Preliminary Investigation

of a Calculus with

Intersection and Union Types

Benjamin C. Pierce

June 29, 1990

Abstract

Type systems based on the concept of intersection types [3, 4, 5] have been extensively studied in recent years, both as tools for the analysis of the pure λ-calculus and, more recently, as the basis for practical programming languages such as Forsythe [16].

The dual construction, *union types*, also appears to be of potential practical interest. For example, by expressing more refined types for functions, union types allow a restricted form of abstract interpretation to be performed during typechecking.

This report summarizes the results of a preliminary investigation of a simple typed λ-calculus with intersection and union types. Section 1 introduces the concrete syntax and typing rules of the language. Section 2 presents a number of small programs illustrating various aspects of the language and indicating some of its expressive power. Section 3 gives a semantics in the form of rules for equality between typed terms and exhibits a PER model validating these rules. Section 4 discusses the decidability of typechecking. Section 5 relates this work to some other recent investigations of similar formulations of union types. Section 6 describes planned future work.

N.b. This report describes work in a preliminary stage of development. It is likely that further investigation of both semantic considerations (e.g. the validity of deMorgan's laws and the general categorical characterization of union types) and algorithmic issues (e.g. the correctness and efficiency of the typechecking procedure proposed here) will lead to substantial changes in the formulation of the language and its interpretation.

1 Syntax

1.1 Notational Conventions

This section introduces the notational conventions used in the rest of the report and defines the concrete syntax of a simple typed λ-calculus $\lambda^{\vee}$ ("lambda union") with intersection and union types.

The metavariables σ, τ, θ, ϕ, and ψ range over types; ρ ranges over primitive types; `e`, `f`, and `b` range over terms; `x` ranges over a set VAR of variables.

The notation $\tau_1, \tau_2, \ldots, \tau_n$ denotes a finite sequence of types indexed by the set $\{1, \ldots, n\}$, where $n \geq 0$. To save space in formulas $\tau_1, \tau_2, \ldots, \tau_n$ is normally shortened to $\tau_1..\tau_n$.

The sets of types and typed terms are defined by the following abstract grammar:

$$
\begin{array}{lcl}
\tau & ::= & \rho \quad | \quad \tau \rightarrow \tau' \quad | \quad \bigwedge[\tau_1..\tau_n] \quad | \quad \bigvee[\tau_1..\tau_n] \\
\mathtt{e} & ::= & \mathtt{x} \quad | \quad \lambda\mathtt{x}{:}\tau_1..\tau_n.\,\mathtt{e} \quad | \quad \mathtt{e}\ \mathtt{e}'
\end{array}
$$

Except for λ-abstractions, the syntax of expressions is just that of the simply typed λ-calculus. As in Forsythe, a λ-abstraction may be annotated with a finite *list* of alternative domain types. (See the discussion of the ARROW-I rule on page 4.)

A *context* is a finite sequence of pairs $\mathtt{x} : \tau$ of a variable and a type, with no variable mentioned twice. The metavariable Γ ranges over contexts. The set of variables in Γ is written $\mathrm{dom}(\Gamma)$.

$\mathrm{FV}(\mathtt{e})$ is the set of variables free in $\mathtt{e}$. $\mathrm{FTV}(\tau)$ is the set of type variables free in τ. $\mathrm{FTV}(\Gamma)$ is $\bigcup_{\mathtt{x}_i:\tau_i\in\Gamma} \mathrm{FTV}(\tau_i)$

the union of $\mathrm{FTV}(\tau_i)$, where for some $\mathtt{x}_i$ the pair $\mathtt{x}_i : \tau_i$ is in Γ.

Terms and types are identified up to renaming of bound variables. When $\mathtt{e}$ and $\mathtt{e}'$ are the same term modulo renaming of bound variables, we write $\mathtt{e} \equiv \mathtt{e}'$.

When a term $\mathtt{e}$ is substituted for a variable $\mathtt{x}$ in another term $\mathtt{e}'$, written $[\mathtt{e}/\mathtt{x}]\mathtt{e}'$, the bound variables of $\mathtt{e}'$ are first renamed to be different from the free variables of $\mathtt{e}$.

The following abbreviations for common intersection and union types are useful for making complicated expressions more readable:

$$
\begin{array}{lcl}
\mathtt{ns} & \stackrel{\mathrm{def}}{=} & \bigwedge[\,] \\
\sigma\wedge\tau & \stackrel{\mathrm{def}}{=} & \bigwedge[\sigma, \tau] \\
\mathtt{void} & \stackrel{\mathrm{def}}{=} & \bigvee[\,] \\
\sigma\vee\tau & \stackrel{\mathrm{def}}{=} & \bigvee[\sigma, \tau].
\end{array}
$$

The "nonsense" type `ns` is a maximal element in the type preorder; `void` is a minimal element. The expression "$\sigma\wedge\tau$" may be read as "σ intersect τ," "σ meet τ," or "σ and τ." The expression "$\sigma\vee\tau$" may be read as "σ union τ," "σ join τ," or "σ or τ."

1.2 The Subtype Relation

We begin by defining a preorder $\leq$ on the set of type expressions, where $\sigma \leq \tau$ asserts that "every value in type σ is also in type τ," or, operationally, that a term of type σ may safely be used in any context where a term of type τ is expected.

When $\sigma \leq \tau$ we say that σ is a *subtype* of τ and that τ is a supertype of σ. When σ and τ inhabit the same equivalence class in the subtype preorder ($\sigma \leq \tau$ and $\tau \leq \sigma$), we say that σ and τ are equivalent and write $\sigma{\sim}\tau$.

The first two rules state that $\leq$ is a preorder.

$$\tau \leq \tau \qquad \text{(SUB-REFL)}$$

$$\frac{\sigma \leq \theta \qquad \theta \leq \tau}{\sigma \leq \tau} \qquad \text{(SUB-TRANS)}$$

The subtyping rule for arrow types is covariant on the right hand side and contravariant on the left, as usual.

$$\frac{\tau_1 \leq \sigma_1 \qquad \sigma_2 \leq \tau_2}{\sigma_1{\rightarrow}\sigma_2 \ \leq\ \tau_1{\rightarrow}\tau_2} \qquad \text{(SUB-ARROW)}$$

$\bigwedge[\tau_1..\tau_n]$ is a greatest lower bound of $\tau_1..\tau_n$.

$$\frac{\text{for all } i,\ \sigma \leq \tau_i}{\sigma \ \leq \ \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\bigwedge[\tau_1..\tau_n] \ \leq \ \tau_i \qquad \text{(Sub-Inter-LB)}$$

(When the set that an index variable ranges over is clear from context, we usually omit it.)

$\bigvee[\tau_1..\tau_n]$ is a least upper bound of $\tau_1..\tau_n$.

$$\frac{\text{for all } i,\ \sigma_i \leq \tau}{\bigvee[\sigma_1..\sigma_n] \ \leq \ \tau} \qquad \text{(Sub-Union-L)}$$

$$\tau_i \ \leq \ \bigvee[\tau_1..\tau_n] \qquad \text{(Sub-Union-UB)}$$

Finally, arrows may be distributed over intersections on the right hand side and over unions of the left hand side. When an arrow is distributed over a union, the union changes to an intersection.

$$\bigwedge[\sigma{\rightarrow}\tau_1 \mathinner{..} \sigma{\rightarrow}\tau_n] \ \leq \ \sigma{\rightarrow}\bigwedge[\tau_1..\tau_n] \qquad \text{(Dist-AI-R)}$$

$$\bigvee[\sigma_1{\rightarrow}\tau \mathinner{..} \sigma_n{\rightarrow}\tau] \ \leq \ \bigwedge[\sigma_1..\sigma_n]{\rightarrow}\tau \qquad \text{(Dist-AU-L)}$$

1.3 Derived Rules

Some additional rules that can be derived from those above are:

$$\sigma{\rightarrow}\bigwedge[\tau_1..\tau_n] \ \leq \ \bigwedge[\sigma{\rightarrow}\tau_1 \mathinner{..} \sigma{\rightarrow}\tau_n] \qquad \text{(Dist-AI-R')}$$

$$\bigvee[\sigma_1..\sigma_n]{\rightarrow}\tau \ \leq \ \bigwedge[\sigma_1{\rightarrow}\tau \mathinner{..} \sigma_n{\rightarrow}\tau] \qquad \text{(Dist-AU-L')}$$

$$\bigvee[\sigma{\rightarrow}\tau_1 \mathinner{..} \sigma{\rightarrow}\tau_n] \ \leq \ \sigma{\rightarrow}\bigvee[\tau_1..\tau_n] \qquad \text{(Dist-AU-R)}$$

$$\bigvee[\sigma_1{\rightarrow}\tau \mathinner{..} \sigma_n{\rightarrow}\tau] \ \leq \ \bigwedge[\sigma_1..\sigma_n]{\rightarrow}\tau \qquad \text{(Dist-AU-L)}$$

$$\bigwedge[\sigma_1..\sigma_n] \vee \tau \ \leq \ \bigwedge[\sigma_1{\vee}\tau \mathinner{..} \sigma_n{\vee}\tau] \qquad \text{(Dist-UI)}$$

$$\bigvee[\sigma_1{\wedge}\tau \mathinner{..} \sigma_n{\wedge}\tau] \ \leq \ \bigvee[\sigma_1..\sigma_n] \wedge \tau \qquad \text{(Dist-IU)}$$

1.4 Non-Derivable Rules

The converses of rules Dist-AU-R through Dist-IU cannot be derived from the rules defining $\leq$. In Section 3.4 they are shown to be unsound in the current "intended model" of the calculus. However, there appear to be both pragmatic (algorithmic) and semantic reasons for adding them and adjusting the model construction accordingly.

$$\sigma{\rightarrow}\bigvee[\tau_1..\tau_n] \ \leq \ \bigvee[\sigma{\rightarrow}\tau_1 \mathinner{..} \sigma{\rightarrow}\tau_n] \qquad \text{(Bad-Dist-AU-R')}$$

$$\bigwedge[\sigma_1..\sigma_n]{\rightarrow}\tau \ \leq \ \bigvee[\sigma_1{\rightarrow}\tau \mathinner{..} \sigma_n{\rightarrow}\tau] \qquad \text{(Bad-Dist-AU-L')}$$

$$\bigwedge[\sigma_1{\vee}\tau \mathinner{..} \sigma_n{\vee}\tau] \ \leq \ \bigwedge[\sigma_1..\sigma_n] \vee \tau \qquad \text{(Bad-Dist-UI')}$$

$$\bigvee[\sigma_1..\sigma_n] \wedge \tau \ \leq \ \bigvee[\sigma_1{\wedge}\tau \mathinner{..} \sigma_n{\wedge}\tau] \qquad \text{(Bad-Dist-IU')}$$

1.5 Well-Typed Terms

We can now define the set of well typed $\lambda^{\vee}$ terms. Except for UNION-E, the rules given in this section are essentially the same as those defining the "core language" of Forsythe.

The SUBSUMPTION rule states that when a type has been derived for a term it may be promoted, as desired, to any supertype.

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} \in \tau} \tag{SUBSUMPTION}$$

The rule for typing variables is the usual one. A variable has at most one type in a given environment.

$$\Gamma_1, \mathtt{x} : \tau, \Gamma_2 \vdash \mathtt{x} \in \tau \tag{VAR}$$

As in Forsythe, λ-abstractions in $\lambda^{\vee}$ may be annotated with a finite sequence of alternative domain types. The arrow introduction rule allows a separate type to be derived for each domain type listed.

$$\frac{\Gamma, \mathtt{x} : \sigma_i \vdash \mathtt{e} \in \tau}{\Gamma, \vdash \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \in \sigma_i \to \tau} \tag{ARROW-I}$$

For example, if $\mathtt{plus} : (\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Real}\to\mathtt{Real}\to\mathtt{Real})$ is in Γ, then

$$\begin{array}{lll}
\Gamma \vdash \lambda\mathtt{x{:}Int.plus\ x\ x} & \in & \mathtt{Int}\to\mathtt{Int} \\
\Gamma \vdash \lambda\mathtt{x{:}Real.plus\ x\ x} & \in & \mathtt{Real}\to\mathtt{Real} \\
\Gamma \vdash \lambda\mathtt{x{:}Int{\wedge}Real.plus\ x\ x} & \in & \mathtt{Int}\to\mathtt{Int} \\
\Gamma \vdash \lambda\mathtt{x{:}Int{\vee}Real.plus\ x\ x} & \in & \mathtt{Real}\to\mathtt{Real} \\
\Gamma \vdash \lambda\mathtt{x{:}Int,Real.plus\ x\ x} & \in & \mathtt{Int}\to\mathtt{Int} \wedge \mathtt{Real}\to\mathtt{Real}
\end{array}$$

In the last case, ARROW-I is applied *twice* and the results conjoined using INTER-I (see below).

The arrow elimination rule is the usual one. To apply it, the rule of SUBSUMPTION may need to be applied to the types of the function `f` and/or the argument `e` so that the type of `e` precisely matches the domain type of `f`.

$$\frac{\Gamma \vdash \mathtt{f} \in \sigma\to\tau \qquad \Gamma \vdash \mathtt{e} \in \sigma}{\Gamma \vdash \mathtt{f\ e} \in \tau} \tag{ARROW-E}$$

If `f` has a type like $\mathtt{Int}\to\mathtt{Int} \wedge \mathtt{Bool}\to\mathtt{Char}$, ARROW-E may be applied more than once to give different types for (`f e`), which may then be conjoined using INTER-I.

The intersection introduction rule states that if all of the types $\tau_1..\tau_n$ have been derived for an expression `e`, then $\bigwedge[\tau_1..\tau_n]$ may also be derived.

$$\frac{\text{for all } i,\ \Gamma \vdash \mathtt{e} \in \tau_i}{\Gamma \vdash \mathtt{e} \in \bigwedge[\tau_1..\tau_n]} \tag{INTER-I}$$

No elimination rule is needed for intersection types; its effect is already provided by the SUBSUMPTION rule. For the same reason, no introduction rule is needed for union types.

The union elimination rule is the most novel element of the calculus. Operationally, it says that if an expression `e` has type $\bigvee[\tau_1..\tau_n]$ and appears in a context $\mathtt{e}'$ whose type would be τ if `e` could be given any one of the smaller types $\tau_1..\tau_n$, then $\mathtt{e}'$ may be taken to have type τ.

$$\frac{\Gamma \vdash \mathtt{e\ in} \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{e}' \in \tau} \tag{UNION-E}$$

For example, if $\mathtt{f} : \mathtt{A}{\rightarrow}\mathtt{C} \bigwedge \mathtt{B}{\rightarrow}\mathtt{C}$ and $\mathtt{e} : \mathtt{A}\vee\mathtt{B}$, then $\mathtt{f\ e}\ : \mathtt{C}$, since the value of $\mathtt{e}$ will certainly have either type $\mathtt{A}$ or typer $\mathtt{B}$, and in either case will be mapped by $\mathtt{f}$ to a result of type $\mathtt{C}$.

This rule presents severe difficulties for a typechecking procedure for $\lambda^{\vee}$. Because the syntactic forms of $\mathtt{e}$ and $\mathtt{e}'$ are unconstrained in the conclusion, Union-E can potentially be applied to *any* term $\mathtt{e}'$, choosing any subset of the occurrences of any subterm $\mathtt{e}$. Worse yet, every term $\mathtt{e}$ has an infinite number of union types; any $\mathtt{e}$ with a type strictly less than $\mathtt{ns}$ has an infinite number of *inequivalent* union types.

