in Appendix A, we note here the most important reduction rule, involving constants:

$$f\ c \longrightarrow_{<:} c' \ \text{ iff } \ \delta(f, c) = c'$$

where $\delta : F \times C \rightharpoonup C$ is a partial function defining the result of applying a primitive operation to a base constant.

As previously noted, we do not allow primitive operations to be polymorphic. However, we can easily use the fact that they can take on many types to write polymorphic wrappers. For example, we can write a polymorphic wrapper $\Lambda\alpha{<:}\tau(\lambda x{:}\alpha(f\ x))$ to capture the expected behavior of a function $f$ that may be applied to any subtype of $\tau$. We will see shortly that this function is well-typed.

The typing rules for $\lambda^{\text{DM}}_{<:}$ are the standard Damas-Milner typing rules, modified to account for subtyping. The full set of rules is given in Appendix A. Subtyping is given by a judgment $\Delta \vdash_{<:} \tau_1 <: \tau_2$ and is derived from the subtyping on the base types. The interesting rules are:

$$\frac{t_1 \leq t_2}{\Delta \vdash_{<:} t_1 <: t_2} \qquad \frac{\Delta \vdash_{<:} \tau_2 <: \tau_2'}{\Delta \vdash_{<:} \tau_1 \rightarrow \tau_2 <: \tau_1 \rightarrow \tau_2'}.$$

Notice that subtyping at higher types only involves the result type, following our discussion in Sections 2 and 4. The typing rules are given by judgments $\Delta;\Gamma \vdash_{<:} e : \tau$ for types and $\Delta;\Gamma \vdash_{<:} p : \sigma$ for type schemes. The rule for primitive operations is interesting:

$$\frac{\begin{array}{c}\text{For all } i \text{ and for all } \tau_i' \text{ such that } \vdash_{<:} \tau_i' <: \tau_i,\\ (\tau' \rightarrow \tau)\{\tau_1'/\alpha_1,\ldots,\tau_n'/\alpha_n\} \in \pi_F(f)\end{array}}{\Delta, \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n; \Gamma \vdash_{<:} f : \tau' \rightarrow \tau} \left(\begin{array}{c} f \in F,\\ FV(\tau') = \langle\alpha_1,\ldots,\alpha_n\rangle\end{array}\right).$$

The syntactic restriction on type variable bounds ensures that each $\tau_i$ has no type variables, so each $\tau_i' <: \tau_i$ is well-defined. The rule captures the notion that any subtyping on a primitive operation through the use of bounded polymorphism is in fact realized by the "many types" interpretation of the operation.

Subtyping occurs at type application:

$$\frac{\Delta;\Gamma \vdash_{<:} p : \forall\alpha_1{<:}\tau_1,\ldots,\alpha_n{<:}\tau_n(\tau) \quad \Delta \vdash_{<:} \tau_1' <: \tau_1 \quad \ldots \quad \Delta \vdash_{<:} \tau_n' <: \tau_n}{\Delta;\Gamma \vdash_{<:} p\ [\tau_1',\ldots,\tau_n'] : \tau\{\tau_1'/\alpha_1,\ldots,\tau_n'/\alpha_n\}}.$$

As discussed in the previous section, there is no subsumption in the system: subtyping must be witnessed by type application. Hence, there is a difference between the type $t_1 \rightarrow t_2$ (where $t_1, t_2 \in T$) and $\forall\alpha{<:}t_1(\alpha \rightarrow t_2)$; namely, the former does not allow any subtyping. The restrictions of Section 4 are formalized by prenex quantification and the syntactic restriction on type variable bounds.

Clearly, type soundness of the above system depends on the definition of $\delta$ over the constants. We say that $\pi_F$ is sound with respect to $\delta$ if for all $f \in F$ and $c \in C$, $\vdash_{<:} f\ c : \tau$ implies that $\delta(f, c)$ is defined and $\pi_C(\delta(f, c)) = \tau$. This definition ensures that any application of a primitive operation $f$ to a base constant $c$ yields exactly one value $\delta(f, c)$ at exactly one type $\pi_C(\delta(f, c)) = \tau$. This leads to the following conditional type soundness result for $\lambda^{\mathrm{DM}}_{<:}$:

*Theorem 5.1*
If $\pi_F$ is sound with respect to $\delta$, $\vdash_{<:} e : \tau$, and $e \longrightarrow_{<:} e'$, then $\vdash_{<:} e' : \tau$ and either $e'$ is a value or there exists $e''$ such that $e' \longrightarrow_{<:} e''$.

*Proof*
See Appendix A. □

### *5.2 The Target Calculus* $\lambda^{\mathrm{DM}}_{\mathsf{T}}$

Our target calculus, $\lambda^{\mathrm{DM}}_{\mathsf{T}}$, is meant to capture the appropriate aspects of SML that are relevant for the phantom types encoding of subtyping. Essentially, it is the Damas-Milner calculus (Damas & Milner, 1982) extended with a single type constructor $\mathsf{T}$.

**Types of $\lambda^{\mathrm{DM}}_{\mathsf{T}}$:**

| | |
|---|---|
| $\tau ::=$ | types |
| $\quad\alpha$ | type variable |
| $\quad\tau_1 \rightarrow \tau_2$ | function type |
| $\quad\mathsf{T}\ \tau$ | type constructor $\mathsf{T}$ |
| $\quad 1$ | unit type |
| $\quad\tau_1 \times \tau_2$ | product type |
| $\sigma ::=$ | prenex quantified type scheme |
| $\quad\forall\alpha_1, \ldots, \alpha_n(\tau)$ | |

**Expression Syntax of $\lambda^{\mathrm{DM}}_{\mathsf{T}}$:**

| | |
|---|---|
| $e ::=$ | monomorphic expressions |
| $\quad c$ | base constant ($c \in C$) |
| $\quad f$ | primitive operation ($f \in F$) |
| $\quad\lambda x{:}\tau(e)$ | functional abstraction |
| $\quad e_1\ e_2$ | function application |
| $\quad x$ | variable |
| $\quad p\ [\tau_1, \ldots, \tau_n]$ | type application |
| $\quad\mathbf{let}\ x = p\ \mathbf{in}\ e$ | local binding |
| $p ::=$ | polymorphic expressions |
| $\quad x$ | variable |

| | |
|---|---|
| $\Lambda\alpha_1 \ldots, \alpha_n(e)$ | type abstraction |

The operational semantics (via a reduction relation $\longrightarrow_{\mathsf{T}}$) and most typing rules (via a judgment $\Delta; \Gamma \vdash_{\mathsf{T}} e : \tau$) are standard. The calculus is fully described in Appendix B. As before, we assume that we have sets of constants $C$ and $F$ and a function $\delta$ providing semantics for primitive applications. Likewise, we assume that $\pi_C$ and $\pi_F$ provide types for constants, with similar restrictions: $\pi_C(c)$ yields a closed type of the form $\mathsf{T}\ \tau$, while $\pi_F(f)$ yields a set of closed types of the form $(\mathsf{T}\ \tau_1) \to (\mathsf{T}\ \tau_2)$. The typing rule for primitive operations in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$ is similar to the corresponding rule in $\lambda_{<:}^{\mathrm{DM}}$. It ensures that a primitive operation can be given a type (possibly with free type variables) if all the substitution instances of that type are allowed by the assignment $\pi_F$. Given two types $\tau$ and $\tau'$ in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$, where $\tau'$ is a closed type, we define their unification $\mathit{unify}(\tau, \tau')$ to be a sequence of bindings $\langle(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n)\rangle$ in depth-first, left-to-right order of appearance of $\alpha_1, \ldots, \alpha_n$ in $\tau$ such that $\tau\{\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\} = \tau'$, or $\varnothing$ if $\tau'$ is not a substitution instance of $\tau$. As for $\lambda_{<:}^{\mathrm{DM}}$, given a type $\tau$ in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$, we define $FV(\tau)$ to be the sequence of free type variables appearing in $\tau$, in depth-first, left-to-right order.

$$\frac{\begin{array}{c}\text{For all } \tau' \in \pi_C(C) \text{ such that} \\ \mathit{unify}(\tau_1, \tau') = \langle(\alpha_1, \tau_1'), \ldots, (\alpha_n, \tau_n'), \ldots\rangle, \\ (\tau_1 \to \tau_2)\{\tau_1'/\alpha_1, \ldots, \tau_n'/\alpha_n\} \in \pi_F(f)\end{array}}{\Delta, \alpha_1, \ldots, \alpha_n; \Gamma \vdash_{\mathsf{T}} f : \tau_1 \to \tau_2} \left(\begin{array}{c} f \in F, \\ FV(\tau_1) = \langle\alpha_1, \ldots, \alpha_n\rangle \end{array}\right).$$

Again, this rule captures our notion of "subtyping through unification" by ensuring that the operation is defined at every base type that unifies with its argument type. Our notion of soundness of $\pi_F$ with respect to $\delta$ carries over and we can again establish a conditional type soundness result:

*Theorem 5.2*
If $\pi_F$ is sound with respect to $\delta$, $\vdash_{\mathsf{T}} e : \tau$, and $e \longrightarrow_{\mathsf{T}} e'$, then $\vdash_{\mathsf{T}} e' : \tau$ and either $e'$ is a value or there exists $e''$ such that $e' \longrightarrow_{\mathsf{T}} e''$.

*Proof*
See Appendix B. □

Note that the types $\mathsf{T}\ \tau$, 1, and $\tau_1 \times \tau_2$ have no corresponding introduction and elimination expressions. We include these types for the exclusive purpose of constructing the phantom types used by the encodings. We could add other types to allow more encodings, but these suffice for the encodings of Section 3.

### *5.3 The Translation*

Thus far, we have a calculus $\lambda_{<:}^{\mathrm{DM}}$ embodying the notion of subtyping that interests us and a calculus $\lambda_{\mathsf{T}}^{\mathrm{DM}}$ capturing the essence of the SML type system. We now establish a translation from the first calculus into the second using phantom types to encode the subtyping, showing that we can indeed capture that particular notion of subtyping in SML. Moreover, we show that the translation preserves the soundness of the types assigned to constants, thereby guaranteeing that if the original system was sound, the system obtained by translation is sound as well.

We first describe how to translate types in $\lambda^{\text{DM}}_{<:}$. Since subtyping is only witnessed at type abstraction, the type translation realizes the subtyping using the phantom types encoding of abstract and concrete subtypes. The translation is parameterized by an environment $\rho$ associating every (free) type variable with a type in $\lambda^{\text{DM}}_{\mathsf{T}}$ representing the abstract encoding of the bound.

**Types Translation:**

$$\mathcal{T}[\![\alpha]\!]\rho \triangleq \rho(\alpha)$$
$$\mathcal{T}[\![t]\!]\rho \triangleq \mathsf{T}\ \langle t\rangle_C$$
$$\mathcal{T}[\![\tau_1 \to \tau_2]\!]\rho \triangleq \mathcal{T}[\![\tau_1]\!]\rho \to \mathcal{T}[\![\tau_2]\!]\rho$$
$$\mathcal{T}[\![\forall\alpha_1{<:}\tau_1,\ldots,\alpha_n{<:}\tau_n(\tau)]\!]\rho \triangleq \forall\alpha_{11},\ldots,\alpha_{1k_1},\ldots,\alpha_{n1},\ldots,\alpha_{nk_n}(\mathcal{T}[\![\tau]\!]\rho[\alpha_i \mapsto \tau_i^A])$$
$$\text{where } \tau_i^A = \mathcal{A}[\![\tau_i]\!]$$
$$\text{and } FV(\tau_i^A) = \langle \alpha_{i1},\ldots,\alpha_{ik_i}\rangle$$

If $\rho$ is empty, we simply write $\mathcal{T}[\![\tau]\!]$ for $\mathcal{T}[\![\tau]\!]\rho$. To compute the abstract and concrete encodings of a type, we define the functions $\mathcal{A}$ and $\mathcal{C}$.

**Abstract and Concrete Encodings:**

$$\mathcal{A}[\![t]\!] \triangleq \mathsf{T}\ \langle t\rangle_A$$
$$\mathcal{A}[\![\tau_1 \to \tau_2]\!] \triangleq \mathcal{C}[\![\tau_1]\!] \to \mathcal{A}[\![\tau_2]\!]$$
$$\mathcal{C}[\![t]\!] \triangleq \mathsf{T}\ \langle t\rangle_C$$
$$\mathcal{C}[\![\tau_1 \to \tau_2]\!] \triangleq \mathcal{C}[\![\tau_1]\!] \to \mathcal{C}[\![\tau_2]\!]$$

The syntactic restriction on type variable bounds ensures that $\mathcal{A}$ and $\mathcal{C}$ are always well defined, as they are never applied to type variables. Furthermore, the above translation depends on the fact that the type encodings $\langle t\rangle_C$ and $\langle t\rangle_A$ are expressible in the $\lambda^{\text{DM}}_{\mathsf{T}}$ type system using $\mathsf{T}$, 1, and $\times$.

We extend the type transformation $\mathcal{T}$ to type contexts $\Gamma$ in the obvious way:

**Type Contexts Translation:**

$$\mathcal{T}[\![x_1 : \tau_1,\ldots,x_n : \tau_n]\!]\rho \triangleq x_1 : \mathcal{T}[\![\tau_1]\!]\rho,\ldots,x_n : \mathcal{T}[\![\tau_n]\!]\rho$$

Finally, if we take the base constants and the primitive operations in $\lambda^{\text{DM}}_{<:}$ and assume that $\pi_F$ is sound with respect to $\delta$, then the translation can be used to assign types to the constants and operations such that they are sound in the target calculus. We first extend the definition of $\mathcal{T}$ to $\pi_C$ and $\pi_F$ in the obvious way:

**Interpretations Translation:**

$\mathcal{T}[\![\pi_C]\!] \triangleq \pi'_C \quad$ where $\pi'_C(c) = \mathcal{T}[\![\pi_C(c)]\!]$
$\mathcal{T}[\![\pi_F]\!] \triangleq \pi'_F \quad$ where $\pi'_F(f) = \{\mathcal{T}[\![\tau]\!] \mid \tau \in \pi_F(f)\}$

We can further show that the translated types do not allow us to "misuse" the constants in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$:

*Theorem 5.3*
If $\pi_F$ is sound with respect to $\delta$ in $\lambda_{<:}^{\mathrm{DM}}$, then $\mathcal{T}[\![\pi_F]\!]$ is sound with respect to $\delta$ in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$.

*Proof*
See Appendix C. □

We therefore take $\mathcal{T}[\![\pi_C]\!]$ and $\mathcal{T}[\![\pi_F]\!]$ to be the interpretations in the target calculus $\lambda_{\mathsf{T}}^{\mathrm{DM}}$.

We can now define the translation of expressions via a translation of typing derivations, $\mathcal{E}$, taking care to respect the types given by the above type translation. We note that the translation below works only if the concrete encodings being used do not contain free type variables. Again, the translation is parameterized by an environment $\rho$, as in the type translation.

**Expressions Translation:**

$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} x : \tau]\!]\rho \triangleq x$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} c : \tau]\!]\rho \triangleq c$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} f : \tau]\!]\rho \triangleq f$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} \lambda x{:}\tau'(e) : \tau]\!]\rho \triangleq \lambda x{:}\mathcal{T}[\![\tau']\!]\rho(\mathcal{E}[\![e]\!]\rho)$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} e_1\ e_2 : \tau]\!]\rho \triangleq (\mathcal{E}[\![e_1]\!]\rho)\ \mathcal{E}[\![e_2]\!]\rho$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} \mathbf{let}\ x = p\ \mathbf{in}\ e : \tau]\!]\rho \triangleq \mathbf{let}\ x = \mathcal{E}[\![p]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![e]\!]\rho$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} p\ [\tau_1, \ldots, \tau_n] : \tau]\!]\rho \triangleq$
  $(\mathcal{E}[\![p]\!]\rho)\ [\tau_{11}, \ldots, \tau_{1k_1}, \ldots, \tau_{n1}, \ldots, \tau_{nk_n}]$
    where $\mathcal{B}[\![p]\!]\Gamma = \langle(\alpha_1, \tau_1^B), \ldots, (\alpha_n, \tau_n^B)\rangle$ and $\tau_i^A = \mathcal{A}[\![\tau_i^B]\!]$
    and $FV(\tau_i^B) = \langle \alpha_{i1}, \ldots, \alpha_{ik_i}\rangle$ and $\tau_i^T = \mathcal{T}[\![\tau_i]\!]\rho$
    and $\mathit{unify}(\tau_i^A, \tau_i^T) = \langle(\alpha_{i1}, \tau_{i1}), \ldots, (\alpha_{ik_i}, \tau_{ik_i}), \ldots\rangle$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} x : \sigma]\!]\rho \triangleq x$
$\mathcal{E}[\![\Delta; \Gamma \vdash_{<:} \Lambda\alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(e) : \sigma]\!]\rho \triangleq$
  $\Lambda\alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}(\mathcal{E}[\![e]\!]\rho[\alpha_i \mapsto \tau_i^A])$
    where $\tau_i^A = \mathcal{A}[\![\tau_i]\!]$ and $FV(\tau_i^A) = \langle\alpha_{i1}, \ldots, \alpha_{ik_i}\rangle$

Again, if $\rho$ is empty, we simply write $\mathcal{E}[\![e]\!]$ for $\mathcal{E}[\![e]\!]\rho$. The function $\mathcal{B}$ returns the bounds of a type abstraction, using the environment $\Gamma$ to resolve variables.

**Bounds of a Type Abstraction:**

$\mathcal{B}[\![x]\!]\Gamma \triangleq \langle(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n)\rangle \quad$ where $\Gamma(x) = \forall\alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau)$
$\mathcal{B}[\![\Lambda\alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(e)]\!]\Gamma \triangleq \langle(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n)\rangle$

We use $\mathcal{B}$ and *unify* to perform unification "by hand." In most programming languages, type inference performs this automatically.

We can verify that this translation is type-preserving:

*Theorem 5.4*
If $\vdash_{<:} e : \tau$, then $\vdash_{\mathsf{T}} \mathcal{E}[\![\vdash_{<:} e : \tau]\!] : \mathcal{T}[\![\tau]\!]$.

*Proof*
See Appendix C. $\square$

Theorem 5.4 shows that the translation captures the right notion of subtyping, in the sense that interests us, particularly when designing an interface. Given a set of constants making up the interface, suppose we can assign types to those constants in $\lambda^{\mathrm{DM}}_{<:}$ in a way that gives the desired subtyping; that is, we can write type-correct expressions of the form $\Lambda\alpha{<:}t(\lambda x{:}\alpha(f\ x))$ with type $\forall\alpha{<:}t(\alpha \to \tau)$. In other words, the typing $\pi_F$ is sound with respect to the semantics of $\delta$. By Theorem 5.1, this means that $\lambda^{\mathrm{DM}}_{<:}$ with these constants is sound and we can safely use these constants in $\lambda^{\mathrm{DM}}_{<:}$. In particular, we can write the program:

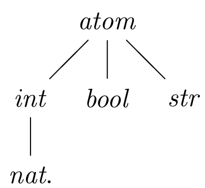
$$
\begin{array}{l}
\textbf{let } g_1 = \Lambda\alpha{<:}t_{i_1}(\lambda x{:}\alpha(f_1\ x)) \textbf{ in} \\
\quad \vdots \\
\textbf{let } g_n = \Lambda\alpha{<:}t_{i_n}(\lambda x{:}\alpha(f_n\ x)) \textbf{ in} \\
\quad e.
\end{array}
$$

By Theorem 5.4, the translation of the above program executes without run-time errors. Furthermore, by Theorem 5.3, the phantom-types encoding of the types of these constants are sound with respect to $\delta$ in $\lambda^{\mathrm{DM}}_{\mathsf{T}}$. Hence, by Theorem 5.2, $\lambda^{\mathrm{DM}}_{\mathsf{T}}$ with these constants is sound and we can safely use these constants in $\lambda^{\mathrm{DM}}_{\mathsf{T}}$. Therefore, we can replace the body of the translated program with an arbitrary $\lambda^{\mathrm{DM}}_{\mathsf{T}}$ expression that type-checks in that context and the resulting program will still execute without run-time errors. Essentially, the translation of the let bindings corresponds to a "safe" interface to the primitives; programs that use this interface in a type-safe manner are guaranteed to execute without run-time errors.

### *5.4 Example and Remarks*

In this section, we work through a mostly complete example before turning our attention to some general remarks.

Recall the subtyping hierarchy introduced in Section 2 and here extended to include natural numbers and strings.