Section 4 proposes one method for typechecking the system with this form of the rule. Another possibility is to help the typechecker by introducing a syntactic marker indicating where the Union-E rule is to be applied. For example, assuming that $\mathtt{f} : \mathtt{Int}{\rightarrow}\mathtt{Bool} \wedge \mathtt{Char}{\rightarrow}\mathtt{Bool}$, we might write

$$\lambda\mathtt{x}{:}\mathtt{Int}\vee\mathtt{Char}.\,\mathtt{uelim}\ \mathtt{y} = \mathtt{x}\ \mathtt{in}\ \mathtt{f}\ \mathtt{y}$$

instead of

$$\lambda\mathtt{x}{:}\mathtt{Int}\vee\mathtt{Char}.\,\mathtt{f}\ \mathtt{x}.$$

Notice that the typechecker must still guess *which* union type to use for $\mathtt{x}$. Trying to annotate the $\mathtt{uelim}$ form with an explicit indication of this type may prevent some types from being derived for the body. For example, if $\mathtt{f} : \mathtt{Int}{\rightarrow}\mathtt{Bool} \wedge \mathtt{Char}{\rightarrow}\mathtt{Bool} \wedge \mathtt{Bool}{\rightarrow}\mathtt{Int}$, then

$$\lambda\mathtt{x}{:}\mathtt{Int}\vee\mathtt{Char}, \mathtt{Bool}.\,\mathtt{uelim}\ \mathtt{y} = \mathtt{x}\ \mathtt{in}\ \mathtt{f}\ \mathtt{y}\ \in\ (\mathtt{Int}\vee\mathtt{Char}){\rightarrow}\mathtt{Bool} \wedge \mathtt{Bool}{\rightarrow}\mathtt{Int}$$

in the original system, but

$$\lambda\mathtt{x}{:}\mathtt{Int}\vee\mathtt{Char}, \mathtt{Bool}.\,\mathtt{uelim}\ \mathtt{y} : \mathtt{Int}\vee\mathtt{Char} = \mathtt{x}\ \mathtt{in}\ \mathtt{f}\ \mathtt{y}$$

does not have the type $\mathtt{Bool}{\rightarrow}\mathtt{Int}$, since $\mathtt{Bool} \not\leq \mathtt{Int}\vee\mathtt{Char}$.

1.6 Record Types

As in Forsythe, the core language above can be extended to include simple single-field record types.

Let the metavariable ι range over a countable set of labels. The concrete syntax of $\lambda^{\vee}$ types and terms is extended as follows:

$$\begin{array}{lcl} \tau & ::= & \rho \quad|\quad \tau \rightarrow \tau' \quad|\quad \bigwedge[\tau_1..\tau_n] \quad|\quad \bigvee[\tau_1..\tau_n] \quad|\quad \iota : \tau \\ \mathtt{e} & ::= & \mathtt{x} \quad|\quad \lambda\mathtt{x}{:}\tau_1..\tau_n.\,\mathtt{e} \quad|\quad \mathtt{e}\ \mathtt{e}' \quad|\quad \mathtt{e}\ \mathtt{with}\ \iota = \mathtt{e}' \quad|\quad \mathtt{e}.\iota \end{array}$$

The subtyping rules for record types are:

$$\frac{\sigma \leq \tau}{\iota : \sigma\ \leq\ \iota : \tau} \qquad \text{(Sub-Record)}$$

$$\iota : \sigma \wedge \iota : \tau\ \leq\ \iota : (\sigma\wedge\tau) \qquad \text{(Dist-RI)}$$

$$\iota : (\sigma\vee\tau)\ \leq\ \iota : \sigma \vee \iota : \tau \qquad \text{(Dist-RU)}$$

The converses of Dist-RI and Dist-RU can be derived from Sub-Record, Sub-Inter, and Sub-Union.

The introduction and elimination rules for records are:

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma \qquad \Gamma \vdash \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{e}\ \mathtt{with}\ \iota = \mathtt{e}'\ \in\ (\sigma\backslash\iota) \wedge \iota : \tau} \qquad \text{(Record-I)}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \tau}{\Gamma \vdash \mathsf{e}.\iota \in \tau.\iota} \qquad (\textsc{Record-E})$$

where

$$\begin{array}{lcl}
\rho\backslash\iota & = & \rho \\
(\sigma{\rightarrow}\tau)\backslash\iota & = & \sigma{\rightarrow}\tau \\
(\bigwedge[\tau_1..\tau_n])\backslash\iota & = & \bigwedge[\tau_1\backslash\iota \mathinner{..} \tau_n\backslash\iota] \\
(\bigvee[\tau_1..\tau_n])\backslash\iota & = & \bigvee[\tau_1\backslash\iota \mathinner{..} \tau_n\backslash\iota] \\
(\iota : \tau)\backslash\iota & = & \mathtt{ns} \\
(\iota' : \tau)\backslash\iota & = & \iota' : \tau \quad \text{when } \iota' \neq \iota
\end{array}$$

and

$$\begin{array}{lcl}
\rho.\iota & = & \mathtt{ns} \\
(\sigma{\rightarrow}\tau).\iota & = & \mathtt{ns} \\
(\bigwedge[\tau_1..\tau_n]).\iota & = & \bigwedge[\tau_1.\iota \mathinner{..} \tau_n.\iota] \\
(\bigvee[\tau_1..\tau_n]).\iota & = & \bigvee[\tau_1.\iota \mathinner{..} \tau_n.\iota] \\
(\iota : \tau).\iota & = & \tau \\
(\iota' : \tau).\iota & = & \mathtt{ns} \quad \text{when } \iota' \neq \iota.
\end{array}$$

To reduce the number of cases that must be considered in proofs, record types are omitted in the formal sections of this report.

2 Examples

To make the foregoing definitions more concrete and demonstrate some of the expressive power of $\lambda^{\vee}$, we now develop several small programming examples. Aside from minor reformatting, the interactions shown are reproduced exactly from sessions with a prototype typechecker implemented in Standard ML.

N.b. The algorithms embodied in this prototype are not yet known to be either sound or complete.

2.1 Arithmetic

The first example shows how union types can be used to simulate a subtype ordering on the primitive types.

Section 1.2 defines $\leq$ as the least relation closed under the rules Sub-Refl through Dist-AU-L. Richer languages like Forsythe generally define the subtype relation as the least relation containing a given relation on the primitive types and closed under a similar set of rules. For example, in Forsythe, `Int` $\leq$ `Real`. In the present formulation of $\lambda^{\vee}$, it is never the case that $\rho \leq \rho'$ when $\rho \neq \rho'$. But it is possible to construct a non-primitive type that behaves like `Real`.

Assume that `Int` is a primitive type. Choose another primitive type `Frac` to denote the set of non-integral reals and define `Real` as an abbreviation for the union of `Int` and `Frac`. (Lines beginning with `>` represent expressions typed to a simple read-typecheck-print loop.)

```
> type Real = Int \/ Frac;
type Real = Frac \/ Int
```

Of course, this definition of `Real` is semantically unsatisfying because it ignores the fact that the injection from `Int` to `Real` at run time is almost certainly not an inclusion — converting 5 to

5.0 involves real computation, not just forgetting some type information. But it provides a good illustration of the interaction of union and intersection types.

Define a constant 5 of type `Int` and a constant `pi` of type `Real`:

```
> 5 : Int;
5 : Int

> pi : Real;
pi : Real
```

The built-in `plus` function takes two integers to an integer and two reals to a real. Before echoing the declared type, the typechecker normalizes it by distributing intersections and unions over arrows as much as possible, throwing away redundant conjuncts and disjuncts, and sorting lists of types appearing in meets and joins.

```
> plus: Int->Int->Int /\ Real->Real->Real;
plus :
     Frac->Frac->Real
  /\ Frac->Int->Real
  /\ Int->Frac->Real
  /\ Int->Int->Int
```

Applying `plus` to various combinations of integer and real arguments gives the expected results.

```
> plus 5 5;
it : Int

> plus pi pi;
it : Real

> plus 5 pi;
it : Real
```

Other functions built using `plus` are able to inherit its polymorphism, provided they declare enough domain types. (The backslash character is used to represent λ.)

```
> double_Int = \x:Int. plus x x;
double_Int : Int -> Int

> double_Real = \x:Real. plus x x;
double_Real : Frac->Real /\ Int->Real

> double = \x:Int,Real. plus x x;
double : Frac->Real /\ Int->Int
```

2.2 Booleans

`Bool` can be defined as the least upper bound of the type `T` of true booleans, whose only member is `true`, and the type `F` of false booleans, whose only member is `false`.

```
> type Bool = T \/ F;
type Bool = F \/ T

> true : T;
true : T

> false : F;
false : F
```

To examine boolean values, we introduce a constant `if` that selects its second or third argument depending on the truth value of its first argument. Since we are defining it as a constant rather than extending the base language, we need to explicitly declare which kinds of values we expect `if` to be used to select between — in this case, just `Bool` and `Int`.

```
> if : Bool -> Bool -> Bool -> Bool
>    /\ Bool -> Int  -> Int  -> Int;
if :
     F->F->F->Bool
  /\ F->F->T->Bool
  /\ F->Int->Int->Int
  /\ F->T->F->Bool
  /\ F->T->T->Bool
  /\ T->F->F->Bool
  /\ T->F->T->Bool
  /\ T->Int->Int->Int
  /\ T->T->F->Bool
  /\ T->T->T->Bool

> or = \x:T,F. \y:T,F. if x true y;
or : F->F->Bool /\ F->T->Bool /\ T->F->Bool /\ T->T->Bool

> if false pi 5;
it : NS
```

(The best type for the last example is `ns` because `pi` has neither type `Int` nor type `Bool`.)

Since we've distinguished the type of true booleans from the type of false booleans, we can actually give a better type for `if`.

```
> if : T -> Int -> NS  -> Int
>    /\ F -> NS  -> Int -> Int
>    /\ T -> T   -> NS  -> T
>    /\ T -> F   -> NS  -> F
>    /\ F -> NS  -> T   -> T
>    /\ F -> NS  -> F   -> F;
if :
     F->NS->T->T
  /\ F->NS->F->F
  /\ F->NS->Int->Int
  /\ T->F->NS->F
```

```
  /\ T->Int->NS->Int
  /\ T->T->NS->T
```

This type allows more refined types to be derived for expressions using `if`.

```
> or = \x:T,F. \y:T,F. if x true y;
or : F->F->F /\ F->T->T /\ T->F->T /\ T->T->T

> if false pi 5;
it : Int
```

We can also express a convenient type for a "parallel `if`" primitive [15]. Assume that `parif a b c` evaluates the selector `a` and the (boolean) arms `b` and `c` in parallel. If `a` evaluates to `true`, it aborts the evaluation of `c` and waits for the computation of `b` to finish; if `a` evaluates to `false`, it aborts the evaluation of `b` and waits for the computation of `c` to finish; if both `b` and `c` finish and their values agree, it terminates the computation of `a` and returns the common result immediately.

```
> parif :
>       T  -> T  -> NS -> T
>    /\ T  -> F  -> NS -> F
>    /\ F  -> NS -> T  -> T
>    /\ F  -> NS -> F  -> F
>    /\ NS -> T  -> T  -> T
>    /\ NS -> F  -> F  -> F;
parif :
     F->NS->T->T
  /\ F->NS->F->F
  /\ T->F->NS->F
  /\ T->T->NS->T
  /\ NS->T->T->T
  /\ NS->F->F->F
```

The `parif` primitive can be used to write a "parallel `or`" with the usual type:

```
> paror = \x:T,F,NS. \y:T,F,NS. parif x true y;
paror : F->F->F /\ T->NS->T /\ NS->T->T
```

2.3 List Processing

Union types can also be used to derive interesting types for list processing functions. [This example was suggested by Tim Freeman and Frank Pfenning.]

To keep the language simple, we introduce `List` as a primitive type rather than extending the type system to provide general type constructors, and assume that the `case` construct for lists always returns lists. Furthermore, to avoid introducing polymorphism we only provide lists of integers.

We distinguish between lists of even length and lists of odd length. `EvenCons` is the primitive type of nonempty, even-length lists; `Odd` is the type of odd-length lists. `Nil` is the type of the empty list. From these, we build the type of all even-length lists, the type of nonempty lists, and the type of all lists.

```
> type Even = EvenCons \/ Nil;
type Even = EvenCons \/ Nil

> type Cons = EvenCons \/ Odd;
type Cons = EvenCons \/ Odd

> type List = Cons \/ Nil;
type List = EvenCons \/ Nil \/ Odd
```

The basic operations on lists are given types describing their actions on even and odd length lists.

```
> nil  : Nil;
nil : Nil

> cons : Int->Even->Odd /\ Int->Odd->EvenCons;
cons : Int->EvenCons->Odd /\ Int->Nil->Odd /\ Int->Odd->EvenCons

> car  : List -> Int;
car : EvenCons->Int /\ Nil->Int /\ Odd->Int

> cdr  : Even->Odd /\ Odd->Even;
cdr : EvenCons->Odd /\ Nil->Odd /\ Odd->Even
```

More accurate types can be given for `car` and `cdr`, for example `car` : `Cons`→`Int` and `cdr` : `EvenCons`→`Odd` ∧ `Odd`→`Even`, but this would prevent the desired type from being derived for `append` below. This is a reminder that we're still only in the business of typechecking, not full abstract interpretation.

These "refined" types are propagated through expressions by the typechecker, allowing types involving `Even` and `Odd` to be derived for user-defined functions too:

```
> cddr = \l:Even,Odd. cdr (cdr l);
cddr : EvenCons->Even /\ Nil->Even /\ Odd->Odd
```

The `case` construct takes a list `l` and two "arms" and chooses between the arms depending on the form of `l`. If `l` is `nil`, (`case l` e_1 e_2) returns e_1, which should be of type `List`. If `l` is (`cons n l'`) for some integer `n` and list `l'`, then `case l` $e_1 e_2$ returns the result of applying e_2 to `n` and `l'`. So one type of `case` is `List` → `List` → (`Int`→`List`→`List`) → `List`. The more refined type given here allows more information to be inferred about the result type when more is known about the types of the selector and the arms.

```
> case :  Even -> Even -> (Int -> Odd  -> Even) -> (Even\/Even)
>      /\ Even -> Even -> (Int -> Odd  -> Odd ) -> (Even\/Odd)
>      /\ Even ->  Odd -> (Int -> Odd  -> Even) -> (Odd\/Even)
>      /\ Even ->  Odd -> (Int -> Odd  -> Odd ) -> (Odd\/Odd)
>      /\ Odd  -> List -> (Int -> Even -> Even) -> (Even)
>      /\ Odd  -> List -> (Int -> Even -> Odd ) -> (Odd);
case :
     EvenCons->EvenCons->(Int->Odd->Odd)->List
  /\ EvenCons->EvenCons->(Int->Odd->Even)->Even
```

```
  /\ EvenCons->Nil->(Int->Odd->Odd)->List
  /\ EvenCons->Nil->(Int->Odd->Even)->Even
  /\ EvenCons->Odd->(Int->Odd->Odd)->Odd
  /\ EvenCons->Odd->(Int->Odd->Even)->List
  /\ Nil->EvenCons->(Int->Odd->Odd)->List
  /\ Nil->EvenCons->(Int->Odd->Even)->Even
  /\ Nil->Nil->(Int->Odd->Odd)->List
  /\ Nil->Nil->(Int->Odd->Even)->Even
  /\ Nil->Odd->(Int->Odd->Odd)->Odd
  /\ Nil->Odd->(Int->Odd->Even)->List
  /\ Odd->EvenCons->(Int->EvenCons->Odd/\Int->Nil->Odd)->Odd
  /\ Odd->EvenCons->(Int->EvenCons->Even/\Int->Nil->Even)->Even
  /\ Odd->Nil->(Int->EvenCons->Odd/\Int->Nil->Odd)->Odd
  /\ Odd->Nil->(Int->EvenCons->Even/\Int->Nil->Even)->Even
  /\ Odd->Odd->(Int->EvenCons->Odd/\Int->Nil->Odd)->Odd
  /\ Odd->Odd->(Int->EvenCons->Even/\Int->Nil->Even)->Even
```

This type for `case` can be used to derive a type for the `append` function that encodes some of the information that might be discovered by an abstract interpreter. The desired type of `append` is:

```
> type AppType =    Even->Even->Even
>                /\ Odd ->Even->Odd
>                /\ Even->Odd ->Odd
>                /\ Odd ->Odd ->Even;
type AppType =
     EvenCons->EvenCons->(EvenCons\/Nil)
  /\ EvenCons->Nil->(EvenCons\/Nil)
  /\ EvenCons->Odd->Odd
  /\ Nil->EvenCons->(EvenCons\/Nil)
  /\ Nil->Nil->(EvenCons\/Nil)
  /\ Nil->Odd->Odd
  /\ Odd->EvenCons->Odd
  /\ Odd->Nil->Odd
  /\ Odd->Odd->(EvenCons\/Nil)
```