We can encode this hierarchy with phantom types as follows:

$$\begin{array}{rclcrcl} \langle atom\rangle_A &=& \alpha \times (\beta \times \gamma) & & \langle atom\rangle_C &=& 1 \times (1 \times 1) \\ \langle int\rangle_A &=& \mathsf{T}\ \alpha \times (\beta \times \gamma) & & \langle int\rangle_C &=& \mathsf{T}\ 1 \times (1 \times 1) \\ \langle nat\rangle_A &=& \mathsf{T}\ (\mathsf{T}\ \alpha) \times (\beta \times \gamma) & & \langle nat\rangle_C &=& \mathsf{T}\ (\mathsf{T}\ 1) \times (1 \times 1) \\ \langle bool\rangle_A &=& \alpha \times (\mathsf{T}\ \beta \times \gamma) & & \langle bool\rangle_C &=& 1 \times (\mathsf{T}\ 1 \times 1) \\ \langle str\rangle_A &=& \alpha \times (\beta \times \mathsf{T}\ \gamma) & & \langle str\rangle_C &=& 1 \times (1 \times \mathsf{T}\ 1). \end{array}$$

We consider two primitive operations `double` and `toString` with

$$\begin{array}{rcl} \pi_F(\texttt{double}) &=& \{int \to int,\ nat \to nat\} \\ \pi_F(\texttt{toString}) &=& \{atom \to str,\ int \to str,\ nat \to str,\ bool \to str,\ str \to str\}. \end{array}$$

We can derive the following typing judgments in $\lambda^{\text{DM}}_{<:}$, which capture the intended subtyping:

$$\begin{array}{l} \vdash_{<:} \Lambda\alpha{<:}int(\lambda x{:}\alpha(\texttt{double}\ x)) : \forall\alpha{<:}int(\alpha \to \alpha) \\ \vdash_{<:} \Lambda\alpha{<:}atom(\lambda x{:}\alpha(\texttt{toString}\ x)) : \forall\alpha{<:}atom(\alpha \to \texttt{str}). \end{array}$$

Applying our translation to these functions yields:

$$\begin{array}{rcl} \mathcal{E}[\![\Lambda\alpha{<:}int(\lambda x{:}\alpha(\texttt{double}\ x))]\!] &=& \Lambda\alpha,\beta,\gamma(\lambda x{:}\mathsf{T}\ (\mathsf{T}\ \alpha \times (\beta \times \gamma))(\texttt{double}\ x)) \\ \mathcal{E}[\![\Lambda\alpha{<:}atom(\lambda x{:}\alpha(\texttt{toString}\ x))]\!] &=& \Lambda\alpha,\beta,\gamma(\lambda x{:}\mathsf{T}\ (\alpha \times (\beta \times \gamma))(\texttt{toString}\ x)). \end{array}$$

As expected from Theorem 5.4, we can derive typing judgments that assign the translated types to these functions:

$$\begin{array}{l} \vdash_{\mathsf{T}} \Lambda\alpha,\beta,\gamma(\lambda x{:}\mathsf{T}\ (\mathsf{T}\ \alpha \times (\beta \times \gamma))(\texttt{double}\ x)) : \\ \qquad \forall\alpha,\beta,\gamma(\mathsf{T}\ (\mathsf{T}\ \alpha \times (\beta \times \gamma)) \to \mathsf{T}\ (\mathsf{T}\ \alpha \times (\beta \times \gamma))) \\ \vdash_{\mathsf{T}} \Lambda\alpha,\beta,\gamma(\lambda x{:}\mathsf{T}\ (\alpha \times (\beta \times \gamma))(\texttt{toString}\ x)) : \\ \qquad \forall\alpha,\beta,\gamma(\mathsf{T}\ (\alpha \times (\beta \times \gamma)) \to \mathsf{T}\ (1 \times (1 \times \mathsf{T}\ 1))). \end{array}$$

Interestingly, we can also derive the following typing judgments:

$$\begin{array}{l} \vdash_{\mathsf{T}} \Lambda\alpha(\lambda x{:}\mathsf{T}\ (\mathsf{T}\ \alpha \times (\alpha \times \alpha))(\texttt{double}\ x)) \\ \qquad \forall\alpha(\mathsf{T}\ (\mathsf{T}\ \alpha \times (\alpha \times \alpha)) \to \mathsf{T}\ (\mathsf{T}\ \alpha \times (\alpha \times \alpha))) \\ \vdash_{\mathsf{T}} \Lambda\alpha,\beta(\lambda x{:}\mathsf{T}\ (\alpha \times (\beta \times \beta))(\texttt{toString}\ x)) \\ \qquad \forall\alpha,\beta(\mathsf{T}\ (\alpha \times (\beta \times \beta)) \to \mathsf{T}\ (1 \times (1 \times \mathsf{T}\ 1))). \end{array}$$

The first function type-checks because, of all base types, only $\mathsf{T}\ \langle int\rangle_C$ unifies with $\mathsf{T}\ (\mathsf{T}\ \alpha \times (\alpha \times \alpha))$, by the substitution $(\alpha, 1)$, and $\{\mathsf{T}\ \langle int\rangle_C \to \mathsf{T}\ \langle int\rangle_C\} \subseteq \mathcal{T}[\![\pi_F]\!](\texttt{double})$. Likewise, the second function type-checks because, of all base types, only $\mathsf{T}\ \langle atom\rangle_C$, $\mathsf{T}\ \langle int\rangle_C$, $\mathsf{T}\ \langle nat\rangle_C$ unify with $\mathsf{T}\ (\mathsf{T}\ \alpha \times (\beta \times \beta))$ and $\{\mathsf{T}\ \langle atom\rangle_C \to \mathsf{T}\ \langle str\rangle_C,\ \mathsf{T}\ \langle int\rangle_C \to \mathsf{T}\ \langle str\rangle_C,\ \mathsf{T}\ \langle nat\rangle_C \to \mathsf{T}\ \langle str\rangle_C\} \subseteq \mathcal{T}[\![\pi_F]\!](\texttt{toString})$. We can interpret the first as a function that can only be applied to integers (but not naturals) and the second as a function that can only be applied to atoms, integers, and naturals (but not booleans or strings). Observe that while these functions do not capture all of the subtyping available in their wrapped primitive operations, neither do they violate the subtyping available. This corresponds to the fact that the second set of types are instances of the first set of types under appropriate substitutions for $\beta$ and $\gamma$.

The existence of these typing judgments sheds some light on the practical aspects of using the phantom-types technique in real programming languages. Recall that the typing judgment for primitive operations is somewhat non-standard. Specifically, in contrast to most typing judgments for primitives (like the typing judgment for base constants), this judgment is not syntax directed; that is, the type is not uniquely determined by the primitive operation. This complicates a type-inference system for $\lambda_{\tau}^{\mathrm{DM}}$. At the same time, we cannot expect to integrate this typing judgment into an existing language with a Hindley-Milner style type system. Rather, we expect to integrate a primitive operation into a programming language through a foreign-function interface, at which point we give the introduced function a very base type that does not reflect the subtyping inherent in its semantics.[5] After introducing the primitive operation in this fashion, we wrap it with a function to which we can assign the intended type using the phantom types encoding, because the type system will not, in general, infer the appropriate type. It is for this reason that we have stressed the application of phantom-types technique to developing and implementing interfaces.

## 6 Conclusion

The phantom-types technique uses the definition of type equivalence in a programming language such as SML or Haskell to encode information in a free type variable of a type. Unification can then be used to enforce a particular structure on the information carried by two such types. In this paper, we have focused on encoding subtyping information. We were able to provide encodings for hierarchies with various characteristics, and more generally, hinted at a theory for how such encodings can be derived. Because the technique relies on encoding the subtyping hierarchy, the problem of extensibility arises: how resilient are the encodings to additions to the subtyping hierarchy? This is especially important when designing library interfaces. We showed in this paper that our encodings can handle extensions to the subtyping hierarchy as long as the extensions are always made with respect to a single parent in the hierarchy. We also showed how to extend the techniques we developed to encode a form of prenex bounded polymorphism, with subsumption occurring only at type application. The correctness of this encoding is established by showing how a calculus with that form of subtyping can be translated faithfully (using the encoding) into a calculus embodying the type system of SML.

It goes without saying that this approach to encoding subtyping is not without its problems from a practical point of view. As the encodings in this paper show, the types involved can become quite large. Type abbreviations can help simplify the

[5] In general, foreign-function interfaces have strict requirements on the types of foreign functions that can be called. Due to internal implementation details, language implementations rarely allow foreign functions to be given polymorphic types or types with user defined datatypes, both of which are used by the phantom types encodings.

presentation of concrete types, but abstract encodings require type variables and those variables need to appear in the interface. Having such complex types lead to interfaces themselves becoming complex, and, more seriously, the type errors reported to the user are fairly unreadable. Although the process of encoding the subtyping hierarchies can be automated—for instance, by deriving the encodings from a declarative description of the hierarchy—we see no good solution for the complexity problem. The compromise between providing safety and complicating the interface must be decided on a per-case basis.

We also note that the source language of Section 5 provides only a lower bound on the power of phantom types. For example, one can use features of the *specific* encoding used to further constrain or refine the type of operations. This is used, for instance, by Reppy (1996) to type socket operations. There is yet no general methodology for exploiting properties of encodings beyond them respecting the subtyping hierarchy.

### *Acknowledgments*

A preliminary version of this paper appeared in the *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science*, pp. 448–460, 2002. This work was mostly done when the second author was at Cornell University. We have benefited from discussions with Greg Morrisett and Dave MacQueen. John Reppy pointed out the work of Burton. Stephanie Weirich, Vicky Weissman, and Steve Zdancewic provided helpful comments on an early draft of this paper. Thanks also to the anonymous referees for many suggestions to improve the presentation. The second author was partially supported by ONR grant N00014-00-1-03-41.

## A The Calculus $\lambda^{\mathrm{DM}}_{<:}$

**Values:**

| | |
|---|---|
| $v ::=$ | values |
| $c$ | base constant ($c \in C$) |
| $f$ | primitive operation ($f \in F$) |
| $\lambda x{:}\tau(e)$ | functional abstraction |

**Evaluation Contexts:**

| | |
|---|---|
| $E ::=$ | evaluation contexts |
| $[\,]$ | empty context |
| $E\ e$ | application context |
| $v\ E$ | argument context |
| $E\ [\tau_1, \ldots, \tau_n]$ | type application context |

**Operational Semantics:**

$$(\lambda x{:}\tau(e))\ v \longrightarrow_{<:} e\{v/x\}$$

$(\Lambda\alpha_1{<:}\tau_1,\ldots,\alpha_n{<:}\tau_n(e))\ [\tau'_1,\ldots,\tau'_n] \longrightarrow_{<:} e\{\tau'_1/\alpha_1,\ldots,\tau'_n/\alpha_n\}$
$\textbf{let}\ x = p\ \textbf{in}\ e \longrightarrow_{<:} e\{p/x\}$
$f\ c \longrightarrow_{<:} c'$ iff $\delta(f,c) = c'$
$E[e_1] \longrightarrow_{<:} E[e_2]$ iff $e_1 \longrightarrow_{<:} e_2$

The function $\delta : F \times C \rightharpoonup C$ is a partial function defining the result of applying a primitive operation to a base constant.