Introducing a primitive `fix` construct would not be problematic as far as typechecking is concerned, but for present purposes it suffices to introduce a new constant `append_rec` to take the place of the recursive call in the definition of `append`:

```
> append_rec : AppType;
append_rec : AppType
```

The `append` function itself can now be defined as follows:

```
> append = \l1:Even,Odd.
>            \l2:Even,Odd.
>                 case l1
>                      l2
>                      (\n:Int. \l1':Even,Odd.
```

```
>                              cons n (append_rec l1' l2));
append : AppType
```

Finally, we can check that append also has type List→List→List by explicitly annotating its definition with this type. The typechecker generates a warning because the declared type is strictly greater than the one it has inferred so information is being thrown away.

```
> boring_append =
>         \l1:Even,Odd.
>              \l2:Even,Odd.
>                   case l1
>                       l2
>                       (\n:Int. \l1':Even,Odd.
>                           cons n (append_rec l1' l2))
>        : List -> List -> List;
boring_append : List -> List->List
Warning: actual type was AppType
```

2.4 Record operations

As in Forsythe, multi-field record types can be built in $\lambda^{\vee}$ from a single-field record type constructor. (To simplify parsing in the prototype typechecker, record types are written $\tau@\iota$ instead of $\iota : \tau$.)

```
> type R1 = Int@i1;
type R1 = Int@i1

> type R2 = Bool@i2;
type R2 = F@i2 \/ T@i2

> type R12 = R1 /\ R2;
type R12 = (F@i2\/T@i2) /\ Int@i1

> type R = (Int/\Bool)@i;
type R = (F@i\/T@i) /\ Int@i
```

The type of a record with no fields is a conjunction of zero single-field record types — that is, ns.

```
> empty : NS;
empty : NS
```

Record values can be built up by extending the empty record a field at a time.

```
> r1 = empty with j1=true;
r1 : T@j1

> r2 = r1 with j2=5;
r2 : T@j1 /\ Int@j2
```

To simplify parsing, field selection is written e$@\iota$ instead of e.ι.

```
> r2@j1;
it : T
```

Since the left component of the `with` construct may be a term of any type whatsoever, it is possible to construct record types with one conjunct that does not have the form $\tau @ \iota$.

```
> (\x:Int.x) with i1=false;
it : Int->Int /\ F@i1
```

Record types can be built up with unions as well as intersections.

```
> type R1 = T1@a1 /\ T2@a2;
type R1 = T1@a1 /\ T2@a2

> type R2 = S2@a2 /\ S3@a3;
type R2 = S2@a2 /\ S3@a3

> x : R1/\R2;
x : T1@a1 /\ S2@a2 /\ T2@a2 /\ S3@a3

> x@a2;
it : S2 /\ T2

> y : R1\/R2;
y : R1 \/ R2

> y@a2;
it : S2 \/ T2
```

2.5 Disjoint Unions

Single-field records and union types can be used to construct disjoint union types like those found in many other programming languages.

Let $\mathtt{S}_1$ and $\mathtt{S}_2$ be two given types, with constants $\mathtt{c}_1$ and $\mathtt{c}_2$ of types $\mathtt{S}_1$ and $\mathtt{S}_2$.

```
> c1 : S1;
c1 : S1

> c2 : S2;
c2 : S2
```

The disjoint union of $\mathtt{S}_1$ and $\mathtt{S}_2$ can be thought of as the ordinary union of the sets

$$\{(\mathtt{tag} = \mathtt{true}, \mathtt{data} = \mathtt{s}_1) \mid \mathtt{s}_1 \in \mathtt{S}_1\}$$

and

$$\{(\mathtt{tag} = \mathtt{false}, \mathtt{data} = \mathtt{s}_2) \mid \mathtt{s}_2 \in \mathtt{S}_2\}.$$

In $\lambda^{\vee}$ this can be expressed as follows:

```
> type S1orS2 =  (T@tag /\ S1@data)
>              \/ (F@tag /\ S2@data);
type S1orS2 = (S2@data/\F@tag) \/ (S1@data/\T@tag)
```

The injection functions $\mathtt{in}_1$ and $\mathtt{in}_2$ take elements of $\mathtt{S}_1$ and $\mathtt{S}_2$ to element of the disjoint union by tagging them appropriately.

```
> in1 = \x:S1. (empty with tag=true) with data=x;
in1 : S1->S1@data /\ S1->T@tag

> in2 = \x:S2. (empty with tag=false) with data=x;
in2 : S2->S2@data /\ S2->F@tag

> v1 = (in1 c1) : S1orS2;
v1 : S1orS2
Warning: actual type was S1@data /\ T@tag

> v2 = (in2 c2) : S1orS2;
v2 : S1orS2
Warning: actual type was S2@data /\ F@tag
```

Selector functions can now be written to examine the `tag` component of values of the disjoint union type and process the `data` component using a function appropriate for its type.

```
> select_Int = \f1:S1->Int.
>                  \f2:S2->Int.
>                      \x:S1orS2. if (x@tag) (f1 (x@data)) (f2 (x@data));
select_Int :
     (S1->Int)->(S2->Int)->(S1@data/\T@tag)->Int
  /\ (S1->Int)->(S2->Int)->(S2@data/\F@tag)->Int

> f1 : S1 -> Int;
f1 : S1 -> Int

> f2 : S2 -> Int;
f2 : S2 -> Int

> select_Int f1 f2 v1;
it : Int
```

This construction also sheds some light on the issue of adding axioms corresponding to deMorgan's laws for distributing unions and intersections to $\lambda^{\vee}$. In the presence of these axioms, the disjoint union type `S1orS2` is equivalent to:

```
> type S1orS2dm =   Bool@tag
>                /\ (S1\/S2)@data
>                /\ (T@tag \/ S2@data)
>                /\ (F@tag \/ S1@data);
type S1orS2dm =
     (S1@data\/S2@data)
  /\ (S1@data\/F@tag)
  /\ (S2@data\/T@tag)
  /\ (F@tag\/T@tag)
```

Somewhat surprisingly, this type also contains sufficient information to derive the appropriate type for the `select_Int` function:

```
> select_Int = \x:S1orS2dm. if (x@tag) (f1 (x@data)) (f2 (x@data));
select_Int :
     ((S1@data\/F@tag)/\(S1@data\/S2@data)/\(F@tag\/T@tag)/\(S2@data\/T@tag))
  -> Int
```

Operationally, the body of the abstraction must be typechecked 16 times — once for each possible combination of choices from the four conjuncts in the type of `x`, these being all the "interesting" types above `S1orS2dm`:

```
T@tag  /\ S1@data /\  T@tag  /\  F@tag
T@tag  /\ S1@data /\  T@tag  /\ S1@data
T@tag  /\ S1@data /\ S2@data /\  F@tag
T@tag  /\ S1@data /\ S2@data /\ S1@data
T@tag  /\ S2@data /\  T@tag  /\  F@tag
T@tag  /\ S2@data /\  T@tag  /\ S1@data
T@tag  /\ S2@data /\ S2@data /\  F@tag
T@tag  /\ S2@data /\ S2@data /\ S1@data
F@tag  /\ S1@data /\  T@tag  /\  F@tag
F@tag  /\ S1@data /\  T@tag  /\ S1@data
F@tag  /\ S1@data /\ S2@data /\  F@tag
F@tag  /\ S1@data /\ S2@data /\ S1@data
F@tag  /\ S2@data /\  T@tag  /\  F@tag
F@tag  /\ S2@data /\  T@tag  /\ S1@data
F@tag  /\ S2@data /\ S2@data /\  F@tag
F@tag  /\ S2@data /\ S2@data /\ S1@data
```

The type of the body in each case is `Int`, so by UNION-E the body has type $\bigwedge[\texttt{Nat}\,..\,\texttt{Nat}] \sim \texttt{Nat}$.

2.6 The Expressive Power of Union Types

In all of the previous examples, union types were used to specify the relations between refined types like `T` and `EvenCons` and less refined ones like `Bool` and `List`. If we were willing to give up inclusions like `AppType` $\leq$ `List`$\rightarrow$`List`$\rightarrow$`List`, we could ignore these relationships and deal only in terms of the more refined types, which can be expressed in terms of just intersection and arrow. Perhaps union types are actually superfluous.

This example shows that some real expressive power is gained when union types are allowed.

Define `Number` to be the union of `Neg`, `Pos`, and `Zero`. (As before, this is an "inessential" use of union types.)

```
> type Number = Neg /\ Zero /\ Pos;
type Number = Neg /\ Pos /\ Zero

> one : Pos;
one : Pos

> minus_one : Neg;
minus_one : Neg
```

```
> is_zero : Neg->F /\ Zero->T /\ Pos->F;
is_zero : Neg->F /\ Pos->F /\ Zero->T
```

Let `b` be some boolean constant.

```
> b : Bool;
b : Bool
```

Redefine `if` to operate on the subtypes of `Number`:

```
> if :
>       Bool -> Neg -> Neg -> (Neg\/Neg)
>    /\ Bool -> Neg -> Zero -> (Neg\/Zero)
>    /\ Bool -> Neg -> Pos -> (Neg\/Pos)
>    /\ Bool -> Zero -> Neg -> (Zero\/Neg)
>    /\ Bool -> Zero -> Zero -> (Zero\/Zero)
>    /\ Bool -> Zero -> Pos -> (Zero\/Pos)
>    /\ Bool -> Pos -> Neg -> (Pos\/Neg)
>    /\ Bool -> Pos -> Zero -> (Pos\/Zero)
>    /\ Bool -> Pos -> Pos -> (Pos\/Pos);
if :
     F->Neg->Neg->Neg
  /\ F->Neg->Pos->(Neg\/Pos)
  /\ F->Neg->Zero->(Neg\/Zero)
  /\ F->Pos->Neg->(Neg\/Pos)
  /\ F->Pos->Pos->Pos
  /\ F->Pos->Zero->(Pos\/Zero)
  /\ F->Zero->Neg->(Neg\/Zero)
  /\ F->Zero->Pos->(Pos\/Zero)
  /\ F->Zero->Zero->Zero
  /\ T->Neg->Neg->Neg
  /\ T->Neg->Pos->(Neg\/Pos)
  /\ T->Neg->Zero->(Neg\/Zero)
  /\ T->Pos->Neg->(Neg\/Pos)
  /\ T->Pos->Pos->Pos
  /\ T->Pos->Zero->(Pos\/Zero)
  /\ T->Zero->Neg->(Neg\/Zero)
  /\ T->Zero->Pos->(Pos\/Zero)
  /\ T->Zero->Zero->Zero
```

Depending on `b`, let `n` be either `one` or `minus_one`.

```
> n = if b one minus_one;
n : Neg \/ Pos
```

Now try to tell if `n` is zero.

```
> z = is_zero n;
z : F
```

If *new* union types could not be formed during typechecking, the best type for `z` would be `Bool`, not `F`.

3 Semantics

This section specifies when two $\lambda^{\vee}$ terms are considered equal and exhibits a PER model validating the typing and equality rules.

N.b. The semantics presented here serves to demonstrate that the language defined above is "sensible." But the construction does not extend — at least not straightforwardly — to allow some desirable extensions of the language, in particular:

- deMorgan's laws for distributing intersections over unions and vice versa;
- coercions between primitive types.

This account of the semantics of $\lambda^{\vee}$ should therefore be regarded with some skepticism. Work on a more comprehensive account is underway.

3.1 Equality

Definition 3.1: Two terms $\mathtt{e}$ and $\mathtt{e}'$ are *equal at type* τ *in context* Γ, written $\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau$, iff this judgement can be derived from the following rules:

$$\frac{\Gamma \vdash \mathtt{e} \in \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e} \in \tau} \qquad (\textsc{Eq-Refl})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{e}' = \mathtt{e} \in \tau} \qquad (\textsc{Eq-Symm})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau \qquad \Gamma \vdash \mathtt{e}' = \mathtt{e}'' \in \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e}'' \in \tau} \qquad (\textsc{Eq-Trans})$$

$$\frac{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) \in \tau}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) = [\mathtt{e}/\mathtt{x}]\mathtt{f} \in \tau} \qquad (\textsc{Eq-Beta})$$

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma_i \rightarrow \tau \qquad \mathtt{x} \notin FV(\mathtt{e})}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}\ \mathtt{x}) = \mathtt{e} \in \sigma_i \rightarrow \tau} \qquad (\textsc{Eq-Eta})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau} \qquad (\textsc{Eq-Subsumption})$$

$$\frac{\Gamma, \mathtt{x} : \sigma_i \vdash \mathtt{e} = \mathtt{e}' \in \tau}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}) = (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}') \in \sigma_i \rightarrow \tau} \qquad (\textsc{Eq-Arrow-I})$$

$$\frac{\Gamma \vdash \mathtt{f} = \mathtt{f}' \in \sigma \rightarrow \tau \qquad \Gamma \vdash \mathtt{e} = \mathtt{e}' \in \sigma}{\Gamma \vdash (\mathtt{f}\ \mathtt{e}) = (\mathtt{f}'\ \mathtt{e}') \in \tau} \qquad (\textsc{Eq-Arrow-E})$$

$$\frac{\text{for all } i,\ \Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau_i}{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \bigwedge[\tau_1..\tau_n]} \qquad (\textsc{Eq-Inter-I})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{f} = \mathtt{f}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{f} = [\mathtt{e}'/\mathtt{x}]\mathtt{f}' \in \tau} \qquad (\textsc{Eq-Union-E})$$

Comments:

- Terms get monotonically "more equal" as their types are promoted. In the limiting case, any two parseable terms are equal in `ns`.
- Equality at a given type is an equivalence relation.
- The rule Eq-Refl subsumes the role of Eq-Var.
- The rules Eq-Refl, Eq-Arrow-I, and Eq-Arrow-E together state that equality is a congruence on typed terms.

Lemma 3.2: If $\Gamma \vdash \mathsf{e} = \mathsf{e}' \in \tau$, then $\Gamma \vdash \mathsf{e} \in \tau$ and $\Gamma \vdash \mathsf{e}' \in \tau$.
Proof: By straightforward induction on the definition. □

3.2 Partial Equivalence Relations

Most of the basic definitions in this section come from Bruce and Longo [2]. Some of the notation is inspired by recent papers of Freyd, Scott, and others (e.g. [8]).

Definition 3.3: Recall that Kleene's applicative structure $(\Omega, \cdot)$ consists of the set Ω of natural numbers equipped with an application operation $\cdot$, where $n \cdot m$ is the result of applying the nth partial recursive function to m.

A *partial equivalence relation* (PER) is a symmetric and transitive relation A on Ω. We write $m\ \{A\}\ n$ when A relates m and n. The domain of A, written $\mathrm{dom}(A)$, is $\{n \mid n\ \{A\}\ n\}$. When n is in $\mathrm{dom}(A)$, the equivalence class of n with respect to A, written $[n : A]$, is $\{m \mid m\ \{A\}\ n\}$. The set of equivalence classes of Ω with respect to A, written $[A]$, is $\{[n : A] \mid n \in \mathrm{dom}(A)\}$.

When a PER A is viewed as the interpretation of a type τ, the elements of $[A]$ correspond to the interpretations of terms of type τ.

Definition 3.4: Let A and B be PERs. $A \to B$ is the PER defined by:

$$m\ \{A \to B\}\ n \quad \text{iff} \quad \text{for all } p \text{ and } q,\ p\ \{A\}\ q \text{ implies } m \cdot p\ \{B\}\ n \cdot q.$$

Lemma 3.5: $A \to B$ is a PER when A and B are PERs.
Proof: (Symmetry) Let $m\ \{A \to B\}\ n$. Then

$$\text{for all } p \text{ and } q, \quad p\ \{A\}\ q \text{ implies } m \cdot p\ \{B\}\ n \cdot q,$$

which by the symmetry of A and B implies that

$$\text{for all } p \text{ and } q, \quad q\ \{A\}\ p \text{ implies } n \cdot q\ \{B\}\ m \cdot p,$$

that is, $n\ \{A \to B\}\ m$.