**Typing Contexts:**

| | |
|---|---|
| $\Gamma$ ::= | type environments |
| $\cdot$ | empty |
| $\Gamma, x : \tau$ | type |
| $\Gamma, x : \sigma$ | type scheme |
| $\Delta$ ::= | subtype environments |
| $\cdot$ | empty |
| $\Delta, \alpha <: \tau$ | subtype |

**Judgments:**

| | |
|---|---|
| $\vdash_{<:} \Delta$ ctxt | good context $\Delta$ |
| $\Delta \vdash_{<:} \tau$ type | good type $\tau$ |
| $\Delta \vdash_{<:} \sigma$ scheme | good type scheme $\sigma$ |
| $\Delta \vdash_{<:} \Gamma$ ctxt | good context $\Gamma$ |
| $\Delta \vdash_{<:} \tau_1 <: \tau_2$ | type $\tau_1$ subtype of $\tau_2$ |
| $\Delta; \Gamma \vdash_{<:} e : \tau$ | good expression $e$ with type $\tau$ |
| $\Delta; \Gamma \vdash_{<:} p : \sigma$ | good expression $p$ with type scheme $\sigma$ |

**Judgment** $\vdash_{<:} \Delta$ **ctxt:**

$$\frac{}{\vdash_{<:} \cdot \text{ ctxt}} \qquad \frac{\vdash_{<:} \Delta \text{ ctxt} \quad \Delta \vdash_{<:} \tau \text{ type}}{\vdash_{<:} \Delta, \alpha <: \tau \text{ ctxt}}$$

**Judgment** $\Delta \vdash_{<:} \tau$ **type:**

$$\frac{}{\Delta \vdash_{<:} t \text{ type}} \qquad \frac{\vdash_{<:} \Delta \text{ ctxt} \quad \alpha \in \text{dom}(\Delta)}{\Delta \vdash_{<:} \alpha \text{ type}} \qquad \frac{\Delta \vdash_{<:} \tau_1 \text{ type} \quad \Delta \vdash_{<:} \tau_2 \text{ type}}{\Delta \vdash_{<:} \tau_1 \rightarrow \tau_2 \text{ type}}$$

**Judgment** $\Delta \vdash_{<:} \sigma$ **scheme:**

$$\frac{\Delta, \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n \vdash_{<:} \tau \text{ type}}{\Delta \vdash_{<:} \forall\alpha_1{<:}\tau_1,\ldots,\alpha_n{<:}\tau_n(\tau) \text{ scheme}}$$

**Judgment** $\Delta \vdash_{<:} \Gamma$ ctxt**:**

$$\frac{}{\Delta \vdash_{<:} \cdot \text{ ctxt}} \qquad \frac{\Delta \vdash_{<:} \Gamma \text{ ctxt} \quad \Delta \vdash_{<:} \tau \text{ type}}{\Delta \vdash_{<:} \Gamma, x : \tau \text{ ctxt}} \qquad \frac{\Delta \vdash_{<:} \Gamma \text{ ctxt} \quad \Delta \vdash_{<:} \sigma \text{ scheme}}{\Delta \vdash_{<:} \Gamma, x : \sigma \text{ ctxt}}$$

**Judgment** $\Delta \vdash_{<:} \tau_1 <: \tau_2$**:**

$$\frac{}{\Delta \vdash_{<:} \tau <: \tau} \qquad \frac{}{\Delta, \alpha <: \tau \vdash_{<:} \alpha <: \tau} \qquad \frac{t_1 \leq t_2}{\Delta \vdash_{<:} t_1 <: t_2}$$

$$\frac{\Delta \vdash_{<:} \tau_2 <: \tau_3}{\Delta \vdash_{<:} \tau_1 \rightarrow \tau_2 <: \tau_1 \rightarrow \tau_3} \qquad \frac{\Delta \vdash_{<:} \tau_1 <: \tau_2 \quad \Delta \vdash_{<:} \tau_2 <: \tau_3}{\Delta \vdash_{<:} \tau_1 <: \tau_3}$$

**Judgment** $\Delta; \Gamma \vdash_{<:} e : \tau$**:**

$$\frac{}{\Delta; \Gamma \vdash_{<:} c : \pi_C(c)} \; (c \in C)$$

$$\frac{\text{For all } i \text{, and for all } \tau'_i \text{ such that } \vdash_{<:} \tau'_i <: \tau_i, \quad (\tau' \rightarrow \tau)\{\tau'_1/\alpha_1, \ldots, \tau'_n/\alpha_n\} \in \pi_F(f)}{\Delta, \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n; \Gamma \vdash_{<:} f : \tau' \rightarrow \tau} \; \left( \begin{array}{c} f \in F, \\ FV(\tau') = \langle \alpha_1, \ldots, \alpha_n \rangle \end{array} \right)$$

$$\frac{\Delta \vdash_{<:} \Gamma \text{ ctxt}}{\Delta; \Gamma, x : \tau \vdash_{<:} x : \tau} \qquad \frac{\Delta; \Gamma, x : \tau \vdash_{<:} e : \tau'}{\Delta; \Gamma \vdash_{<:} \lambda x{:}\tau(e) : \tau \rightarrow \tau'}$$

$$\frac{\Delta; \Gamma \vdash_{<:} e_1 : \tau_1 \rightarrow \tau_2 \quad \Delta; \Gamma \vdash_{<:} e_2 : \tau_1}{\Delta; \Gamma \vdash_{<:} e_1 \; e_2 : \tau_2}$$

$$\frac{\Delta; \Gamma \vdash_{<:} p : \forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau) \quad \Delta \vdash_{<:} \tau'_1 <: \tau_1 \quad \ldots \quad \Delta \vdash_{<:} \tau'_n <: \tau_n}{\Delta; \Gamma \vdash_{<:} p \; [\tau'_1, \ldots, \tau'_n] : \tau\{\tau'_1/\alpha_1, \ldots, \tau'_n/\alpha_n\}}$$

$$\frac{\Delta; \Gamma, x : \sigma \vdash_{<:} e : \tau \quad \Delta; \Gamma \vdash_{<:} p : \sigma}{\Delta; \Gamma \vdash_{<:} \textbf{let } x = p \textbf{ in } e : \tau}$$

**Judgment** $\Delta; \Gamma \vdash_{<:} p : \sigma$**:**

$$\frac{\Delta \vdash_{<:} \Gamma \text{ ctxt} \quad \Delta, \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n; \Gamma \vdash_{<:} e : \tau}{\Delta; \Gamma \vdash_{<:} \Lambda \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(e) : \forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau)} \; (\alpha_1, \ldots, \alpha_n \notin \Delta)$$

### *A.1 Proofs*

The proof of soundness for $\lambda^{\mathrm{DM}}_{<:}$ is mostly standard, relying on preservation and progress lemmas. For completeness, we present all the lemmas needed to derive the proof, but leave most of the straightforward details to the reader.

*Lemma A.1*

(a) Monomorphic expression substitution preserves typing:

— If $\Delta; \Gamma, x : \tau' \vdash_{<:} e : \tau$ and $\Delta; \Gamma \vdash_{<:} e' : \tau'$, then $\Delta; \Gamma \vdash_{<:} e\{e'/x\} : \tau$.
— If $\Delta; \Gamma, x : \tau' \vdash_{<:} p : \sigma$ and $\Delta; \Gamma \vdash_{<:} e' : \tau'$, then $\Delta; \Gamma \vdash_{<:} p\{e'/x\} : \tau$.

(b) Polymorphic expression substitution preserves typing:

— If $\Delta; \Gamma, x : \sigma' \vdash_{<:} e : \tau$ and $\Delta; \Gamma \vdash_{<:} p' : \sigma'$, then $\Delta; \Gamma \vdash_{<:} e\{p'/x\} : \tau$.
— If $\Delta; \Gamma, x : \sigma' \vdash_{<:} p : \sigma$ and $\Delta; \Gamma \vdash_{<:} p' : \sigma'$, then $\Delta; \Gamma \vdash_{<:} p\{p'/x\} : \sigma$.

(c) Type subsumption preserves subtyping:

— If $\Delta, \alpha <: \tau' \vdash_{<:} \tau_1 <: \tau_2$, and $\tau'' <: \tau'$ then $\Delta, \alpha <: \tau'' \vdash_{<:} \tau_1 <: \tau_2$.

(d) Type subsumption preserves typing:

— If $\Delta, \alpha <: \tau'; \Gamma \vdash_{<:} e : \tau$ and $\tau'' <: \tau'$, then $\Delta, \alpha <: \tau''; \Gamma \vdash_{<:} e : \tau$.
— If $\Delta, \alpha <: \tau'; \Gamma \vdash_{<:} p : \sigma$ and $\tau'' <: \tau'$, then $\Delta, \alpha <: \tau''; \Gamma \vdash_{<:} p : \sigma$.

(e) Type substitution preserves subtyping:

— If $\Delta, \alpha <: \tau' \vdash_{<:} \tau_1 <: \tau_2$, then $\Delta \vdash_{<:} \tau_1\{\tau'/\alpha\} <: \tau_2\{\tau'/\alpha\}$.

(f) Type substitution preserves typing:

— If $\Delta, \alpha <: \tau'; \Gamma \vdash_{<:} e : \tau$ then $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{<:} e\{\tau'/\alpha\} : \tau\{\tau'/\alpha\}$.
— If $\Delta, \alpha <: \tau'; \Gamma \vdash_{<:} p : \sigma$ then $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{<:} p\{\tau'/\alpha\} : \sigma\{\tau'/\alpha\}$.

(g) Canonical forms:

— If $\vdash_{<:} v : t$, then $v$ has the form $c$ (for $c \in C$).
— If $\vdash_{<:} v : \tau_a \to \tau_b$, then either $v$ has the form $f$ (for $f \in F$) or $v$ has the form $\lambda x{:}\tau_a(e_a)$.
— If $\vdash_{<:} p : \forall \alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(\tau_a)$, then $p$ has the form $\Lambda \alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(e_a)$

*Proof*

(a) Proceed by simultaneous induction on the derivations $\Delta; \Gamma, x : \tau' \vdash_{<:} e : \tau$ and $\Delta; \Gamma, x : \tau' \vdash_{<:} p : \sigma$.

(b) Proceed by simultaneous induction on the derivations $\Delta; \Gamma, x : \sigma' \vdash_{<:} e : \tau$ and $\Delta; \Gamma, x : \sigma' \vdash_{<:} p : \sigma$.

(c) Proceed by induction on the derivation $\Delta, \alpha <: \tau' \vdash_{<:} \tau_1 <: \tau_2$.

(d) Proceed by simultaneous induction on the derivations $\Delta, \alpha : \tau'; \Gamma \vdash_{<:} e : \tau$ and $\Delta, \alpha : \tau'; \Gamma \vdash_{<:} p : \sigma$. We give the one interesting case of the induction. In the primitive operation case, $e = f$ ($f \in F$), $FV(\tau) = \langle \alpha_1, \ldots, \alpha_n \rangle$, and for all $\tau_1^* <: \tau_1, \ldots, \tau_n^* <: \tau_n$, we have $\tau\{\tau_1^*/\alpha_1, \ldots, \tau_n^*/\alpha_n\} \in \pi_F(f)$. If $\alpha \neq \alpha_i$, the result is immediate. If $\alpha = \alpha_i$, then for all $\tau_1^* <: \tau_1, \ldots, \tau_i^* <: \tau', \ldots, \tau_n^* <: \tau_n$, $\tau\{\tau_1^*/\alpha_1, \ldots, \tau_i^*/\alpha, \ldots, \tau_n^*/\alpha_n\} \in \pi_F(f)$ and $\Delta \vdash_{<:} \tau'' <: \tau'$ implies that for all $\tau_1^* <: \tau_1, \ldots, \tau_i^* <: \tau'', \ldots, \tau_n^* <: \tau_n$, $\tau\{\tau_1^*/\alpha_1, \ldots, \tau_i^*/\alpha_i, \ldots, \tau_n^*/\alpha_n\} \in \pi_F(f)$. Thus, $\Delta, \alpha <: \tau''; \Gamma \vdash_{<:} f : \tau$.