(Transitivity) Let $m\ \{A \to B\}\ n$ and $n\ \{A \to B\}\ o$. Then

$$\begin{array}{lll}
 & \text{for all } p, q, \text{ and } r, & (p\ \{A\}\ q \text{ implies } m \cdot p\ \{B\}\ n \cdot q) \text{ and } (q\ \{A\}\ r \text{ implies } n \cdot q\ \{B\}\ o \cdot r) \\
\Rightarrow & \text{for all } p, q, \text{ and } r, & (p\ \{A\}\ q \text{ and } q\ \{A\}\ r) \text{ implies } (m \cdot p\ \{B\}\ n \cdot q \text{ and } n \cdot q\ \{B\}\ o \cdot r) \\
\Rightarrow & \text{for all } p \text{ and } r, & (p\ \{A\}\ p \text{ and } p\ \{A\}\ r) \text{ implies } m \cdot p\ \{B\}\ o \cdot r \\
\Rightarrow & \text{for all } p \text{ and } r, & p\ \{A\}\ r \text{ implies } m \cdot p\ \{B\}\ o \cdot r,
\end{array}$$

that is, $m\ \{A \to B\}\ o$. □

Definition 3.6: A function $f : [A] \rightarrow [B]$ is said to be *computable* iff there is some n in Ω such that

$$\text{for all } p \in \text{dom}(A),\ f([p:A]) = [n \cdot p : B].$$

The category PER has PERs as objects and computable functions as morphisms.

Definition 3.7: Let A and B be PERs. We say that A *is contained in* B or A *is finer than* B, written $A \subseteq B$, if the graph of A is a subset of the graph of B, that is, if $p\ \{A\}\ q$ implies $p\ \{B\}\ q$.

Definition 3.8: Let $A_1..A_n$ be PERs. The *intersection* of $A_1..A_n$, written $\sqcap[A_1..A_n]$, is the relation defined as follows:

$$p\ \{\sqcap[A_1..A_n]\}\ q \quad \text{iff} \quad \text{for all } i,\ p\ \{A_i\}\ q.$$

Lemma 3.9: $\sqcap[A_1..A_n]$ is a PER when $A_1..A_n$ are PERs.

Proof: (Symmetry) Let $p\ \{\sqcap[A_1..A_n]\}\ q$. Then for each i, $p\ \{A_i\}\ q$. By the symmetry of A_i, $q\ \{A_i\}\ p$. So $q\ \{\sqcap[A_1..A_n]\}\ p$.

(Transitivity) Let $p\ \{\sqcap[A_1..A_n]\}\ q$ and $q\ \{\sqcap[A_1..A_n]\}\ r$. Then for each i, $p\ \{A_i\}\ q$ and $q\ \{A_i\}\ r$. By the transitivity of A_i, $p\ \{A_i\}\ r$. So $p\ \{\sqcap[A_1..A_n]\}\ r$. □

Definition 3.10: Let A and B be relations. The composite relation $A \circ B$ is defined by:

$$p\ \{A \circ B\}\ r \quad \text{iff} \quad \text{for some } q,\ p\ \{A\}\ q \text{ and } q\ \{B\}\ r.$$

Abbreviate $A^1 = A$, $A^2 = A \circ A$, etc.

Definition 3.11: Let $A_1..A_n$ be PERs. Let

$$A \ = \bigcup_{1 \le i \le n} A_i.$$

The *union* of $A_1..A_n$, written $\sqcup[A_1..A_n]$ is the relation defined as follows:

$$\sqcup[A_1..A_n] = \bigcup_{1 \le j \le \infty} A^j.$$

Equivalently, $\sqcup[A_1..A_n]$ is the finest transitive relation containing all of the A_is.

Lemma 3.12: Whenever $p\ \{\sqcup[A_1..A_n]\}\ q$ there is some j such that $p\ \{A^j\}\ q$ — that is, there is some sequence of elements $o_1..o_{j+1} \in \Omega$ such that

$$p = o_1\ \{A_1\}\ o_2\ \{A_2\}\ \cdots\ \{A_j\}\ o_{j+1} = q,$$

where each $A_i \in \{A_1..A_n\}$.

Proof: Straightforward, from the definition. □

Lemma 3.13: $\sqcup[A_1..A_n]$ is a PER when $A_1..A_n$ are PERs.

Proof: (Transitivity) Immediate.

(Symmetry) Let $p \; \{\bigsqcup[A_1..A_n]\} \; q$. Use Lemma 3.12 to choose a sequence $o_1..o_{j+1}$

$$p = o_1 \; \{A_1\} \; o_2 \; \{A_2\} \cdots \{A_j\} \; o_{j+1} = q.$$

Since all of the A_i are symmetric, we may reverse this sequence, giving

$$q = o_{j+1} \; \{A_j\} \cdots \{A_2\} \; o_2 \; \{A_1\} \; o_1 = p.$$

Hence by Definition 3.11, $q \; \{\bigsqcup[A_1..A_n]\} \; p$. □

Observe that $\bigsqcap[\,]$ relates every element of Ω to every other, and that $\bigsqcup[\,]$ is empty.

The PER model of $\lambda^{\vee}$ given in the next section is based on an underlying untyped model, defined as follows.

Definition 3.14: An *environment* $E : \mathrm{V{\scriptsize AR}} \rightarrow \Omega$ is a total mapping from the set of variables to elements of Ω.

Definition 3.15: For each pure λ-term $\mathtt{e}$ and environment E, the *interpretation* of $\mathtt{e}$ with respect to E, written $\langle\!\langle \mathtt{e} \rangle\!\rangle E$, is defined as follows:

$$\begin{array}{lcl}
\langle\!\langle \mathtt{x} \rangle\!\rangle E & = & E(\mathtt{x}) \\
\langle\!\langle \lambda \mathtt{x}.\, \mathtt{e} \rangle\!\rangle E & = & \mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow m]) \\
\langle\!\langle \mathtt{e}\ \mathtt{e}' \rangle\!\rangle E & = & \langle\!\langle \mathtt{e} \rangle\!\rangle E \cdot \langle\!\langle \mathtt{e}' \rangle\!\rangle E.
\end{array}$$

where, when f is computable,

$$\mathrm{uabs}(f) = \text{the least } n \in \Omega \text{ such that } n \cdot m = f(m) \text{ for all } m.$$

[*To be rigorous, the S-N-M theorem should be used here to show that this definition is proper.*]

Definition 3.16: The definition of $\langle\!\langle \mathtt{e} \rangle\!\rangle$ for untyped lambda-terms is extended to $\lambda^{\vee}$ terms by ignoring type annotations on bound variables:

$$\begin{array}{lcl}
\langle\!\langle \mathtt{x} \rangle\!\rangle E & = & E(\mathtt{x}) \\
\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e} \rangle\!\rangle E & = & \mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow m]) \\
\langle\!\langle \mathtt{e}\ \mathtt{e}' \rangle\!\rangle E & = & \langle\!\langle \mathtt{e} \rangle\!\rangle E \cdot \langle\!\langle \mathtt{e}' \rangle\!\rangle E.
\end{array}$$

Lemma 3.17: If $\mathtt{x} \notin \mathrm{FV}(\mathtt{e})$ and $E(\mathtt{y}) = E'(\mathtt{y})$ whenever $\mathtt{y} \neq \mathtt{x}$, then

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E = \langle\!\langle \mathtt{e} \rangle\!\rangle E'.$$

Proof: By straightforward induction on the definition of $\langle\!\langle - \rangle\!\rangle$. □

Lemma 3.18: If $\langle\!\langle \mathtt{e} \rangle\!\rangle E = n$, then

$$\langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow n] = \langle\!\langle [\mathtt{e}/\mathtt{x}]\mathtt{e}' \rangle\!\rangle E.$$

Proof: By induction on the structure of $\mathtt{e}'$.

Case $\mathtt{e}' \equiv \mathtt{y}$ with $\mathtt{y} \neq \mathtt{x}$ or $\mathtt{e}' \equiv \lambda\mathtt{x}.\,\mathtt{e}''$:
Immediate, since $[\mathtt{e}/\mathtt{x}]\mathtt{e}' \equiv \mathtt{e}'$.

Case $\mathtt{e}' \equiv \mathtt{x}$:
Since $[\mathtt{e}/\mathtt{x}]\mathtt{e}' = \mathtt{e}$, we have $\langle\!\langle[\mathtt{e}/\mathtt{x}]\mathtt{e}'\rangle\!\rangle E = n = \langle\!\langle\mathtt{e}'\rangle\!\rangle E[\mathtt{x} \leftarrow n]$.

Case $\mathtt{e}' \equiv \lambda\mathtt{y}.\,\mathtt{e}''$ with $\mathtt{y} \neq \mathtt{x}$:
Rename $\mathtt{y}$, if necessary, so that it is not among the free variables of $\mathtt{e}$. Then by Lemma 3.17, $\langle\!\langle\mathtt{e}\rangle\!\rangle\mathtt{e}[\mathtt{y} \leftarrow p] = n$ for any p. By the induction hypothesis, $\langle\!\langle\mathtt{e}''\rangle\!\rangle E[\mathtt{y} \leftarrow p][\mathtt{x} \leftarrow n] = \langle\!\langle[\mathtt{e}/\mathtt{x}]\mathtt{e}''\rangle\!\rangle E[\mathtt{y} \leftarrow p]$. By the definition of $\langle\!\langle - \rangle\!\rangle$,

$$
\begin{aligned}
\langle\!\langle\mathtt{e}'\rangle\!\rangle E[\mathtt{x} \leftarrow n] &= \mathrm{uabs}(\lambda p.\, \langle\!\langle\mathtt{e}''\rangle\!\rangle E[\mathtt{x} \leftarrow n][\mathtt{y} \leftarrow p] \\
&= \mathrm{uabs}(\lambda p.\, \langle\!\langle\mathtt{e}''\rangle\!\rangle E[\mathtt{y} \leftarrow p][\mathtt{x} \leftarrow n]) \\
&= \mathrm{uabs}(\lambda p.\, \langle\!\langle[\mathtt{e}/\mathtt{x}]\mathtt{e}''\rangle\!\rangle E[\mathtt{y} \leftarrow p]) \\
&= \langle\!\langle\lambda\mathtt{y}.\,[\mathtt{e}/\mathtt{x}]\mathtt{e}''\rangle\!\rangle E \\
&= \langle\!\langle[\mathtt{e}/\mathtt{x}](\lambda\mathtt{y}.\,\mathtt{e}'')\rangle\!\rangle E.
\end{aligned}
$$

Case $\mathtt{e}' \equiv (\mathtt{e}_1\ \mathtt{e}_2)$:
Reason as follows:

$$
\begin{aligned}
\langle\!\langle\mathtt{e}_1\ \mathtt{e}_2\rangle\!\rangle E[\mathtt{x} \leftarrow n] &= \langle\!\langle\mathtt{e}_1\rangle\!\rangle E[\mathtt{x} \leftarrow n] \cdot \langle\!\langle\mathtt{e}_2\rangle\!\rangle E[\mathtt{x} \leftarrow n] \\
&= \langle\!\langle[\mathtt{e}/\mathtt{x}]\mathtt{e}_1\rangle\!\rangle E \cdot \langle\!\langle[\mathtt{e}/\mathtt{x}]\mathtt{e}_2\rangle\!\rangle E \text{ (by the IH)} \\
&= \langle\!\langle[\mathtt{e}/\mathtt{x}](\mathtt{e}_1\ \mathtt{e}_2)\rangle\!\rangle E.
\end{aligned}
$$

□

3.3 A PER Model of $\lambda^{\vee}$

Definition 3.19: For each primitive type ρ_i, fix a PER P_i. The interpretation of type expressions is then given as follows:

$$
\begin{aligned}
[\![\rho_i]\!] &= P_i \\
[\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!] \\
[\![\bigwedge[\tau_1..\tau_n]]\!] &= \bigsqcap[[\![\tau_1]\!]\,..\,[\![\tau_n]\!]] \\
[\![\bigvee[\tau_1..\tau_n]]\!] &= \bigsqcup[[\![\tau_1]\!]\,..\,[\![\tau_n]\!]]
\end{aligned}
$$

Proposition 3.20: For all types ϕ and ψ, $\phi \leq \psi$ implies $[\![\phi]\!] \subseteq [\![\psi]\!]$.
Proof: By induction on the structure of a derivation of $\phi \leq \psi$.

Case SUB-REFL: $\tau \leq \tau$
$[\![\tau]\!] \subseteq [\![\tau]\!]$ by the reflexivity of containment of PERS.

Case SUB-TRANS: $\dfrac{\sigma \leq \theta \qquad \theta \leq \tau}{\sigma \leq \tau}$

By the induction hypothesis, $[\![\sigma]\!] \subseteq [\![\theta]\!]$ and $[\![\theta]\!] \subseteq [\![\tau]\!]$. By the transitivity of the containment relation on PERs, $[\![\sigma]\!] \subseteq [\![\tau]\!]$.

Case Sub-Arrow: $\dfrac{\tau_1 \le \sigma_1 \qquad \sigma_2 \le \tau_2}{\sigma_1 \to \sigma_2 \ \le \ \tau_1 \to \tau_2}$

By the induction hypothesis, $[\![\tau_1]\!] \subseteq [\![\sigma_1]\!]$ and $[\![\sigma_2]\!] \subseteq [\![\tau_2]\!]$. Assume that $p\ \{[\![\sigma_1 \to \sigma_2]\!]\}\ q$. We must show that $p\ \{[\![\tau_1 \to \tau_2]\!]\}\ q$. Reason as follows:

$$\begin{array}{ll} & p\ \{[\![\sigma_1 \to \sigma_2]\!]\}\ q \\ \Leftrightarrow & p\ \{[\![\sigma_1]\!] \to [\![\sigma_2]\!]\}\ q \\ \Leftrightarrow & \text{for all } r \text{ and } s, \quad r\ \{[\![\sigma_1]\!]\}\ s \text{ implies } p \cdot r\ \{[\![\sigma_2]\!]\}\ q \cdot s \\ \Rightarrow & \text{for all } r \text{ and } s, \quad r\ \{[\![\tau_1]\!]\}\ s \text{ implies } p \cdot r\ \{[\![\tau_2]\!]\}\ q \cdot s \\ \Leftrightarrow & p\ \{[\![\tau_1]\!] \to [\![\tau_2]\!]\}\ q \\ \Leftrightarrow & p\ \{[\![\tau_1 \to \tau_2]\!]\}\ q. \end{array}$$

Case Sub-Inter-G: $\dfrac{\text{for all } i,\ \sigma \le \tau_i}{\sigma \ \le \ \bigwedge[\tau_1..\tau_n]}$

By the induction hypothesis, $[\![\sigma]\!] \subseteq [\![\tau_i]\!]$ for every i. Thus, $p\ \{[\![\sigma]\!]\}\ q$ implies $p\ \{[\![\tau_i]\!]\}\ q$ for every i. By definition, $p\ \{\bigsqcap[[\![\tau_1]\!] .. \tau_n]\}\ q$, that is, $p\ \{[\![\bigwedge[\tau_1..\tau_n]]\!]\}\ q$.

Case Sub-Inter-LB: $\bigwedge[\tau_1..\tau_n] \ \le \ \tau_i$

Reason as follows:

$$\begin{array}{ll} & p\ \{[\![\bigwedge[\tau_1..\tau_n]]\!]\}\ q \\ \Leftrightarrow & p\ \{\bigsqcap[[\![\tau_1]\!] .. [\![\tau_n]\!]]\}\ q \\ \Rightarrow & p\ \{[\![\tau_i]\!]\}\ q. \end{array}$$

Case Sub-Union-L: $\dfrac{\text{for all } i,\ \sigma_i \le \tau}{\bigvee[\sigma_1..\sigma_n] \ \le \ \tau}$

If $p\ \{[\![\bigvee[\tau_1..\tau_n]]\!]\}\ q$, then by definition $p\ \{\bigsqcup[[\![\sigma_1]\!] .. [\![\sigma_n]\!]]\}\ q$, which by Lemma 3.12 implies the existence of $o_1..o_{j+1} \in \Omega$ and $A_1..A_j \in \{[\![\sigma_1]\!] .. [\![\sigma_n]\!]\}$ such that

$$p = o_1\ \{A_1\}\ o_2 \cdots \{A_j\}\ o_{j+1} = q.$$

By the induction hypothesis, $[\![\sigma_i]\!] \subseteq [\![\tau]\!]$ for every i, so

$$o_1\ \{[\![\tau]\!]\}\ o_2 \cdots \{[\![\tau]\!]\}\ o_{j+1}.$$

Hence, by the transitivity of $[\![\tau]\!]$,

$$p\ \{[\![\tau]\!]\}\ q.$$

Case Sub-Union-UB: $\tau_i \ \le \ \bigvee[\tau_1..\tau_n]$

By definition, $[\![\tau_i]\!] \subseteq \bigsqcup[[\![\tau_1]\!] .. [\![\tau_n]\!]] = [\![\bigvee[\tau_1..\tau_n]]\!]$.

Case Dist-AI-R: $\bigwedge[\sigma \to \tau_1 .. \sigma \to \tau_n] \ \le \ \sigma \to \bigwedge[\tau_1..\tau_n]$

Reason as follows:

$$\begin{array}{ll} & p\ \{[\![\bigwedge[\sigma \to \tau_1 .. \sigma \to \tau_n]]\!]\}\ q \\ \Leftrightarrow & p\ \{\bigsqcap[[\![\sigma \to \tau_1]\!] .. [\![]\!]\sigma \to \tau_n\}\ q \\ \Leftrightarrow & \text{for all } i, \text{ for all } r \text{ and } s,\ r\ \{[\![\sigma]\!]\}\ s \text{ implies } p \cdot r\ \{[\![\tau_i]\!]\}\ q \cdot s \\ \Leftrightarrow & \text{for all } r \text{ and } s,\ r\ \{[\![\sigma]\!]\}\ s \text{ implies (for all } i,\ p \cdot r\ \{[\![\tau_i]\!]\}\ q \cdot s) \\ \Leftrightarrow & \text{for all } r \text{ and } s,\ r\ \{[\![\sigma]\!]\}\ s \text{ implies } p \cdot r\ \{\bigsqcap[[\![\tau_1]\!] .. [\![\tau_n]\!]]\}\ q \cdot s \\ \Leftrightarrow & \text{for all } r \text{ and } s,\ r\ \{[\![\sigma]\!]\}\ s \text{ implies } p \cdot r\ \{[\![\bigwedge[\tau_1..\tau_n]]\!]\}\ q \cdot s \\ \Leftrightarrow & p\ \{[\![\sigma]\!] \to [\![\bigwedge[\tau_1..\tau_n]]\!]\}\ q \\ \Leftrightarrow & p\ \{[\![\sigma \to \bigwedge[\tau_1..\tau_n]]\!]\}\ q. \end{array}$$

Case DIST-AU-L: $\bigwedge[\sigma_1 \to \tau \mathinner{..} \sigma_n \to \tau] \ \leq \ \bigvee[\sigma_1 \mathinner{..} \sigma_n] \to \tau$

Suppose p and q are related by $[\![\bigwedge[\sigma_1 \to \tau \mathinner{..} \sigma_n \to \tau]]\!]$. Then:

$$\begin{array}{ll} & p\ \{[\![\bigwedge[\sigma_1 \to \tau \mathinner{..} \sigma_n \to \tau]]\!]\}\ q \\ \Leftrightarrow & p\ \{\bigsqcap[([\![\sigma_1]\!] \to [\![\tau]\!]) \mathinner{..} ([\![\sigma_n]\!] \to [\![\tau]\!])]\}\ q \\ \Leftrightarrow & \text{for all } i,\ p\ \{[\![\sigma_i]\!] \to [\![\tau]\!]\}\ q \\ \Leftrightarrow & \text{for all } i, \text{ for all } r \text{ and } s,\ r\ \{[\![\sigma_i]\!]\}\ s \text{ implies } p \cdot r\ \{[\![\tau]\!]\}\ q \cdot s. \end{array}$$

Now, assume that

$$r\ \{\bigsqcup[[\![\sigma_1]\!] \mathinner{..} [\![\sigma_n]\!]]\}\ s,$$

for some r and s. We must show that

$$p \cdot r\ \{[\![\tau]\!]\}\ q \cdot s.$$

By Lemma 3.12, there exist $o_1 \mathinner{..} o_{j+1} \in \Omega$ and $A_1 \mathinner{..} A_j \in \{[\![\sigma_1]\!] \mathinner{..} [\![\sigma_n]\!]\}$ such that

$$r = o_1\ \{A_1\}\ o_2\ \{A_2\} \cdots \{A_j\}\ o_{j+1} = s.$$

This implies

$$\begin{array}{ll} & p \cdot o_1\ \{[\![\tau]\!]\}\ q \cdot o_2 \\ \text{and} & p \cdot o_2\ \{[\![\tau]\!]\}\ q \cdot o_3 \\ \vdots & \\ \text{and} & p \cdot o_j\ \{[\![\tau]\!]\}\ q \cdot o_{j+1}. \end{array}$$

Since $o_i\ \{A_i\}\ o_{i+1}$ implies $o_i\ \{A_i\}\ o_i$, we also have

$$\begin{array}{ll} & p \cdot o_2\ \{[\![\tau]\!]\}\ q \cdot o_2 \\ \text{and} & p \cdot o_3\ \{[\![\tau]\!]\}\ q \cdot o_3 \\ \vdots & \\ \text{and} & p \cdot o_{j+1}\ \{[\![\tau]\!]\}\ q \cdot o_{j+1}, \end{array}$$

and so by symmetry,

$$\begin{array}{ll} & q \cdot o_2\ \{[\![\tau]\!]\}\ p \cdot o_2 \\ \text{and} & q \cdot o_3\ \{[\![\tau]\!]\}\ p \cdot o_3 \\ \vdots & \\ \text{and} & q \cdot o_{j+1}\ \{[\![\tau]\!]\}\ p \cdot o_{j+1}. \end{array}$$

Combining these gives

$$p \cdot o_1\ \{[\![\tau]\!]\}\ q \cdot o_2\ \{[\![\tau]\!]\}\ p \cdot o_2\ \{[\![\tau]\!]\}\ q \cdot o_3 \cdots \{[\![\tau]\!]\}\ q \cdot o_{j+1}.$$

By the transitivity of $[\![\tau]\!]$,

$$p \cdot r\ \{[\![\tau]\!]\}\ q \cdot s.$$

Since r and s were chosen arbitrarily, we have shown that

$$p\ \{\bigsqcup[[\![\sigma_1]\!] \mathinner{..} [\![\sigma_n]\!]]\}\ q,$$

that is,

$$p\ \{[\![\bigvee[\sigma_1 \mathinner{..} \sigma_n]]\!]\}\ q.$$

$\square$

Definition 3.21: The *interpretation* of a typed $\lambda^\vee$ term $\mathtt{e}$, where $\Gamma \vdash \mathtt{e} \in \tau$, is:

$$[\![\Gamma \vdash \mathtt{e} \in \tau]\!]E = [\langle\!\langle \mathtt{e} \rangle\!\rangle E : [\![\tau]\!]].$$

Definition 3.22: An environment $E : \mathrm{V\scriptstyle{AR}} \rightarrow \Omega$ is said to *satisfy* a context Γ, written $E \models \Gamma$, iff for every $\mathtt{x} \in \mathrm{dom}(\Gamma)$, $E(\mathtt{x}) \in \mathrm{dom}([\![\Gamma(\mathtt{x})]\!])$.

Proposition 3.23: If

$$\begin{array}{l} E_1 \models \Gamma \\ E_2 \models \Gamma \\ \Gamma \vdash \mathtt{e} \in \tau, \end{array}$$

and, for every $\mathtt{x} \in \mathrm{dom}(\Gamma)$,

$$E_1(\mathtt{x}) \; \{[\![\Gamma(\mathtt{x})]\!]\} \; E_2(\mathtt{x}),$$

then

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2.$$

Proof: By induction on the structure of a derivation of $\Gamma \vdash \mathtt{e} \in \tau$.

Case SUBSUMPTION: $\dfrac{\Gamma \vdash \mathtt{e} \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} \in \tau}$

By the induction hypothesis, $\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\sigma]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2$. By Proposition 3.20, $[\![\sigma]\!] \subseteq [\![\tau]\!]$, so $\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2$.

Case VAR: $\Gamma_1, \mathtt{x} : \tau, \Gamma_2 \vdash \mathtt{x} \in \tau$

By assumption, $\langle\!\langle \mathtt{x} \rangle\!\rangle E_1 = E_1(\mathtt{x}) \; \{[\![\tau]\!]\} \; E_2(\mathtt{x}) = \langle\!\langle \mathtt{x} \rangle\!\rangle E_2$.

Case ARROW-I: $\dfrac{\Gamma, \mathtt{x} : \sigma_i \vdash \mathtt{e} \in \tau}{\Gamma, \vdash \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e} \in \sigma_i \rightarrow \tau}$

Assume that $p \; \{[\![\sigma_i]\!]\} \; q$ for some p and q. By the definition of $\langle\!\langle - \rangle\!\rangle$,

$$\begin{array}{rcl} (\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e} \rangle\!\rangle E) \cdot m & = & (\lambda m.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow m])(m) \\ & = & \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow m], \end{array}$$

for any m. In particular,

$$(\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e} \rangle\!\rangle E) \cdot p = \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow p]$$

and

$$(\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e} \rangle\!\rangle E) \cdot q = \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow q].$$

By the induction hypothesis, if

$$\begin{array}{l} E_1' \models \Gamma, \mathtt{x} : \sigma_i \\ E_2' \models \Gamma, \mathtt{x} : \sigma_i \end{array}$$

and for every $\mathtt{y} \in \mathrm{dom}(\Gamma, \mathtt{x} : \sigma_i)$,

$$E_1'(\mathtt{y}) \; \{[\![\Gamma(\mathtt{y})]\!]\} \; E_2'(\mathtt{y}),$$

then

$$(\langle\!\langle \mathtt{e} \rangle\!\rangle E_1') \; \{[\![\tau]\!]\} \; (\langle\!\langle \mathtt{e} \rangle\!\rangle E_2').$$

In particular,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1[\mathtt{x} \leftarrow p] \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2[\mathtt{x} \leftarrow q],$$

that is,

$$\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \rangle\!\rangle E_1 \cdot p \; \{[\![\tau]\!]\} \; \langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \rangle\!\rangle E_2 \cdot p.$$

Since p and q were chosen arbitrarily, the definition of $[\![\sigma_i]\!] \rightarrow [\![\tau]\!]$ gives

$$\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \rangle\!\rangle E_1 \; \{[\![\sigma_i]\!] \rightarrow [\![\tau]\!]\} \; \langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \rangle\!\rangle E_2,$$

that is,

$$\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \rangle\!\rangle E_1 \; \{[\![\sigma_i \rightarrow \tau]\!]\} \; \langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \rangle\!\rangle E_2.$$

Case ARROW-E: $\dfrac{\Gamma \vdash \mathtt{f} \in \sigma \rightarrow \tau \qquad \Gamma \vdash \mathtt{e} \in \sigma}{\Gamma \vdash \mathtt{f\ e} \in \tau}$

By the induction hypothesis,

$$\langle\!\langle \mathtt{f} \rangle\!\rangle E_1 \; \{[\![\sigma \rightarrow \tau]\!]\} \; \langle\!\langle \mathtt{f} \rangle\!\rangle E_2$$

and

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2.$$

By the definition of $[\![\sigma \rightarrow \tau]\!]$,

$$\langle\!\langle \mathtt{f\ e} \rangle\!\rangle E_1 = (\langle\!\langle \mathtt{f} \rangle\!\rangle E_1) \cdot (\langle\!\langle \mathtt{e} \rangle\!\rangle E_1) \; \{[\![\tau]\!]\} \; (\langle\!\langle \mathtt{f} \rangle\!\rangle E_2) \cdot (\langle\!\langle \mathtt{e} \rangle\!\rangle E_2) = \langle\!\langle \mathtt{f\ e} \rangle\!\rangle E_2.$$

Case INTER-I: $\dfrac{\text{for all } i,\ \Gamma \vdash \mathtt{e} \in \tau_i}{\Gamma \vdash \mathtt{e} \in \bigwedge[\tau_1..\tau_n]}$

By the induction hypothesis, $\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\tau_i]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2$ for all i. By the definition of $\bigsqcap[[\![\tau_1]\!] \,..\, [\![\tau_n]\!]]$,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{\bigsqcap[[\![\tau_1]\!] \,..\, [\![\tau_n]\!]]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2,$$

that is,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\bigwedge[\tau_1..\tau_n]]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2.$$

Case UNION-E: $\dfrac{\Gamma \vdash \mathtt{e\ in} \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{e}' \in \tau}$

By the induction hypothesis,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 \; \{[\![\bigvee[\tau_1..\tau_n]]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E_2.$$

By Lemma 3.12, there are $o_1..o_{j+1} \in \Omega$ and $A_1..A_j \in \{[\![\tau_1]\!] \,..\, [\![\tau_n]\!]\}$ such that

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E_1 = o_1 \; \{A_1\} \; o_2 \cdots \{A_j\} \; o_{j+1} = \langle\!\langle \mathtt{e} \rangle\!\rangle E_2.$$

We show by induction on j that

$$\langle\!\langle \mathtt{e}' \rangle\!\rangle E_1[\mathtt{x} \leftarrow o_1] \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_{j+1}].$$

Assume $j = 1$. Since $o_1 \; \{[\![\tau_i]\!]\} \; o_2$ for some i, the symmetry and transitivity of $[\![\tau_i]\!]$ give $o_1 \; \{[\![\tau_i]\!]\} \; o_1$ and $o_2 \; \{[\![\tau_i]\!]\} \; o_2$. By the outer induction hypothesis,

$$\langle\!\langle \mathsf{e}' \rangle\!\rangle E_1[\mathtt{x} \leftarrow o_1] \; \{[\![\tau]\!]\} \; \langle\!\langle \mathsf{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_2],$$

as required. Now assume $j > 1$. By the inner induction hypothesis, we know that

$$\langle\!\langle \mathsf{e}' \rangle\!\rangle E_1[\mathtt{x} \leftarrow o_1] \; \{[\![\tau]\!]\} \; \langle\!\langle \mathsf{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_j].$$

Since $E_1(\mathtt{y}) \; \{[\![\Gamma(\mathtt{y})]\!]\} \; E_2(\mathtt{y})$ for all $\mathtt{y} \in \mathrm{dom}(\Gamma)$, the symmetry and transitivity of $[\![\Gamma(\mathtt{y})]\!]$ gives

$$E_2(\mathtt{y}) \; \{[\![\Gamma(\mathtt{y})]\!]\} \; E_2(\mathtt{y})$$

for all $\mathtt{y} \in \mathrm{dom}(\Gamma)$. So the outer induction hypothesis applies, giving

$$\langle\!\langle \mathsf{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_j] \; \{[\![\tau]\!]\} \; \langle\!\langle \mathsf{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_{j+1}].$$

By the transitivity of $[\![\tau]\!]$,

$$\langle\!\langle \mathsf{e}' \rangle\!\rangle E_1[\mathtt{x} \leftarrow o_1] \; \{[\![\tau]\!]\} \; \langle\!\langle \mathsf{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_{j+1}],$$

as required.

Finally, Lemma 3.18 gives

$$\langle\!\langle \mathsf{e}' \rangle\!\rangle E_1[\mathtt{x} \leftarrow o_1] = \langle\!\langle [\mathsf{e}/\mathtt{x}]\mathsf{e}' \rangle\!\rangle E_1$$

and

$$\langle\!\langle \mathsf{e}' \rangle\!\rangle E_2[\mathtt{x} \leftarrow o_{j+1}] = \langle\!\langle [\mathsf{e}/\mathtt{x}]\mathsf{e}' \rangle\!\rangle E_2,$$

so

$$\langle\!\langle [\mathsf{e}/\mathtt{x}]\mathsf{e}' \rangle\!\rangle E_1 \; \{[\![\tau]\!]\} \; \langle\!\langle [\mathsf{e}/\mathtt{x}]\mathsf{e}' \rangle\!\rangle E_2.$$

□

Corollary 3.24: If $E \models \Gamma$ and $\Gamma \vdash \mathsf{e} \in \tau$, then $[\![\Gamma \vdash \mathsf{e} \in \tau]\!]E$ is well defined, that is, $\langle\!\langle \mathsf{e} \rangle\!\rangle E \in \mathrm{dom}([\![\tau]\!])$.