(e) Proceed by induction on the derivation $\Delta, \alpha <: \tau' \vdash_{<:} \tau_1 <: \tau_2$.

(f) Proceed by simultaneous induction on the derivations $\Delta, \alpha <: \tau'; \Gamma \vdash_{<:} e : \tau$ and $\Delta, \alpha <: \tau'; \Gamma \vdash_{<:} p : \sigma$. We give the interesting cases of the induction.

In the primitive operation case, $e = f$ ($f \in F$), $FV(\tau) = \langle \alpha_1, \ldots, \alpha_n \rangle$, and for all $\tau_1^* <: \tau_1, \ldots, \tau_n^* <: \tau_n$ we have $\tau\{\tau_1^*/\alpha_1, \ldots, \tau_n^*/\alpha_n\} \in \pi_F(f)$. Note $f\{\tau'/\alpha\} = f$. If $\alpha \neq \alpha_i$, the result is immediate. If $\alpha = \alpha_i$, then for all $\tau_1^* <: \tau_1, \ldots, \tau_i^* <: \tau', \ldots, \tau_n^* <: \tau_n$ we have $\tau\{\tau_1^*/\alpha_1, \ldots, \tau_i^*/\alpha, \ldots, \tau_n^*/\alpha_n\} \in \pi_F(f)$, which implies that for all $\tau_1^* <: \tau_1, \ldots, \tau_{i-1}^* <: \tau_{i-1}, \tau_{i+1}^* <: \tau_{i+1}, \ldots, \tau_n^* <: \tau_n$, we have $\tau\{\tau'/\alpha\}\{\tau_1^*/\alpha_1, \ldots, \tau_{i-1}^*/\alpha_{i-1}, \tau_{i+1}^*/\alpha_{i+1}, \ldots, \tau_n^*/\alpha_n\} \in \pi_F(f)$. Thus, $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{<:} f : \tau\{\tau'/\alpha\}$.

In the type abstraction case, $p = \Lambda\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(e_a)$, $\sigma = \forall\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(\tau_a)$, and $\Delta, \alpha <: \tau', \alpha_1 <: \tau_{a,1}, \ldots, \alpha_n <: \tau_{a,n}; \Gamma \vdash_{<:} e_a : \tau_a$. Assume $\alpha_1, \ldots, \alpha_n \neq \alpha$. Note $(\Lambda\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(e_a))\{\tau'/\alpha\} = \Lambda\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(e_a\{\tau'/\alpha\})$ and $(\forall\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(\tau_a)\{\tau'/\alpha\} = \forall\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(\tau_a\{\tau'/\alpha\})$ (because type variables are precluded from the types of quantified type variables). Furthermore, $\Delta, \alpha_1 <: \tau_{a,1}, \ldots, \alpha_n <: \tau_{a,n}, \alpha <: \tau'; \Gamma \vdash_{<:} e_a : \tau_a$. By the induction hypothesis, $\Delta, \alpha_1 <: \tau_{a,1}, \ldots, \alpha_n <: \tau_{a,n}; \Gamma\{\tau'/\alpha\} \vdash_{<:} e_a\{\tau'/\alpha\} : \tau_a\{\tau'/\alpha\}$. Hence, $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{<:} (\Lambda\alpha_1{<:}\tau_{a,1}, \ldots, \alpha_n{<:}\tau_{a,n}(e_a))\{\tau'/\alpha\} : \sigma\{\tau'/\alpha\}$.

(g) For the first part, proceed by case analysis of the derivation $\vdash_{<:} v : t$. For the second part, proceed by case analysis of the derivation $\vdash_{<:} v : \tau_a \to \tau_b$. For the third part, proceed by case analysis of $p$. □

*Theorem 5.1*
If $\pi_F$ is sound with respect to $\delta$, $\vdash_{<:} e : \tau$ and $e \longrightarrow_{<:}^* e'$, then $\vdash_{<:} e' : \tau$ and either $e'$ is a value or there exists $e''$ such that $e' \longrightarrow_{<:} e''$.

*Proof*
This is a standard proof of soundness, relying on progress and preservation lemmas:

- Progress: if $\pi_F$ is sound with respect to $\delta$ and $\vdash_{<:} e : \tau$, then either $e$ is a value or there exists $e'$ such that $e \longrightarrow_{<:} e'$. This follows by induction on the derivation $\vdash_{<:} e : \tau$. The only interesting case is the application case $e = e_1\ e_2$ when $e_1$ has the form $f$ (for $f \in F$) and $e_2$ is a value. Then $\tau_a \to \tau = t_a \to t$ for $t_a, t \in T$ and $t_a \to t \in \pi_F(f)$ by the typing judgment for primitive operations. By part 1 of Lemma A.1(g), $e_2$ has the form $c$ (for $c \in C$). Hence, $\vdash_{<:} f\ c : \tau$ and $\delta(f, c)$ is defined by the definition of $\pi_F$ sound with respect to $\delta$, and the primitive step applies to $e$.
- Preservation: if $\pi_F$ is sound with respect to $\delta$, $\vdash_{<:} e : \tau$ and $e \longrightarrow_{<:} e'$, then $\vdash_{<:} e' : \tau$. This follows by induction on the derivation $\vdash_{<:} e : \tau$. The only interesting case is the application case $e = e_1\ e_2$, when $e_1 = f$ (for $f \in F$), $e_2 = c$ (for $c \in C$), and $e' = \delta(f, c)$. the result follows by the definition of $\pi_F$ sound with respect to $\delta$.

To prove soundness, we assume $\vdash_{<:} e : \tau$ and $e \longrightarrow_{<:}^* e'$. Then $e \longrightarrow_{<:}^n e'$ for some $n$. Proceed by induction on $n$. In the base case, the theorem is equivalent to progress. In the step case, the inductive hypothesis, preservation, and progress suffice to prove the theorem. □

## B The Calculus $\lambda_{\mathsf{T}}^{\mathrm{DM}}$

**Value:**

| | |
|---|---|
| $v ::=$ | values |
| $c$ | base constant ($c \in C$) |
| $f$ | primitive operation ($f \in F$) |
| $\lambda x{:}\tau(e)$ | functional abstraction |

**Evaluation Contexts:**

| | |
|---|---|
| $E ::=$ | evaluation contexts |
| $[\,]$ | empty context |
| $E\ e$ | application context |
| $v\ E$ | argument context |
| $E\ [\tau_1, \ldots, \tau_n]$ | type application context |
| $\textbf{let}\ x = E\ \textbf{in}\ e$ | local binding context |

**Operational Semantics:**

$(\lambda x{:}\tau(e))\ v \longrightarrow_{\mathsf{T}} e\{v/x\}$
$(\Lambda\alpha_1, \ldots, \alpha_n(e))\ [\tau'_1, \ldots, \tau'_n] \longrightarrow_{\mathsf{T}} e\{\tau'_1/\alpha_1, \ldots, \tau'_n/\alpha_n\}$
$\textbf{let}\ x = p\ \textbf{in}\ e \longrightarrow_{\mathsf{T}} e\{p/x\}$
$f\ c \longrightarrow_{\mathsf{T}} c'$ iff $\delta(f, c) = c'$
$E[e_1] \longrightarrow_{\mathsf{T}} E[e_2]$ iff $e_1 \longrightarrow_{\mathsf{T}} e_2$

The function $\delta : F \times C \rightharpoonup C$ is a partial function defining the result of applying a primitive operation to a base constant.

**Typing Contexts:**

| | |
|---|---|
| $\Gamma ::=$ | type environments |
| $\cdot$ | empty |
| $\Gamma, x : \tau$ | type |
| $\Gamma, x : \sigma$ | type scheme |
| $\Delta ::=$ | type variable environments |
| $\cdot$ | empty |
| $\Delta, \alpha$ | type variable |

**Judgments:**

| | |
|---|---|
| $\Delta \vdash_{\mathsf{T}} \Gamma$ ctxt | good context $\Gamma$ |
| $\Delta \vdash_{\mathsf{T}} \tau$ type | good type $\tau$ |
| $\Delta \vdash_{\mathsf{T}} \sigma$ scheme | good type scheme $\sigma$ |
| $\Delta; \Gamma \vdash_{\mathsf{T}} e : \tau$ | good expression $e$ with type $\tau$ |
| $\Delta; \Gamma \vdash_{\mathsf{T}} p : \sigma$ | good expression $p$ with type scheme $\sigma$ |

**Judgment** $\Delta \vdash_{\mathsf{T}} \Gamma$ ctxt**:**

$$\frac{}{\Delta \vdash_{\mathsf{T}} \cdot \text{ ctxt}} \qquad \frac{\Delta \vdash_{\mathsf{T}} \Gamma \text{ ctxt}}{\Delta \vdash_{\mathsf{T}} \Gamma, x : \tau \text{ ctxt}} \qquad \frac{\Delta \vdash_{\mathsf{T}} \Gamma \text{ ctxt}}{\Delta \vdash_{\mathsf{T}} \Gamma, x : \sigma \text{ ctxt}}$$

**Judgment** $\Delta \vdash_{\mathsf{T}} \tau$ type**:**

$$\frac{}{\Delta \vdash_{\mathsf{T}} t \text{ type}} \qquad \frac{\vdash_{\mathsf{T}} \Delta \text{ ctxt} \quad \alpha \in \text{dom}(\Delta)}{\Delta \vdash_{\mathsf{T}} \alpha \text{ type}} \qquad \frac{\Delta \vdash_{\mathsf{T}} \tau_1 \text{ type} \quad \Delta \vdash_{\mathsf{T}} \tau_2 \text{ type}}{\Delta \vdash_{\mathsf{T}} \tau_1 \rightarrow \tau_2 \text{ type}}$$

**Judgment** $\Delta \vdash_{\mathsf{T}} \sigma$ scheme**:**

$$\frac{\Delta, \alpha_1, \ldots, \alpha_n \vdash_{\mathsf{T}} \tau \text{ type}}{\Delta \vdash_{\mathsf{T}} \forall \alpha_1, \ldots, \alpha_n(\tau) \text{ scheme}}$$