Proposition 3.25: If $\Gamma \vdash \mathsf{e} = \mathsf{e}' \in \tau$, then for any environment E satisfying Γ,

$$\langle\!\langle \mathsf{e} \rangle\!\rangle E \; \{[\![\tau]\!]\} \; \langle\!\langle \mathsf{e}' \rangle\!\rangle E.$$

Proof: By induction on the structure of a derivation of $\Gamma \vdash \mathsf{e} = \mathsf{e}' \in \tau$. Lemma 3.2 and Corollary 3.24 are used in several places to justify the assumption that $\langle\!\langle \mathsf{e} \rangle\!\rangle E \in \mathrm{dom}(\tau)$ and $\langle\!\langle \mathsf{e}' \rangle\!\rangle E \in \mathrm{dom}(\tau)$.

Case EQ-REFL: $\dfrac{\Gamma \vdash \mathsf{e} \in \tau}{\Gamma \vdash \mathsf{e} = \mathsf{e} \in \tau}$

We are given that $\Gamma \vdash \mathsf{e} \in \tau$. By Corollary 3.24,

$$\langle\!\langle \mathsf{e} \rangle\!\rangle E \; \{[\![\tau]\!]\} \; \langle\!\langle \mathsf{e} \rangle\!\rangle E.$$

Case EQ-SYMM: $\dfrac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{e}' = \mathtt{e} \in \tau}$

By the induction hypothesis and the symmetry of equality.

Case EQ-TRANS: $\dfrac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau \qquad \Gamma \vdash \mathtt{e}' = \mathtt{e}'' \in \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e}'' \in \tau}$

By the induction hypothesis and the transitivity of equality.

Case EQ-BETA: $\dfrac{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) \in \tau}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) = [\mathtt{e}/\mathtt{x}]\mathtt{f} \in \tau}$

By definition

$$
\begin{aligned}
& \langle\!\langle (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) \rangle\!\rangle E \\
= \;& \langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f} \rangle\!\rangle E \cdot \langle\!\langle \mathtt{e} \rangle\!\rangle E \\
= \;& \mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{f} \rangle\!\rangle E[\mathtt{x} \leftarrow m]) \cdot \langle\!\langle \mathtt{e} \rangle\!\rangle E \\
= \;& \langle\!\langle \mathtt{f} \rangle\!\rangle E[\mathtt{x} \leftarrow \langle\!\langle \mathtt{e} \rangle\!\rangle E] \\
= \;& \langle\!\langle [\mathtt{e}/\mathtt{x}]\mathtt{f} \rangle\!\rangle E,
\end{aligned}
$$

by Lemma 3.18. So by Corollary 3.24,

$$\langle\!\langle (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) \rangle\!\rangle E \; \{[\![\tau]\!]\} \; \langle\!\langle [\mathtt{e}/\mathtt{x}]\mathtt{f} \rangle\!\rangle E.$$

Case EQ-ETA: $\dfrac{\Gamma \vdash \mathtt{e} \in \sigma_i \to \tau \qquad \mathtt{x} \notin \mathrm{FV}(\mathtt{e})}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}\ \mathtt{x}) = \mathtt{e} \in \sigma_i \to \tau}$

By definition

$$
\begin{aligned}
& \langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}\ \mathtt{x} \rangle\!\rangle E \\
= \;& \mathrm{uabs}(\lambda n.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow n] \cdot \langle\!\langle \mathtt{x} \rangle\!\rangle E[\mathtt{x} \leftarrow n]) \\
= \;& \mathrm{uabs}(\lambda n.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow n] \cdot n \\
= \;& \mathrm{uabs}(\lambda n.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E \cdot n,
\end{aligned}
$$

by Lemma 3.17. But by the definition of uabs, the last expression is a number denoting the same function as $\langle\!\langle \mathtt{e} \rangle\!\rangle E$, so by Corollary 3.24,

$$\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}\ \mathtt{x} \rangle\!\rangle E \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e} \rangle\!\rangle E.$$

Case EQ-SUBSUMPTION: $\dfrac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau}$

By the induction hypothesis,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E \; \{[\![\sigma]\!]\} \; \langle\!\langle \mathtt{e}' \rangle\!\rangle E.$$

By Proposition 3.20, $[\![\sigma]\!] \subseteq [\![\tau]\!]$. So

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E \; \{[\![\tau]\!]\} \; \langle\!\langle \mathtt{e}' \rangle\!\rangle E.$$

Case EQ-ARROW-I: $\dfrac{\Gamma,\, \mathtt{x} : \sigma_i \;\vdash\; \mathtt{e} = \mathtt{e}' \;\in\; \tau}{\Gamma \;\vdash\; (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e}) = (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e}') \;\in\; \sigma_i \rightarrow \tau}$

Assume that $p \;\{[\![\sigma_i]\!]\}\; q$. By the induction hypothesis,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow p] \;\{[\![\tau]\!]\}\; \langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow p].$$

By Proposition 3.23,

$$\langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow p] \;\{[\![\tau]\!]\}\; \langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow q].$$

By the transitivity of $[\![\tau]\!]$,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow p] \;\{[\![\tau]\!]\}\; \langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow q],$$

that is,

$$\mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow m]) \cdot p \;\{[\![\tau]\!]\}\; \mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow m]) \cdot q.$$

Since p and q were chosen arbitrarily, we have

$$\mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{e} \rangle\!\rangle E[\mathtt{x} \leftarrow m]) \;\{[\![\sigma_i]\!] \rightarrow [\![\tau]\!]\}\; \mathrm{uabs}(\lambda m.\, \langle\!\langle \mathtt{e}' \rangle\!\rangle E[\mathtt{x} \leftarrow m]),$$

that is,

$$\langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e} \rangle\!\rangle E \;\{[\![\sigma_i \rightarrow \tau]\!]\}\; \langle\!\langle \lambda \mathtt{x}{:}\sigma_1..\sigma_n.\, \mathtt{e}' \rangle\!\rangle E.$$

Case EQ-ARROW-E: $\dfrac{\Gamma \;\vdash\; \mathtt{f} = \mathtt{f}' \;\in\; \sigma \rightarrow \tau \qquad \Gamma \;\vdash\; \mathtt{e} = \mathtt{e}' \;\in\; \sigma}{\Gamma \;\vdash\; (\mathtt{f}\ \mathtt{e}) = (\mathtt{f}'\ \mathtt{e}') \;\in\; \tau}$

By the induction hypothesis,

$$\langle\!\langle \mathtt{f} \rangle\!\rangle E \;\{[\![\sigma \rightarrow \tau]\!]\}\; \langle\!\langle \mathtt{f}' \rangle\!\rangle E,$$

that is,

$$\text{for all } p \text{ and } q,\; p \;\{[\![\sigma]\!]\}\; q \text{ implies } \langle\!\langle \mathtt{f} \rangle\!\rangle E \cdot p \;\{[\![\tau]\!]\}\; \langle\!\langle \mathtt{f}' \rangle\!\rangle E \cdot q,$$

and

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E \;\{[\![\sigma]\!]\}\; \langle\!\langle \mathtt{e}' \rangle\!\rangle E.$$

So

$$\langle\!\langle \mathtt{f} \rangle\!\rangle E \cdot \langle\!\langle \mathtt{e} \rangle\!\rangle E \;\{[\![\tau]\!]\}\; \langle\!\langle \mathtt{f}' \rangle\!\rangle E \cdot \langle\!\langle \mathtt{e}' \rangle\!\rangle E,$$

that is

$$\langle\!\langle \mathtt{f}\ \mathtt{e} \rangle\!\rangle E \;\{[\![\tau]\!]\}\; \langle\!\langle \mathtt{f}'\mathtt{e}' \rangle\!\rangle E.$$

Case EQ-INTER-I: $\dfrac{\text{for all } i,\; \Gamma \;\vdash\; \mathtt{e} = \mathtt{e}' \;\in\; \tau_i}{\Gamma \;\vdash\; \mathtt{e} = \mathtt{e}' \;\in\; \bigwedge[\tau_1..\tau_n]}$

By the induction hypothesis,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E \;\{[\![\tau_i]\!]\}\; \langle\!\langle \mathtt{e}' \rangle\!\rangle E$$

for all i. By definition,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E \;\{\textstyle\prod[[\![\tau_1]\!] \,..\, [\![\tau_n]\!]]\}\; \langle\!\langle \mathtt{e}' \rangle\!\rangle E,$$

that is,

$$[\![\Gamma \;\vdash\; \mathtt{e} \in \textstyle\bigwedge[\tau_1..\tau_n]]\!] E = [\![\Gamma \;\vdash\; \mathtt{e}' \in \textstyle\bigwedge[\tau_1..\tau_n]]\!] E.$$

Case Eq-Union-E: $\dfrac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{f} = \mathtt{f}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{f} = [\mathtt{e}'/\mathtt{x}]\mathtt{f}' \in \tau}$

By the induction hypothesis,

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E \ \{\bigsqcup[[\![\tau_1]\!] \,..\, [\![\tau_n]\!]]\} \ \langle\!\langle \mathtt{e}' \rangle\!\rangle E.$$

By Lemma 3.12, there exist $o_1..o_{j+1} \in \Omega$ and $A_1..A_j \in \{\tau_1..\tau_n\}$ such that

$$\langle\!\langle \mathtt{e} \rangle\!\rangle E = o_1 \ \{A_1\} \ o_2 \cdots \{A_j\} \ o_{j+1} = \langle\!\langle \mathtt{e}' \rangle\!\rangle E.$$

For each $1 \leq k \leq j$, the induction hypothesis gives

$$\langle\!\langle \mathtt{f} \rangle\!\rangle E[\mathtt{x} \leftarrow o_k] \ \{[\![\tau]\!]\} \ \langle\!\langle \mathtt{f}' \rangle\!\rangle E[\mathtt{x} \leftarrow o_k]$$

and Proposition 3.23 gives

$$\langle\!\langle \mathtt{f}' \rangle\!\rangle E[\mathtt{x} \leftarrow o_k] \ \{[\![\tau]\!]\} \ \langle\!\langle \mathtt{f}' \rangle\!\rangle E[\mathtt{x} \leftarrow o_{k+1}],$$

so by the transitivity of $[\![\tau]\!]$ we have

$$\langle\!\langle \mathtt{f} \rangle\!\rangle E[\mathtt{x} \leftarrow o_1] \ \{[\![\tau]\!]\} \ \langle\!\langle \mathtt{f}' \rangle\!\rangle E[\mathtt{x} \leftarrow o_{j+1}],$$

that is,

$$\langle\!\langle \mathtt{f} \rangle\!\rangle E[\mathtt{x} \leftarrow \langle\!\langle \mathtt{e} \rangle\!\rangle E] \ \{[\![\tau]\!]\} \ \langle\!\langle \mathtt{f}' \rangle\!\rangle E[\mathtt{x} \leftarrow \langle\!\langle \mathtt{e}' \rangle\!\rangle E].$$

Applying Lemma 3.18 to both sides gives

$$\langle\!\langle [\mathtt{e}/\mathtt{x}]\mathtt{f} \rangle\!\rangle E \ \{[\![\tau]\!]\} \ \langle\!\langle [\mathtt{e}'/\mathtt{x}]\mathtt{f}' \rangle\!\rangle E.$$

□

3.4 Unsoundness of the Non-Derivable Rules

The PER model can be used to demonstrate that the nonderivable subtyping rules listed in Section 1.4 should not be expected to hold.

- The rule

  $$\sigma \rightarrow \bigvee[\tau_1..\tau_n] \ \leq \ \bigvee[\sigma \rightarrow \tau_1 \,..\, \sigma \rightarrow \tau_n] \qquad \text{(Bad-Dist-AU-R')}$$

  is not valid because, for example, the function

  $$\mathtt{f} \ = \{1 \mapsto 1,\ 2 \mapsto \mathtt{true}\}$$

  which maps 1 to 1 and 2 to `true` and diverges on all other inputs, is an element of $[\![\mathtt{Int} \rightarrow \bigvee[\mathtt{Int}, \mathtt{Bool}]]\!]$, given the obvious interpretations of base types, but not of $[\![\bigvee[\mathtt{Int} \rightarrow \mathtt{Int},\ \mathtt{Int} \rightarrow \mathtt{Bool}]]\!]$.

- The rule

  $$\bigwedge[\sigma_1..\sigma_n] \rightarrow \tau \ \leq \ \bigvee[\sigma_1 \rightarrow \tau \,..\, \sigma_n \rightarrow \tau] \qquad \text{(Bad-Dist-AU-L')}$$

  is not valid because, for example, the function

  $$\mathtt{g} \ = \{-1 \mapsto 0,\ 0 \mapsto \mathtt{true},\ 1 \mapsto 0\}$$

  is an element of $[\![\bigwedge[\mathtt{NonNeg}, \mathtt{NonPos}] \rightarrow \mathtt{Bool}]\!]$, but not of $[\![\bigwedge[\mathtt{NonNeg} \rightarrow \mathtt{Bool},\ \mathtt{NonPos} \rightarrow \mathtt{Bool}]]\!]$.

- The rule

$$\bigwedge[\sigma_1 \vee \tau \mathinner{..} \sigma_n \vee \tau] \;\leq\; \bigwedge[\sigma_1 \mathinner{..} \sigma_n] \vee \tau \qquad \text{(Bad-Dist-UI')}$$

fails when

$$\begin{array}{rcl} [\![\tau]\!] & = & \{1,2\}\,\{3,4\} \\ [\![\sigma_1]\!] & = & \{2,3\} \\ [\![\sigma_2]\!] & = & \{1,4\}. \end{array}$$

- The rule

$$\bigvee[\sigma_1 \mathinner{..} \sigma_n] \wedge \tau \;\leq\; \bigvee[\sigma_1 \wedge \tau \mathinner{..} \sigma_n \wedge \tau] \qquad \text{(Bad-Dist-IU')}$$

fails when

$$\begin{array}{rcl} [\![\tau]\!] & = & \{1,4\}\,\{2,3\} \\ [\![\sigma_1]\!] & = & \{1,2\}\,\{3,4\} \\ [\![\sigma_2]\!] & = & \{1,3\}\,\{2,4\}. \end{array}$$

4 Typechecking

This section explores the decidability of the subtype relation and well-typedness judgement for $\lambda^{\vee}$.

N.b. The proof theory of $\lambda^{\vee}$ is currently the area where my understanding is least developed. The absence of the deMorgan laws for distributing meets and joins makes it difficult to normalize type expressions enough to make the correctness proofs of algorithms involving them tractable. At the moment, this section contains mostly conjectures and ideas for possible approaches.

4.1 Basic Lemmas

Definition 4.1: Write $D : \sigma \leq \tau$ to mean that D is a valid derivation whose conclusion is $\sigma \leq \tau$.

Definition 4.2: The rules

$$\frac{\text{for all } i,\ \sigma_i \leq \tau_i}{\bigwedge[\sigma_1 \mathinner{..} \sigma_n] \;\leq\; \bigwedge[\tau_1 \mathinner{..} \tau_n]} \qquad \text{(Sub-Congr-Inter)}$$

and

$$\frac{\text{for all } i,\ \sigma_i \leq \tau_i}{\bigvee[\sigma_1 \mathinner{..} \sigma_n] \;\leq\; \bigvee[\tau_1 \mathinner{..} \tau_n]} \qquad \text{(Sub-Congr-Union)}$$

may be derived from Sub-Inter, Sub-Union, and Sub-Trans.

We say that a derivation $D : \sigma \leq \tau$ from the rules in Section 1.2 plus Sub-Congr-Inter and Sub-Congr-Union *has no inner occurrences of* Sub-Trans if Sub-Trans is never used in D to establish a hypothesis of any rule other than Sub-Trans.

Lemma 4.3: Let $D : \sigma \leq \tau$. Then there exists a derivation $D' : \sigma \leq \tau$ with no inner occurrences of Sub-Trans.

Equivalently, if $D : \sigma \leq \tau$ then there exists a finite sequence of types $\theta_1 \mathinner{..} \theta_n$ such that

$$\sigma \equiv \theta_1 \leq \theta_2 \leq \cdots \leq \theta_n \equiv \tau,$$

where each inclusion can be proved without using Sub-Trans (but perhaps using Sub-Congr-Inter and Sub-Congr-Union).

Proof: By induction on the structure of D.

Case SUB-REFL:
Immediate.

Case SUB-TRANS: $\dfrac{\sigma \le \theta \qquad \theta \le \tau}{\sigma \le \tau}$

By the induction hypothesis, there exist derivations $D_1 : \sigma \le \theta$ and $D_2 : \theta \le \tau$ with no inner occurrences of SUB-TRANS. D' can be constructed from D_1 and D_2 by one more application of SUB-TRANS.

Case SUB-ARROW: $\dfrac{\tau_1 \le \sigma_1 \qquad \sigma_2 \le \tau_2}{\sigma_1 \to \sigma_2 \;\le\; \tau_1 \to \tau_2}$

By the induction hypothesis, there exist sequences

$$\tau_1 \equiv \phi_m \le \phi_{m-1} \le \cdots \le \phi_1 \equiv \sigma_1$$

and

$$\sigma_2 \equiv \psi_1 \le \psi_2 \le \cdots \le \psi_n \equiv \tau_2,$$

where each inclusion can be proved without using SUB-TRANS. Assume WLOG that $m = n$. (If not, extend the shorter sequence with enough copies of one of its members to make them the same length.)

Now use SUB-ARROW n times to construct the following sequence of inclusions:

$$\sigma_1 \to \sigma_2 \;\equiv\; \phi_1 \to \psi_1 \;\le\; \phi_2 \to \psi_2 \;\le\; \cdots \;\le\; \phi_m \to \psi_n \;\equiv\; \tau_1 \to \tau_2.$$

Case SUB-INTER-G: $\dfrac{\text{for all } i,\ \sigma \le \tau_i}{\sigma \;\le\; \bigwedge[\tau_1..\tau_n]}$

By the induction hypothesis, there are n sequences of types

$$\begin{gathered}\sigma \equiv \phi_{1,1} \le \phi_{1,2} \le \cdots \le \phi_{1,m_1} \equiv \tau \\ \vdots \\ \sigma \equiv \phi_{n,1} \le \phi_{n,2} \le \cdots \le \phi_{n,m_n} \equiv \tau\end{gathered}$$

where all of the inclusions may be derived without using SUB-TRANS. Assume WLOG that $m_i = m_j$ for all i and j.

Use SUB-INTER-G to construct the following sequence of inclusions:

$$\bigwedge[\phi_{1,1} .. \phi_{n,1}] \;\le\; \bigwedge[\phi_{1,2} .. \phi_{n,2}] \;\le\; \cdots \;\le\; \bigwedge[\phi_{1,m_1} .. \phi_{n,m_n}].$$

Finally, observe that $\sigma \le \bigwedge[\phi_{1,1} .. \phi_{n,1}]$ by SUB-REFL and SUB-INTER-G and $\bigwedge[\phi_{1,m_1} .. \phi_{n,m_n}] \le \tau$ by SUB-INTER-LB.

Case SUB-INTER-LB:
Immediate.

Case SUB-UNION-L:
Similar to SUB-INTER-G.

Case SUB-UNION-UB, DIST-AI-R, DIST-AU-L:
Immediate.

□

4.2 Canonical Type Expressions

It is often convenient to work with type expressions in a more restricted "canonical" or "normal" form, where arrows have been pushed downward through intersections and unions as much as possible, and where intersections and unions have been simplified to eliminate redundant conjuncts and disjuncts. The crucial conjecture about types in canonical form is that each equivalence class in the subtype preorder contains exactly one type in canonical form.

Definition 4.4: Let $<_{lex}$ be some decidable total ordering on the set of type expressions.

Definition 4.5: The one-step parallel simplification relation $\rightsquigarrow$ is defined by the following rules:

$$\frac{\sigma \rightsquigarrow \sigma' \qquad \tau \rightsquigarrow \tau'}{\sigma \rightarrow \sigma' \rightsquigarrow \tau \rightarrow \tau'} \qquad (\rightsquigarrow\text{-Congr-Arrow})$$

$$\frac{\text{for all } i,\ \sigma_i \rightsquigarrow \tau_i}{\bigwedge[\sigma_1..\sigma_n] \rightsquigarrow \bigwedge[\tau_1..\tau_n]} \qquad (\rightsquigarrow\text{-Congr-Inter})$$

$$\frac{\text{for all } i,\ \sigma_i \rightsquigarrow \tau_i}{\bigvee[\sigma_1..\sigma_n] \rightsquigarrow \bigvee[\tau_1..\tau_n]} \qquad (\rightsquigarrow\text{-Congr-Union})$$

$$\sigma \rightarrow \bigwedge[\tau_1..\tau_n] \rightsquigarrow \bigwedge[\sigma \rightarrow \tau_1 \,..\, \sigma \rightarrow \tau_n] \qquad (\rightsquigarrow\text{-Dist-IA-R})$$

$$\bigvee[\sigma_1..\sigma_n] \rightarrow \tau \rightsquigarrow \bigwedge[\sigma_1 \rightarrow \tau \,..\, \sigma_n \rightarrow \tau] \qquad (\rightsquigarrow\text{-Dist-IA-L})$$

$$\bigwedge[\tau_1 \,..\, \bigwedge[\sigma_1..\sigma_m] \,..\, \tau_n] \rightsquigarrow \bigwedge[\tau_1 \,..\, \sigma_1 \,..\, \sigma_m \,..\, \tau_n] \qquad (\rightsquigarrow\text{-Flatten-Inter})$$

$$\bigvee[\tau_1 \,..\, \bigvee[\sigma_1..\sigma_m] \,..\, \tau_n] \rightsquigarrow \bigvee[\tau_1 \,..\, \sigma_1 \,..\, \sigma_m \,..\, \tau_n] \qquad (\rightsquigarrow\text{-Flatten-Union})$$

$$\bigwedge[\sigma] \rightsquigarrow \sigma \qquad (\rightsquigarrow\text{-Unnest-Inter})$$

$$\bigvee[\sigma] \rightsquigarrow \sigma \qquad (\rightsquigarrow\text{-Unnest-Union})$$

$$\frac{\bigwedge[\sigma_1 \,..\, \sigma_{i-1},\ \sigma_{i+1} \,..\, \sigma_n] \ \leq\ \sigma_i}{\bigwedge[\sigma_1 \,..\, \sigma_i \,..\, \sigma_n] \rightsquigarrow \bigwedge[\sigma_1 \,..\, \sigma_{i-1},\ \sigma_{i+1} \,..\, \sigma_n]} \qquad (\rightsquigarrow\text{-Absorb-Inter})$$

$$\frac{\sigma_i \ \leq\ \bigvee[\sigma_1 \,..\, \sigma_{i-1},\ \sigma_{i+1} \,..\, \sigma_n]}{\bigvee[\sigma_1 \,..\, \sigma_i \,..\, \sigma_n] \rightsquigarrow \bigvee[\sigma_1 \,..\, \sigma_{i-1},\ \sigma_{i+1} \,..\, \sigma_n]} \qquad (\rightsquigarrow\text{-Absorb-Union})$$

$$\frac{\sigma_i <_{lex} \sigma_j}{\bigwedge[\sigma_1 \,..\, \sigma_j \,..\, \sigma_i \,..\, \sigma_n] \rightsquigarrow \bigwedge[\sigma_1 \,..\, \sigma_i \,..\, \sigma_j \,..\, \sigma_n]} \qquad (\rightsquigarrow\text{-Sort-Inter})$$

$$\frac{\sigma_i <_{lex} \sigma_j}{\bigvee[\sigma_1 \,..\, \sigma_j \,..\, \sigma_i \,..\, \sigma_n] \rightsquigarrow \bigvee[\sigma_1 \,..\, \sigma_i \,..\, \sigma_j \,..\, \sigma_n]} \qquad (\rightsquigarrow\text{-Sort-Union})$$

Definition 4.6: A type τ is said to be in normal form if none of the $\rightsquigarrow$ rules apply to it.

Definition 4.7: $\rightsquigarrow^\star$ is the reflexive, transitive closure of $\rightsquigarrow$.

Lemma 4.8: If $\sigma \rightsquigarrow \sigma'$ then $\sigma \sim \sigma'$.

Proof: Straightforward induction on the derivation. □

Lemma 4.9: If σ is in normal form and σ_1 is a subphrase of σ, then σ_1 is in normal form.

Proof: By the $\rightsquigarrow$-Congr rules. □

Proposition 4.10: The relation $\rightsquigarrow^\star$ is strongly normalizing.

Proof: Straightforward. □

Conjecture 4.11: The relation $\rightsquigarrow^\star$ is locally confluent.

Proof: [...] □

Proposition 4.12: The relation $\rightsquigarrow^\star$ is globally confluent.

Proof: By Proposition 4.10 and Conjecture 4.11. □

Definition 4.13: Let τ be a type. Define $\tau^\star$ to be the unique normal form of τ with respect to $\rightsquigarrow^\star$.

Conjecture 4.14: For all types σ and τ, if $\sigma \sim \tau$ then $\sigma^\star = \tau^\star$.

Proof: By Lemma 4.8, it suffices to show that if σ and τ are in normal form with respect to $\rightsquigarrow$ and $\sigma \sim \tau$, then $\sigma = \tau$. [...] □

The importance of Conjecture 4.14 is that $\tau^\star$ can be taken as a canonical representative of the equivalence class of τ.

Definition 4.15: A type is in semi-canonical form if it can be put into normal form by reordering sequences of conjuncts and disjuncts.

4.3 Decidability of the Subtype Relation

Definition 4.16: The algorithm $\leq^?$ is defined on semi-canonical types as follows.

$$
\begin{array}{llll}
\sigma & \leq^? \; \bigwedge[\tau_1..\tau_n] & \text{iff} & \text{for all } i,\ \sigma \leq^? \tau_i \\
\bigvee[\sigma_1..\sigma_n] & \leq^? \; \tau & \text{iff} & \text{for all } i,\ \sigma_i \leq^? \tau \\
\sigma & \leq^? \; \bigvee[\tau_1..\tau_n] & \text{iff} & \text{for some } i,\ \sigma \leq^? \tau_i \\
\bigwedge[\sigma_1..\sigma_n] & \leq^? \; \tau & \text{iff} & \text{for some } i,\ \sigma_i \leq^? \tau \\
\sigma_1 \rightarrow \sigma_2 & \leq^? \; \tau_1 \rightarrow \tau_2 & \text{iff} & \tau_1 \leq^? \sigma_1 \text{ and } \sigma_2 \leq^? \tau_2 \\
\rho & \leq^? \; \rho' & \text{iff} & \rho = \rho' \\
\sigma_1 \rightarrow \sigma_2 & \leq^? \; \rho' & \text{iff} & \texttt{false} \\
\rho & \leq^? \; \tau_1 \rightarrow \tau_2 & \text{iff} & \texttt{false}.
\end{array}
$$

Conjecture 4.17: The definition of $\leq^?$ is well formed.

Proof: [*The idea I'm currently exploring is that the definitions of* $\rightsquigarrow$ *and* $\leq^?$ *can be staged in such a way that each can be shown decidable, under the assumption that the other is decidable when restricted to smaller types...*] □

Proposition 4.18: (Soundness) If σ and τ are semi-canonical types, then $\sigma \leq^? \tau$ implies $\sigma \leq \tau$.

Proof: Straightforward. □

Conjecture 4.19: (Completeness) If σ and τ are semi-canonical types, then $\sigma \leq \tau$ implies $\sigma \leq^? \tau$.

Proof: [...] □

4.4 Decidability of Typechecking

We now consider the decidability of typechecking for $\lambda^\vee$. Assuming that the $\leq$ relation can be decided using the methods described above, there remain several points where the typing rules are not syntax directed:

- The general SUBSUMPTION rule must be replaced by a new ARROW-E rule that checks whether the function part can be coerced to something of functional type and the argument to the corresponding domain type. This is less simple than for Forsythe, because both function and argument may have types involving $\bigvee$.
- The INTER-I rule must be replaced by a modified ARROW-I rule that infers a type for each declared domain type and conjoins the results.
- As described in Section 1.5, some very serious problems are caused by the UNION-E rule.

The following is my current best guess at an algorithm for typechecking $\lambda^\vee$. (The examples in Section 2 were generated using an implementation of this algorithm in Standard ML.)

```
(* Infer a type for a term e, taking union-elimination into account:
   Choose the smallest subterm of e whose type can be derived in env,
   excluding variables that were introduced by earlier instantiations of
   infer_type_outer and are assumed not to have any interesting union
   supertypes.  If the smallest such subterm is e itself, then there are
   no more possible applications of UNION-E to consider; use infer_type
   to find e's minimal type.  Otherwise call this subterm e_sub.  Let e'
   be the result of replacing all occurrences of e_sub in e with a new
   variable x.  Let s_sub be the type inferred for e_sub under env.  Let
   s1 .. sn be the set of all "interesting" types above s_sub (possibly
   including s_sub itself).  Let
        ti = infer_type_outer e' env[x:=si].
   Then the type of e under env is
        t = \/[t1..tn].
   *)
infer_type_outer e env =
    let e_sub = smallest_free_subterm e
        val s_sub = infer_type e_sub env
    in
       if e_sub = e
          then s_sub
          else let x = <a fresh identifier>
                   e' = replace_subterm e x e_sub
                   [s1..sn] = interesting_types_above s_sub
```

```
            in
                \/[   infer_type_outer e' (extend env x s1)
                   .. infer_type_outer e' (extend env x sn)
                  ]
            end
    end


(* Compute the set of all "interesting" types above a given type.
   This set should have the property that if e'=[e/x]f, e has type
   s, and e' has type t, then there is a derivation of "e' in t" in
   which every application of the UNION-E rule to e' at the
   outermost level with e as distinguished subphrase uses one of the
   types in (interesting_types_above s) as the type of e in the first
   hypothesis. *)
interesting_types_above \/[s1..sn] =
        (interesting_types_above s1)
      @ ...
      @ (interesting_types_above sn)
interesting_types_above /\[s1..sn] =
      [/\[t11..t1m1] .. /\[tn1..tnmn]]
      where [ [t11..t1m1] .. [tn1..tnmn]]
            = all_combinations [    [interesting_types_above s1]
                                 .. [interesting_types_above sn] ]
interesting_types_above s =
      [s]


(* Find the minimal type of a term, assuming that there's no need to
   consider union types until we go through an abstraction. *)
fun infer_type v env =
        if v in dom(env) then env(v) else NS
infer_type (\x:s1..sn. body) env =
        /\[   s1 -> (infer_type_outer body (extend env v s1))
           .. sn -> (infer_type_outer body (extend env v sn)) ]
infer_type (e1 e2) env =
        infer_app_type (infer_type e1 env) (infer_type e2 env)
infer_type (e1 with i=e2) env  =
       /\[ remove_field i (infer_type e1 env),
           i : infer_type e2 env ]
infer_type (e.i) env  =
       extract_field i (infer_type e env)


(* Find the minimal type of an application *)
infer_app_type (s1->s2) s1' =
        if (s1' <= s1) then s2 else NS
infer_app_type /\[s1..sn] s1' =
        /\[infer_app_type s1 s1'  ..  infer_app_type sn s1']
infer_app_type \/[t1..t2] s1' =
        NS
infer_app_type i:t s1' =
        NS
infer_app_type p s1' =
        NS


(* Remove all occurences of field i from a type *)
remove_field i p             =  p
remove_field i s->t          =  s->t
remove_field i /\[t1..tn]    =  /\[remove_field t1 .. remove_field tn]
remove_field i \/[t1..tn]    =  \/[remove_field t1 .. remove_field tn]
remove_field i i:t           =  NS
remove_field i i':t          =  i':t
```

```
(* Select the field i from a type *)
extract_field i p            =  NS
extract_field i s->t         =  NS
extract_field i /\[t1..tn]   =  /\[extract_field t1 .. extract_field tn]
extract_field i \/[t1..tn]   =  \/[extract_field t1 .. extract_field tn]
extract_field i i:t          =  t
extract_field i i':t         =  NS


(* Compute the list of all lists whose first element is an element of
   [e11..e1m1], whose second argument is an element of [e21..e2m2],
   etc. *)
all_combinations [ [e11..e1m1] .. [en1..enmn] ]
        = [ [e11 e21 .. en1]
            [e11 e21 .. en2]
            ...
            [e11 e21 .. enmn]
            ...
            [e1m1 e2m2 .. enmn]
          ]
```

4.5 Efficiency of Type Normalization

Naively applying the rules for $\rightsquigarrow$ leads to very bad runtime performance. In particular, the two $\rightsquigarrow$-Absorb rules can involve a tremendous amount of subcomputation.