**Judgment** $\Delta; \Gamma \vdash_{\mathsf{T}} e : \tau$**:**

$$\frac{}{\Delta; \Gamma \vdash_{\mathsf{T}} c : \pi_C(c)} \ (c \in C)$$

$$\frac{\begin{array}{c}\text{For all } \tau' \in \pi_C(C) \text{ such that} \\ \mathit{unify}(\tau_1, \tau') = \langle(\alpha_1, \tau_1'), \ldots, (\alpha_n, \tau_n'), \ldots\rangle, \\ (\tau_1 \rightarrow \tau_2)\{\tau_1'/\alpha_1, \ldots, \tau_n'/\alpha_n\} \in \pi_F(f)\end{array}}{\Delta, \alpha_1, \ldots, \alpha_n; \Gamma \vdash_{\mathsf{T}} f : \tau_1 \rightarrow \tau_2} \ \left(\begin{array}{c} f \in F, \\ FV(\tau_1) = \langle \alpha_1, \ldots, \alpha_n \rangle \end{array}\right)$$

$$\frac{\Delta \vdash_{\mathsf{T}} \Gamma \text{ ctxt}}{\Delta; \Gamma, x : \tau \vdash_{\mathsf{T}} x : \tau} \qquad \frac{\Delta; \Gamma, x : \tau \vdash_{\mathsf{T}} e : \tau'}{\Delta; \Gamma \vdash_{\mathsf{T}} \lambda x{:}\tau(e) : \tau \rightarrow \tau'}$$

$$\frac{\Delta; \Gamma \vdash_{\mathsf{T}} e_1 : \tau_1 \rightarrow \tau_2 \quad \Delta; \Gamma \vdash_{\mathsf{T}} e_2 : \tau_1}{\Delta; \Gamma \vdash_{\mathsf{T}} e_1\ e_2 : \tau_2}$$

$$\frac{\Delta; \Gamma \vdash_{\mathsf{T}} p : \forall \alpha_1, \ldots, \alpha_n(\tau)}{\Delta; \Gamma \vdash_{\mathsf{T}} p\ [\tau_1, \ldots, \tau_n] : \tau\{\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\}} \qquad \frac{\Delta; \Gamma, x : \sigma \vdash_{\mathsf{T}} e : \tau \quad \Delta; \Gamma \vdash_{\mathsf{T}} p : \sigma}{\Delta; \Gamma \vdash_{\mathsf{T}} \textbf{let } x = p \textbf{ in } e : \tau}$$

**Judgment** $\Delta; \Gamma \vdash_{\mathsf{T}} p : \sigma$**:**

$$\frac{\Delta \vdash_{\mathsf{T}} \Gamma \text{ ctxt} \quad \Delta, \alpha_1, \ldots, \alpha_n; \Gamma \vdash_{\mathsf{T}} e : \tau}{\Delta; \Gamma \vdash_{\mathsf{T}} \Lambda \alpha_1, \ldots, \alpha_n(e) : \forall \alpha_1, \ldots, \alpha_n(\tau)} \ (\alpha_1, \ldots, \alpha_n \notin \Delta)$$

### *B.1 Proofs*

The proof of soundness for $\lambda_{\mathsf{T}}^{\text{DM}}$ is mostly standard, relying on preservation and progress lemmas. As we did for $\lambda_{<:}^{\text{DM}}$, we present all the lemmas needed to derive the proof, but leave most of the straightforward details to the reader.

*Lemma B.1*

(a) Monomorphic expression substitution preserves typing:

— If $\Delta; \Gamma, x : \tau' \vdash_{\mathsf{T}} e : \tau$ and $\Delta; \Gamma \vdash_{\mathsf{T}} e' : \tau'$, then $\Delta; \Gamma \vdash_{\mathsf{T}} e\{e'/x\} : \tau$.
— If $\Delta; \Gamma, x : \tau' \vdash_{\mathsf{T}} p : \sigma$ and $\Delta; \Gamma \vdash_{\mathsf{T}} e' : \tau'$, then $\Delta; \Gamma \vdash_{\mathsf{T}} p\{e'/x\} : \tau$.

(b) Polymorphic expression substitution preserves typing:

— If $\Delta; \Gamma, x : \sigma' \vdash_{\mathsf{T}} e : \tau$ and $\Delta; \Gamma \vdash_{\mathsf{T}} p' : \sigma'$, then $\Delta; \Gamma \vdash_{\mathsf{T}} e\{p'/x\} : \tau$.
— If $\Delta; \Gamma, x : \sigma' \vdash_{\mathsf{T}} p : \sigma$ and $\Delta; \Gamma \vdash_{\mathsf{T}} p' : \sigma'$, then $\Delta; \Gamma \vdash_{\mathsf{T}} p\{p'/x\} : \sigma$.

(c) Type substitution preserves typing:

— If $\Delta, \alpha; \Gamma \vdash_{\mathsf{T}} e : \tau$ then $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{\mathsf{T}} e\{\tau'/\alpha\} : \tau\{\tau'/\alpha\}$.
— If $\Delta, \alpha; \Gamma \vdash_{\mathsf{T}} p : \sigma$ then $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{\mathsf{T}} p\{\tau'/\alpha\} : \sigma\{\tau'/\alpha\}$.

(d) Canonical forms:

— If $\vdash_{\mathsf{T}} v : \mathsf{T}\ \tau$, then $v$ has the form $c$ (for $c \in C$).
— If $\vdash_{\mathsf{T}} v : \tau_a \to \tau_b$, then either $v$ has the form $f$ (for $f \in F$) or $v$ has the form $\lambda x{:}\tau_a(e_a)$.
— If $\vdash_{\mathsf{T}} p : \forall \alpha_1, \ldots, \alpha_n(\tau_a)$, then $p$ has the form $\Lambda \alpha_1, \ldots, \alpha_n(e_a)$

*Proof*

(a) Proceed by simultaneous induction on the derivations $\Delta; \Gamma, x : \tau' \vdash_{\mathsf{T}} e : \tau$ and $\Delta; \Gamma, x : \tau' \vdash_{\mathsf{T}} p : \sigma$.

(b) Proceed by simultaneous induction on the derivations $\Delta; \Gamma, x : \sigma' \vdash_{\mathsf{T}} e : \tau$ and $\Delta; \Gamma, x : \sigma' \vdash_{\mathsf{T}} p : \sigma$.

(c) Proceed by simultaneous induction on the derivations $\Delta, \alpha; \Gamma \vdash_{\mathsf{T}} e : \tau$ and $\Delta, \alpha; \Gamma \vdash_{\mathsf{T}} p : \sigma$. We prove the interesting cases of the induction.

In the primitive operation case, $e = f$ ($f \in F$), $\tau = \tau_1 \to \tau_2$, $FV(\tau_1) = \langle \alpha_1, \ldots, \alpha_n \rangle$, and for all $\tau_* \in \pi_C(C)$ such that $\mathit{unify}(\tau_*, \tau_1) = \langle (\alpha_1, \tau_1'), \ldots, (\alpha_n, \tau_n'), \ldots \rangle$, we have $\tau_1 \to \tau_2\{\tau_1'/\alpha_1, \ldots, \tau_n'/\alpha_n\} \in \pi_F(f)$. Note $f\{\tau'/\alpha\} = f$. If $\alpha \neq \alpha_i$ for any $i$, the result is immediate. If $\alpha = \alpha_i$ for some $i$ (without loss of generality, let $i = 1$), then for any $\tau_* \in \pi_C(C)$ such that $\mathit{unify}(\tau_*, \tau_1\{\tau'/\alpha_1\}) = \langle (\alpha_2, \tau_2'), \ldots, (\alpha_n, \tau_n'), \ldots \rangle$, we have $\mathit{unify}(\tau_*, \tau_1) = \langle (\alpha_1, \tau'), (\alpha_2, \tau_2'), \ldots, (\alpha_n, \tau_n'), \ldots \rangle$, so that $(\tau_1 \to \tau_2)\{\tau'/\alpha\}\{\tau_2'/\alpha_2, \ldots, \tau_n'/\alpha_n\} = (\tau_1 \to \tau_2)\{\tau'/\alpha_1, \tau_2'/\alpha_2, \ldots, \tau_n'/\alpha_n\} \in \pi_F(f)$. Thus, $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{\mathsf{T}} f : \tau\{\tau'/\alpha\}$.

In the type abstraction case, $p = \Lambda \alpha_1, \ldots, \alpha_n(e_a)$, $\sigma = \forall \alpha_1, \ldots, \alpha_n(\tau_a)$, and $\Delta, \alpha, \alpha_1, \ldots, \alpha_n; \Gamma \vdash_{\mathsf{T}} e_a : \tau_a$. Assume $\alpha_1, \ldots, \alpha_n \neq \alpha$. Note $(\Lambda \alpha_1, \ldots, \alpha_n(e_a))\{\tau'/\alpha\} = \Lambda \alpha_1, \ldots, \alpha_n(e_a\{\tau'/\alpha\})$ and $(\forall \alpha_1, \ldots, \alpha_n(\tau_a))\{\tau'/\alpha\} = \forall \alpha_1, \ldots, \alpha_n(\tau_a\{\tau'/\alpha\})$ (because type variables are precluded from the types of quantified type variables). Furthermore, $\Delta, \alpha_1, \ldots, \alpha_n, \alpha; \Gamma \vdash_{\mathsf{T}} e_a : \tau_a$. By the induction hypothesis, $\Delta, \alpha_1, \ldots, \alpha_n; \Gamma\{\tau'/\alpha\} \vdash_{\mathsf{T}} e_a\{\tau'/\alpha\} : \tau_a\{\tau'/\alpha\}$. Hence, $\Delta; \Gamma\{\tau'/\alpha\} \vdash_{\mathsf{T}} (\Lambda \alpha_1, \ldots, \alpha_n(e_a))\{\tau'/\alpha\} : \sigma\{\tau'/\alpha\}$.

(d) For the first part, proceed by case analysis of the derivation $\vdash_{\mathsf{T}} v : \mathsf{T}\ \tau$. For the second part, proceed by case analysis of the derivation $\vdash_{\mathsf{T}} v : \tau_a \rightarrow \tau_b$. For the third part, proceed by case analysis of $p$. $\square$

*Theorem 5.2*
If $\pi_F$ is sound with respect to $\delta$, $\vdash_{\mathsf{T}} e : \tau$ and $e \longrightarrow_{\mathsf{T}}^{*} e'$, then $\vdash_{\mathsf{T}} e' : \tau$ and either $e'$ is a value or there exists $e''$ such that $e' \longrightarrow_{\mathsf{T}} e''$.

*Proof*
This is a standard proof of soundness, relying on progress and preservation lemmas:

- Progress: if $\pi_F$ is sound with respect to $\delta$ and $\vdash_{\mathsf{T}} e : \tau$, then either $e$ is a value or there exists $e'$ such that $e \longrightarrow_{\mathsf{T}} e'$. This follows by induction on the derivation $\vdash_{\mathsf{T}} e : \tau$. The interesting case of the induction is the application case $e = e_1\ e_2$ when $e_1$ has the form $f$ (for $f \in F$) and $e_2$ is a value. Then, $\tau_a \rightarrow \tau = t_a \rightarrow t$ for $t_a, t \in T$ and $t_a \rightarrow t \in \pi_F(f)$ by the typing judgment for primitive operations. By part 1 of Lemma B.1, $e_2$ has the form $c$ (for $c \in C$). Hence, $\vdash_{\mathsf{T}} f\ c : \tau$ and $\delta(f, c)$ is defined by the definition of $\pi_F$ sound with respect to $\delta$. Thus, the primitive step applies to $e$.
- Preservation: if $\pi_F$ is sound with respect to $\delta$, $\vdash_{\mathsf{T}} e : \tau$ and $e \longrightarrow_{\mathsf{T}} e'$, then $\vdash_{\mathsf{T}} e' : \tau$. This follows by induction on the derivation $\vdash_{\mathsf{T}} e : \tau$. Again, the interesting case is the application case $e = e_1\ e_2$ when $e_1 = f$ (for $f \in F$), $e_2 = c$ (for $c \in C$), and $e' = \delta(f, c)$. The result follows by the definition of $\pi_F$ sound with respect to $\delta$.

To prove soundness, we assume $\vdash_{\mathsf{T}} e : \tau$ and $e \longrightarrow_{\mathsf{T}}^{*} e'$. Then $e \longrightarrow_{\mathsf{T}}^{n} e'$ for some $n$. Proceed by induction on $n$. In the base case, the theorem is equivalent to progress. In the step case, the inductive hypothesis, preservation, and progress suffice to prove the theorem. $\square$

## C Translation Proofs

*Theorem 5.3*
If $\pi_F$ is sound with respect to $\delta$ in $\lambda_{<:}^{\mathrm{DM}}$, then $\mathcal{T}[\![\pi_F]\!]$ is sound with respect to $\delta$ in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$.

*Proof*
We need to show that for all $f \in F$ and $c \in C$ such that $\vdash_{\mathsf{T}} f\ c : \tau$ for some $\tau$, then $\delta(f, c)$ is defined, and that $\mathcal{T}[\![\pi_C]\!](\delta(f, c)) = \tau$. Given $f \in F$ and $c \in C$, assume that $\vdash_{\mathsf{T}} f\ c : \tau$. This means that $\vdash_{\mathsf{T}} f : \tau' \rightarrow \tau$ and that $\vdash_{\mathsf{T}} c : \tau'$. From $\vdash_{\mathsf{T}} f : \tau' \rightarrow \tau$, we derive that for all $\tau^* \in \mathcal{T}[\![\pi_C]\!](C)$ such that $\mathit{unify}(\tau^*, \tau') \neq \varnothing$ (since $\tau'$ and $\tau^*$ are both closed types), $\tau' \rightarrow \tau \in \mathcal{T}[\![\pi_F]\!](f)$. By definition of $\mathcal{T}$, and by assumption on the form of $\pi_F$, this means that $\tau' \rightarrow \tau$ is of the form $\mathcal{T}[\![t']\!] \rightarrow \mathcal{T}[\![t]\!]$, with $\mathcal{T}[\![t']\!] = \tau'$ and $\mathcal{T}[\![t]\!] = \tau$. Hence, $\vdash_{<:} f : t' \rightarrow t$. From $\vdash_{\mathsf{T}} c : \tau'$, we derive that $\mathcal{T}[\![t']\!] = \tau' = \mathcal{T}[\![\pi_C]\!](c) = \mathcal{T}[\![\pi_C(c)]\!]$. Hence, $\pi_C(c) = t'$, and $\vdash_{<:} c : t'$. We can therefore infer that $\vdash_{<:} f\ c : t$. Therefore, by soundness of $\pi_F$ with respect to $\delta$ in $\lambda_{<:}^{\mathrm{DM}}$, we get that $\delta(f, c)$ is defined, and that $\pi_C(\delta(f, c)) = t$. Thus, $\mathcal{T}[\![\pi_C]\!](\delta(f, c)) = \mathcal{T}[\![\pi_C(\delta(f, c))]\!] = \mathcal{T}[\![t]\!] = \tau$, as required. $\square$

The following lemma, relating the correctness of the subtype encoding and substitution, is used in the proof of Theorem 5.4.

*Lemma C.1*
For all $t$,$t'$, and $\tau$ with $FV(\tau) \subseteq \langle\alpha\rangle$, if $t^A = \langle t\rangle_A$, $FV(t^A) = \langle\alpha_1,\ldots,\alpha_n\rangle$, and $\mathit{unify}(\langle t'\rangle_C, t^A) = \langle(\alpha_1,\tau_1),\ldots,(\alpha_n,\tau_n),\ldots\rangle$, then $\mathcal{T}[\![\tau\{t'/\alpha\}]\!] = \mathcal{T}[\![\tau]\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\}$.

*Proof*
We proceed by induction on the structure of $\tau$.

For $\tau = \alpha$, we immediately get that $\mathcal{T}[\![\tau\{t'/\alpha\}]\!] = \mathcal{T}[\![t']\!] = \langle t'\rangle_C$. Moreover, we have $\mathcal{T}[\![\tau']\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} = t^A\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} = \langle t'\rangle_C$, by the assumption on the unification of $\langle t'\rangle_C$ and $t^A$.

For $\tau = t^*$ for some $t^*$, then $\mathcal{T}[\![\tau\{t'/\alpha\}]\!] = \mathcal{T}[\![\tau]\!] = \langle t^*\rangle_C$. Moreover, $\mathcal{T}[\![\tau]\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} = \mathcal{T}[\![t^*]\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} = \langle t^*\rangle_C\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} = \langle t^*\rangle_C$.

Finally, for $\tau = \tau' \rightarrow \tau''$, we have $\mathcal{T}[\![(\tau' \rightarrow \tau'')\{t'/\alpha\}]\!] = \mathcal{T}[\![\tau'\{t'/\alpha\} \rightarrow \tau''\{t'/\alpha\}]\!] = \mathcal{T}[\![\tau'\{t'/\alpha\}]\!] \rightarrow \mathcal{T}[\![\tau''\{t'/\alpha\}]\!]$. By applying the induction hypothesis, this is equal to $\mathcal{T}[\![\tau']\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} \rightarrow \mathcal{T}[\![\tau'']\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\} = \mathcal{T}[\![\tau' \rightarrow \tau'']\!][\alpha \mapsto t^A]\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\}$, as required. □

*Theorem 5.4*
If $\vdash_{<:} e : \tau$, then $\vdash_{\mathsf{T}} \mathcal{E}[\![\vdash_{<:} e : \tau]\!] : \mathcal{T}[\![\tau]\!]$.

*Proof*
We prove a more general form of this theorem, namely that if $\Delta;\Gamma \vdash_{<:} e : \tau$, then $\mathcal{T}[\![\Delta]\!];\mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_{\mathsf{T}} \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} e : \tau]\!]\rho_\Delta : \mathcal{T}[\![\tau]\!]\rho_\Delta$, where:

$$\begin{aligned}\mathcal{T}[\![\alpha_1 <: \tau_1,\ldots,\alpha_n <: \tau_n]\!] \triangleq\ & \alpha_{11},\ldots,\alpha_{1k_1},\ldots,\alpha_{n1},\ldots,\alpha_{nk_n}\\ & \text{where } \tau_i^A = \mathcal{A}[\![\tau_i]\!]\\ & \text{and } FV(\tau_i^A) = \langle\alpha_{i1},\ldots,\alpha_{ik_i}\rangle\end{aligned}$$

and for $\Delta$ of the form $\alpha_1 <: \tau_1,\ldots,\alpha_n <: \tau_n$,

$$\rho_\Delta \triangleq \{\alpha_1 \mapsto \tau_1^A,\ldots,\alpha_n \mapsto \tau_n^A\}.$$

Similarly, we show that if $\Delta;\Gamma \vdash_{<:} p : \sigma$, then $\mathcal{T}[\![\Delta]\!];\mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_{\mathsf{T}} \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} p : \sigma]\!]\rho_\Delta : \mathcal{T}[\![\sigma]\!]\rho_\Delta$. We establish this by simultaneous induction on the derivations $\Delta;\Gamma \vdash_{<:} e : \tau$ and $\Delta;\Gamma \vdash_{<:} p : \sigma$.

For variables, $\Delta,\Gamma \vdash_{<:} x : \tau$ implies that $x : \tau$ is in $\Gamma$. Hence, $x : \mathcal{T}[\![\tau]\!]\rho_\Delta$ is in $\mathcal{T}[\![\Gamma]\!]\rho_\Delta$. Hence, $\mathcal{T}[\![\Delta]\!];\mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_{\mathsf{T}} x : \mathcal{T}[\![\tau]\!]\rho_\Delta$. Similarly for $\Delta;\Gamma \vdash_{<:} x : \sigma$.

For constants $c \in C$, if $\Delta;\Gamma \vdash_{<:} c : \tau$, then we have $\pi_C(c) = \tau$. Hence, $\mathcal{T}[\![\pi_C(c)]\!]\rho_\Delta = \mathcal{T}[\![\tau]\!]\rho_\Delta$, and by definition, $\mathcal{T}[\![\pi_C]\!]\rho_\Delta(c) = \mathcal{T}[\![\tau]\!]\rho_\Delta$. This implies $\mathcal{T}[\![\Delta]\!];\mathcal{T}[\![\Gamma]\!] \vdash_{\mathsf{T}} c : \mathcal{T}[\![\tau]\!]\rho_\Delta$.