The following algorithm is my current best guess at how to efficiently normalize types:

```
nf(t) = sort_subphrases(semi_nf t)

semi_nf(p)          = p
semi_nf(t1->t2)     = semi_nf_outer(dist(semi_nf(t1),semi_nf(t2)))
semi_nf(/\[t1..tn]) = semi_nf_inter([t1..tn])
semi_nf(\/[t1..tn]) = semi_nf_union([t1..tn])

semi_nf_inter [t1..tn] =
        unnest_inter
              (absorb_inter
                    (flatten_inter [semi_nf t1 .. semi_nf tn]))

semi_nf_union [t1..tn] =
        unnest_union
              (absorb_union
                    (flatten_union [semi_nf t1 .. semi_nf tn]))

semi_nf_outer_inter [t1..tn] =
        unnest_inter
              (absorb_inter
                    (flatten_inter [semi_nf_outer t1 .. semi_nf_outer tn]))

semi_nf_outer_union [t1..tn] =
        unnest_union
              (absorb_union
                    (flatten_union [semi_nf_outer t1 .. semi_nf_outer tn]))

dist(s,/\[t1..tn])  = /\[dist(s,t1) .. dist(s,tn)]
dist(s,\/[t1..tn])  = \/[dist(s,t1) .. dist(s,tn)]
dist1(s,t)          = s->t

unnest_inter([t])      = t
```

```
unnest_inter([t1..tn]) = /\[t1..tn]    when n!=1

unnest_union([t])      = t
unnest_union([t1..tn]) = \/[t1..tn]    when n!=1

flatten_inter([/\[s1..sm],t2..tn) = [s1..sm] @ flatten_inter([t2..tn])
flatten_inter([t1,t2..tn)         = [t1]     @ flatten_inter([t2..tn])
                                    when t1 != /\[s1..sm]
flatten_inter([])             = []

flatten_union([\/[s1..sm],t2..tn) = [s1..sm] @ flatten_union([t2..tn])
flatten_union([t1,t2..tn)         = [t1]     @ flatten_union([t2..tn])
                                    when t1 != \/[s1..sm]
flatten_union([])             = []

absorb_inter([t1..tn]) = collect [] [t1..tn] []
            where  collect [s1..sm] []            [r1..rk] = [r1..rk]
                   collect [s1..sm] [ti,ti+1..tn] [r1..rk] =
                   if /\[s1..sm,ti+1..tn] <= ti
                      then collect [s1..sm]    [ti+1..tn] [r1..rk]
                      else collect [s1..sm,ti] [ti+1..tn] [r1..rk,ti]

absorb_union([t1..tn]) = collect [] [t1..tn] []
            where  collect [s1..sm] []            [r1..rk] = [r1..rk]
                   collect [s1..sm] [ti,ti+1..tn] [r1..rk] =
                   if ti <= \/[s1..sm,ti+1..tn]
                      then collect [s1..sm]    [ti+1..tn] [r1..rk]
                      else collect [s1..sm,ti] [ti+1..tn] [r1..rk,ti]
```

Conjecture 4.20: $\mathtt{nf}(\tau) = \tau'$ iff $\tau \rightsquigarrow^{\star} \tau'$ and τ' is in normal form.

Proof: [...] □

The formulation of normalization algorithm is based on the following observations (among others):

1. It is incorrect to apply the $\rightsquigarrow$-Dist-IA rules only once, before any of the other rules are allowed to fire. For example,
$$\mathtt{a}{\rightarrow}(\mathtt{ns}{\vee}\mathtt{ns}) \sim \mathtt{ns}$$
but this will not be discovered if the $\rightsquigarrow$-Dist-IA rules are not applied after the $\rightsquigarrow$-Absorb-Union rule.

2. The Sort rules can be applied last, since none of the other rules depend on the order in which element of unions and intersections are listed.

3. Even if $\tau_1..\tau_n$ are pairwise incomparable, there may still be a nonempty subset $\sigma_1..\sigma_m$ of $\tau_1..\tau_n$ such that $\bigwedge[\sigma_1..\sigma_m] \le \tau$. For example, let $\sigma_1 = \mathtt{a}$, $\sigma_2 = \mathtt{b}$, and $\tau = \mathtt{c} \vee (\mathtt{a}{\wedge}\mathtt{b})$. This motivates the form of rules $\rightsquigarrow$-Absorb-Inter and $\rightsquigarrow$-Absorb-Union.

5 Related Work

The idea of union types appears to have occurred to a number of people working independently in surprisingly disparate contexts. This section summarizes the related work of which I am aware and relates it to the present research.

N.b. My comments here are based only on informal conversations with the other researchers. Barbanera, Hayashi, and Fagan have written or are in the process of writing about their work but I have not yet read the written accounts.

5.1 Barbanera

Franco Barbanera [1], working in the context of Curry-style type assignment systems, has studied a formulation of union types quite similar to the one presented here. In particular, his rule for union elimination is identical to the rule UNION-E given on page 4. To prove subject reduction for his system, however, he found it necessary to use a different form of the rule,

$$\frac{\Gamma,\, \mathsf{e} : \sigma \;\vdash\; \mathsf{e}' \in \tau \qquad \Gamma \;\vdash\; \mathsf{e} \in \sigma}{\Gamma \;\vdash\; \mathsf{e}' \in \tau}$$

and prove the two systems equivalent. (I had also considered a rule of this form at one stage.) An earlier version of his system used the form

$$\frac{\Gamma,\, \mathtt{x} : \sigma_1 \;\vdash\; \mathsf{e} \in \tau \qquad \Gamma,\, \mathtt{x} : \sigma_2 \;\vdash\; \mathsf{e} \in \tau}{\Gamma,\, \mathtt{x} : \sigma_1 \vee \sigma_2 \;\vdash\; \mathsf{e} \in \tau}$$

but this went wrong for some reason.

The principal differences between Barbanera's formulation of union types and the one given here are:

1. Barbanera's axioms for the $\leq$ relation include the deMorgan laws for distributing intersections and unions.

2. His intepretation of $\sigma \vee \tau$ is $(\sigma \rightarrow \rho \sqcap \sigma \rightarrow \rho) \rightarrow \rho$, rather than something like a pushout as suggested here.

Barbanera's long-term interests in union types include investigating infinitary union types, along the lines suggested by Leivant's work on infinitary intersections [11].

5.2 Hayashi and Takeyama

The ATT type theory of Susumu Hayashi and Yukihide Takeyama [10] includes a formulation of union types in the context of an intuitionistic logic. Like Barbanera's, this system has a union elimination form exactly analogous to the rule UNION-E given here.

5.3 Cartwright and Fagan

Michael Fagan and Robert Cartwright [6] are developing an extension of the ML type system [9, 12] that includes both recursive types and a notion of "true union" of disjoint types. This is not quite the same idea as the union types described in this report, since for Fagan and Cartwright $\sigma \vee \tau$ is defined only when σ and τ have different outermost type constructors. But the motivating intuitions seem to be similar. Their system has principal types and a decidable type reconstruction problem.

5.4 Freeman

Tim Freeman [7] uses intersection types as refinements of ML types, to encode the results of abstract interpretations of ML programs. His prototype implementation gives a typing very close to the one in Section 2.3 for the `append` function. Union types have also played a role in the development of his algorithms.

6 Future Work

The preliminary research described in this report opens a number of intriguing paths for future work. The most important of these are:

Larger examples. The small examples discussed in Section 2 make a good case for some *potential* benefits of programming in a language with intersection and union types, but larger examples are needed to assess the *actual* practical benefits and understand how much they may cost in terms of programmer effort and compilation efficiency.

Semantics. A more comprehensive account of the semantics of the language is a prerequisite for making final decisions about details of the type system (e.g. the deMorgan laws). Since the primary motivation behind the development of the calculus is its application to practical programming languages, the first version should be expressed in an operational style. There is probably also useful insight to be gained by studying the structure of categorical models of union types, along the lines of Reynolds' semantics for Forsythe [16].

Proof theory. Once the syntax of the language is settled, a detailed study of its proof theory will be needed to develop and justify the algorithms used in an implementation.

Polymorphism. It is clear from the examples in Section 2 that types like those given to `if` and `caselist` are beginning to be quite cumbersome. It appears likely that the only way to make this kind of type system efficient enough in practice is to incorporate polymorphic types as well [13, 14].

7 Acknowledgements

I am grateful for productive discussions with Franco Barbanera, Manfred Droste, Tim Freeman, Robert Harper, Nevin Heintze, Susumu Hayashi, Frank Pfenning, Didier Rémy, and John Reynolds.

A Summary of Rules

A.1 Subtyping

$$\tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\sigma \leq \theta \qquad \theta \leq \tau}{\sigma \leq \tau} \qquad \text{(Sub-Trans)}$$

$$\frac{\tau_1 \leq \sigma_1 \qquad \sigma_2 \leq \tau_2}{\sigma_1 \rightarrow \sigma_2 \ \leq \ \tau_1 \rightarrow \tau_2} \qquad \text{(Sub-Arrow)}$$

$$\frac{\text{for all } i,\ \sigma \leq \tau_i}{\sigma \ \leq\ \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\bigwedge[\tau_1..\tau_n] \ \leq\ \tau_i \qquad \text{(Sub-Inter-LB)}$$

$$\frac{\text{for all } i,\ \sigma_i \leq \tau}{\bigvee[\sigma_1..\sigma_n] \ \leq\ \tau} \qquad \text{(Sub-Union-L)}$$

$$\tau_i \ \leq\ \bigvee[\tau_1..\tau_n] \qquad \text{(Sub-Union-UB)}$$

$$\bigwedge[\sigma{\rightarrow}\tau_1 \mathrel{..} \sigma{\rightarrow}\tau_n] \ \leq\ \sigma{\rightarrow}\bigwedge[\tau_1..\tau_n] \qquad \text{(Dist-AI-R)}$$

$$\bigvee[\sigma_1{\rightarrow}\tau \mathrel{..} \sigma_n{\rightarrow}\tau] \ \leq\ \bigwedge[\sigma_1..\sigma_n]{\rightarrow}\tau \qquad \text{(Dist-AU-L)}$$

A.2 Well-Typed Terms

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} \in \tau} \qquad \text{(Subsumption)}$$

$$\Gamma_1, \mathtt{x} : \tau, \Gamma_2 \ \vdash\ \mathtt{x} \in \tau \qquad \text{(Var)}$$

$$\frac{\Gamma, \mathtt{x} : \sigma_i \ \vdash\ \mathtt{e} \in \tau}{\Gamma, \vdash\ \lambda\mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e} \in \sigma_i{\rightarrow}\tau} \qquad \text{(Arrow-I)}$$

$$\frac{\Gamma \vdash \mathtt{f} \in \sigma{\rightarrow}\tau \qquad \Gamma \vdash \mathtt{e} \in \sigma}{\Gamma \vdash \mathtt{f\ e} \ \in\ \tau} \qquad \text{(Arrow-E)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash \mathtt{e} \in \tau_i}{\Gamma \vdash \mathtt{e} \in \bigwedge[\tau_1..\tau_n]} \qquad \text{(Inter-I)}$$

$$\frac{\Gamma \vdash \mathtt{e\ in}\ \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{e}' \ \in\ \tau} \qquad \text{(Union-E)}$$

A.3 Records

$$\frac{\sigma \leq \tau}{\iota : \sigma \ \leq\ \iota : \tau} \qquad \text{(Sub-Record)}$$

$$\iota : \sigma \wedge \iota : \tau \ \leq\ \iota : (\sigma{\wedge}\tau) \qquad \text{(Dist-RI)}$$

$$\iota : (\sigma{\vee}\tau) \ \leq\ \iota : \sigma \vee \iota : \tau \qquad \text{(Dist-RU)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma \qquad \Gamma \vdash \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{e\ with}\ \iota = \mathtt{e}' \ \in\ (\sigma\backslash\iota) \wedge \iota : \tau} \qquad \text{(Record-I)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \tau}{\Gamma \vdash \mathtt{e}.\iota \in \tau.\iota} \qquad \text{(Record-E)}$$

A.4 Equality

$$\frac{\Gamma \vdash \mathtt{e} \in \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e} \in \tau} \qquad (\textsc{Eq-Refl})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{e}' = \mathtt{e} \in \tau} \qquad (\textsc{Eq-Symm})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau \qquad \Gamma \vdash \mathtt{e}' = \mathtt{e}'' \in \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e}'' \in \tau} \qquad (\textsc{Eq-Trans})$$

$$\frac{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) \in \tau}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{f})(\mathtt{e}) = [\mathtt{e}/\mathtt{x}]\mathtt{f} \in \tau} \qquad (\textsc{Eq-Beta})$$

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma_i \rightarrow \tau \qquad \mathtt{x} \notin \mathrm{FV}(\mathtt{e})}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}\ \mathtt{x}) = \mathtt{e} \in \sigma_i \rightarrow \tau} \qquad (\textsc{Eq-Eta})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau} \qquad (\textsc{Eq-Subsumption})$$

$$\frac{\Gamma, \mathtt{x} : \sigma_i \vdash \mathtt{e} = \mathtt{e}' \in \tau}{\Gamma \vdash (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}) = (\lambda \mathtt{x}{:}\sigma_1..\sigma_n.\,\mathtt{e}') \in \sigma_i \rightarrow \tau} \qquad (\textsc{Eq-Arrow-I})$$

$$\frac{\Gamma \vdash \mathtt{f} = \mathtt{f}' \in \sigma \rightarrow \tau \qquad \Gamma \vdash \mathtt{e} = \mathtt{e}' \in \sigma}{\Gamma \vdash (\mathtt{f}\ \mathtt{e}) = (\mathtt{f}'\ \mathtt{e}') \in \tau} \qquad (\textsc{Eq-Arrow-E})$$

$$\frac{\text{for all } i,\ \Gamma \vdash \mathtt{e} = \mathtt{e}' \in \tau_i}{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \bigwedge[\tau_1..\tau_n]} \qquad (\textsc{Eq-Inter-I})$$

$$\frac{\Gamma \vdash \mathtt{e} = \mathtt{e}' \in \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{f} = \mathtt{f}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{f} = [\mathtt{e}'/\mathtt{x}]\mathtt{f}' \in \tau} \qquad (\textsc{Eq-Union-E})$$

References

[1] Franco Barbanera. Personal communication, June 1990.

[2] Kim Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantifcation. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 38–50, 1988.

[3] Felice Cardone and Mario Coppo. Two extensions of curry's type inference system. In Odifreddi, editor, *Logic For Computer Science*. Academic Press, 1990.

[4] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of basic functionality theory for λ-calculus. *Notre-Dame Journal of Formal Logic*, 21:685–693, 1980.

[5] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda calculus semantics. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560, New York, 1980. Academic Press.

[6] Michael Fagan. Personal communication, June 1990.

[7] Tim Freeman. Personal communication, May 1990.

[8] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. In *Fifth Annual Symposium on Logic in Computer Science (Philadelphia, PA)*, pages 346–354. IEEE Computer Society Press, June 1990.

[9] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[10] Susumu Hayashi. Personal communication, June 1990.

[11] Daniel Leivant. Discrete polymorphism. To appear in Proceedings of Lisp and Functional Programming '90, 1990.

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[13] Benjamin C. Pierce. Bounded quantification and intersection types. Thesis proposal, September 1989.

[14] Benjamin C. Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report CMU-CS-89-169, CMU, 1989.

[15] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[16] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.