For operations $f \in F$, if $\Delta,\alpha_1 <: \tau_1,\ldots,\alpha_n <: \tau_n;\Gamma \vdash_{<:} f : \tau' \rightarrow \tau$ (where $FV(\tau') = \langle\alpha_1,\ldots,\alpha_n\rangle$). Hence, for all $\tau_i' <: \tau_i$, we have $(\tau' \rightarrow \tau)\{\tau_1'/\alpha_1,\ldots,\tau_n'/\alpha_n\} \in \pi_F(f)$. Note that this implies that each $\tau_i'$ is of the form $t_i'$ for some $t_i'$, due to the restrictions imposed on $\pi_F$. Furthermore, also due to the restrictions imposed on $\pi_F$, we must have that $\tau'$ is either $t'$ for some $t'$, or a type variable $\alpha_1$. We need to show that for all $\tau^* \in \mathcal{T}[\![\pi_C]\!](C)$, if $\mathit{unify}(\tau^*, \mathcal{T}[\![\tau']\!]) = \langle(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n), \ldots\rangle$, then $\mathcal{T}[\![\tau' \to \tau]\!]\{\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\} \in \mathcal{T}[\![\pi_F]\!](f)$. Take an arbitrary $\tau^* \in \mathcal{T}[\![\pi_C]\!](C)$. By restrictions on $\pi_C$, $\tau^*$ is of the form $\langle t^*\rangle_C$ for some $t^* \in \pi_C(C)$. Now, consider the different forms of $\tau'$. In the case $\tau' = t'$, we have $\mathcal{T}[\![\tau']\!] = \langle t'\rangle_C$, so that if $\mathit{unify}(\langle t^*\rangle_C, \langle t'\rangle_C)$, then $t^* = t'$. Moreover, because we assumed that concrete encodings did not introduce free type variables, then $FV(\tau') = \varnothing$. Thus, $\mathcal{T}[\![\tau' \to \tau]\!] = \mathcal{T}[\![\tau']\!] \to \mathcal{T}[\![\tau]\!] \in \mathcal{T}[\![\pi_F]\!](f)$ follows immediately from the fact that $\tau' \to \tau \in \pi_F(f)$. In the case that $\tau' = \alpha_1$, then $\mathcal{T}[\![\tau']\!]\rho = \langle t'_1\rangle_A$. Let $FV(t'_1) = \langle \alpha_{11}, \ldots, \alpha_{1k_1}\rangle$. Assume $\mathit{unify}(\langle t^*\rangle_C, {t'}^A) = \langle(\alpha_{11}, \tau_1), \ldots, (\alpha_{1k_1}, \tau_{k_1}), \ldots\rangle$. Because the encoding is respectful, $\mathit{unify}(\langle t^*\rangle_C, {t'}^A) \neq \varnothing$ if and only if $t^* \leq t_1$, that is, $t^* <: t_1$. By assumption, we have $(\tau' \to \tau)\{t^*/\alpha_1\} \in \pi_F(f)$. Therefore, $\mathcal{T}[\![(\tau' \to \tau)\{t^*/\alpha_1\}]\!] \in \mathcal{T}[\![\pi_F]\!](f)$. By Lemma C.1, $\mathcal{T}[\![(\tau' \to \tau)\{t^*/\alpha_1\}]\!] = \mathcal{T}[\![\tau' \to \tau]\!]\{\tau_1/\alpha_{11}, \ldots, \tau_{k_1}/\alpha_{1k_1}\}$, and the result follows. Since $\tau^*$ was arbitrary, we can therefore infer that $\mathcal{T}[\![\Delta]\!], \alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}; \mathcal{T}[\![\Gamma]\!]\rho_\Delta[\alpha_i \mapsto \tau_i^A] \vdash_\top f : \mathcal{T}[\![\tau' \to \tau]\!]\rho_\Delta[\alpha_i \mapsto \tau_i^A]$.

For abstractions, if $\Delta; \Gamma \vdash_{<:} \lambda x{:}\tau'(e) : \tau' \to \tau$, then $\Delta; \Gamma, x : \tau' \vdash_{<:} e : \tau$. By the induction hypothesis, $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta; x : \mathcal{T}[\![\tau']\!]\rho_\Delta \vdash_\top \mathcal{E}[\![e]\!]\rho_\Delta : \mathcal{T}[\![\tau]\!]\rho_\Delta$, from which one can infer that $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \lambda x{:}\mathcal{T}[\![\tau']\!]\rho_\Delta(\mathcal{E}[\![e]\!]\rho_\Delta) : \mathcal{T}[\![\tau']\!]\rho_\Delta \to \mathcal{T}[\![\tau]\!]\rho_\Delta$, which yields $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \lambda x{:}\mathcal{T}[\![\tau']\!]\rho_\Delta(\mathcal{E}[\![e]\!]\rho_\Delta) : \mathcal{T}[\![\tau' \to \tau]\!]\rho_\Delta$.

For applications, if $\Delta; \Gamma \vdash_{<:} e_1\ e_2 : \tau$, then for some $\tau'$, $\Delta; \Gamma \vdash_{<:} e_1 : \tau' \to \tau$ and $\Delta; \Gamma \vdash_{<:} e_2 : \tau'$. By the induction hypothesis, we have $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![e_1]\!] : \mathcal{T}[\![\tau' \to \tau]\!]$, so that $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![e_1]\!] : \mathcal{T}[\![\tau']\!] \to \mathcal{T}[\![\tau]\!]$, and $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![e_2]\!] : \mathcal{T}[\![\tau']\!]$. This yields that $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top (\mathcal{E}[\![e_1]\!]\rho_\Delta)\ \mathcal{E}[\![e_2]\!]\rho_\Delta : \mathcal{T}[\![\tau]\!]\rho_\Delta$.

For local bindings, if $\Delta; \Gamma \vdash_{<:} \textbf{let}\ x = p\ \textbf{in}\ e : \tau$, then for some $\sigma$ we have $\Delta; \Gamma, x : \sigma \vdash_{<:} e : \tau$ and $\Delta; \Gamma \vdash_{<:} p : \sigma$. By the induction hypothesis, we have $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta, x : \mathcal{T}[\![\sigma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![e]\!] : \mathcal{T}[\![\tau]\!]\rho_\Delta$ and $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![p]\!]\rho_\Delta : \mathcal{T}[\![\sigma]\!]\rho_\Delta$. Thus, we have $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \textbf{let}\ x = \mathcal{E}[\![p]\!]\rho_\Delta\ \textbf{in}\ \mathcal{E}[\![e]\!]\rho_\Delta : \mathcal{T}[\![\tau]\!]\rho_\Delta$.

For type applications, we have $\Delta; \Gamma \vdash_{<:} p\ [\tau'_1, \ldots, \tau'_n] : \tau$. Then for all $(\alpha_i, \tau_i)$ in $\mathcal{B}[\![p]\!]\Gamma$, we have $\Delta; \Gamma \vdash_{<:} p : \forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau')$, $\Delta \vdash_{<:} \tau'_i <: \tau_i$ for all $i$, and $\tau = \tau'\{\tau'_1/\alpha_1, \ldots, \tau'_n/\alpha_n\}$. By the induction hypothesis, $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![p]\!]\rho_\Delta : \mathcal{T}[\![\forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau')]\!]\rho_\Delta$. Let $\tau_i^A = \mathcal{A}[\![\tau_i]\!]$, and $FV(\tau_i^A) = \langle \alpha_{i1}, \ldots, \alpha_{ik_i}\rangle$. Thus, $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![p]\!]\rho_\Delta : \forall \alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}(\mathcal{T}[\![\tau']\!]\rho_\Delta[\alpha_i \mapsto \tau_i^A])$. We know that $\Delta \vdash_{<:} \tau'_i <: \tau_i$ for all $i$, so we have that $\mathit{unify}(\tau_i^A, \mathcal{T}[\![\tau_i]\!]\rho_\Delta) = \langle(\alpha_{i1}, \tau_{i1}), \ldots, (\alpha_{ik_i}, \tau_{ik_i}), \ldots\rangle$, and $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![p]\!]\rho_\Delta\ [\tau_{11}, \ldots, \tau_{1k_1}, \ldots, \tau_{n1}, \ldots, \tau_{nk_n}] : \mathcal{T}[\![\tau']\!]\rho_\Delta[\alpha_i \mapsto \tau_i^A]$, as required.

For type abstractions, we have $\Delta; \Gamma \vdash_{<:} \Lambda \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(e) : \forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau)$. Thus, we have $\Delta \vdash_{<:} \Gamma$ ctxt, that is, the type variables in $\Gamma$ appear in $\Delta$, and moreover $\Delta, \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n; \Gamma \vdash_{<:} e : \tau$. Let $\tau_i^A = \mathcal{A}[\![\tau_i]\!]$ and $FV(\tau_i^A) = \langle \alpha_{i1}, \ldots, \alpha_{ik_i}\rangle$, for $1 \leq i \leq n$. Let $\rho'_\Delta = \rho_\Delta[\alpha_1 \mapsto \tau_1^A, \ldots, \alpha_n \mapsto \tau_n^A]$. By the induction hypothesis, we have $\mathcal{T}[\![\Delta]\!], \alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}; \mathcal{T}[\![\Gamma]\!]\rho'_\Delta \vdash_\top \mathcal{E}[\![e]\!]\rho'_\Delta : \mathcal{T}[\![\tau]\!]\rho'_\Delta$. ¿From this we can infer that $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho'_\Delta \vdash_\top \Lambda \alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}(\mathcal{E}[\![e]\!]\rho'_\Delta) : \forall \alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}(\mathcal{T}[\![\tau]\!]\rho'_\Delta)$, which is easily seen equivalent to $\mathcal{T}[\![\Delta]\!]; \mathcal{T}[\![\Gamma]\!]\rho_\Delta \vdash_\top \mathcal{E}[\![\Lambda \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(e)]\!]\rho_\Delta : \mathcal{T}[\![\forall \alpha_1{<:}\tau_1, \ldots, \alpha_n{<:}\tau_n(\tau)]\!]\rho_\Delta$.

(We can replace $\mathcal{T}[\![\Gamma]\!]\rho'_\Delta$ by $\mathcal{T}[\![\Gamma]\!]\rho_\Delta$ by the assumption that $\Gamma$ is a good context in $\Delta$.) $\square$

## References


Blume, M. (2001). No-Longer-Foreign: Teaching an ML compiler to speak C "natively". *Proceedings of BABEL'01*. Electronic Notes in Theoretical Computer Science, vol. 59.1. Elsevier Science Publishers.

Burton, F. W. (1990). Type extension through polymorphism. *ACM Transactions on Programming Languages and Systems*, **12**(1), 135–138.

Cardelli, L., Martini, S., Mitchell, J. C., & Scedrov, A. (1994). An extension of System F with subtyping. *Information and Computation*, **109**(1/2), 4–56.

Damas, L., & Milner, R. (1982). Principal type-schemes for functional programs. *Pages 207–212 of: Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Elliott, C., Finne, S., & de Moor, O. (2000). Compiling embedded languages. *Workshop on Semantics, Applications, and Implementation of Program Generation*.

Finne, S., Leijen, D., Meijer, E., & Peyton Jones, S. (1999). Calling hell from heaven and heaven from hell. *Pages 114–125 of: Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

Fluet, M., & Pucella, R. (2005). Practical datatype specializations with phantom types and recursion schemes. *Pages 203–228 of: Proceedings of the ACM SIGPLAN Workshop on ML*. Elsevier Science Publishers.

Leijen, D., & Meijer, E. (1999). Domain specific embedded compilers. *Pages 109–122 of: Proceedings of the Second Conference on Domain-Specific Languages (DSL'99)*.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**(3), 348–375.

Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997). *The Definition of Standard ML (revised)*. Cambridge, Mass.: The MIT Press.

Pessaux, F., & Leroy, X. (1999). Type-based analysis of uncaught exceptions. *Pages 276–290 of: Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Rémy, D. (1989). Records and variants as a natural extension of ML. *Pages 77–88 of: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Reppy, J. H. (1996). *A safe interface to sockets*. Technical memorandum. AT&T Bell Laboratories.

Wand, M. (1987). Complete type inference for simple objects. *Proceedings of the 2nd Annual IEEE Symposium on Logic in Computer Science*.