Principles of Type Refinement
(OPLSS 2016)

Noam Zeilberger

June 29 – July 2, 2016

Contents

Chapter 1

Introduction

1.1 What is a type refinement system?

Since type systems are so useful, it would be nice if we could say exactly what they were, but actually, it's difficult to pin down the concept of "type system" with any mathematical precision while retaining all of the features that make them so useful in practice.[1] Not unrelatedly, the precise denotation of the phrase "typed programming language" has never been completely settled – for example, whether it include languages like Python which are dynamically typed, or pure lambda calculus, which is sometimes said to be "uni-typed". With those grains of salt, the following definition of "type refinement system" nonetheless conveys the right intuitions:

> A type refinement system is a type system built on top of a typed programming language, as an extra layer of typing.

Type refinement systems in this sense have become more popular in recent years, although the need for new theoretical tools for understanding them is not yet widely recognized. No doubt part of the reason for this is that in many instances it is difficult to tell the difference between a type system, a type refinement system, and a typed programming language, or even to argue that the difference is important. Nonetheless, the lectures which follow are offered on the premise that a careful study of these distinctions can be rewarding.

The phrase "type refinement" as I'm using it here comes from a research program initiated by Frank Pfenning (my thesis co-advisor) in the late 1980s, and pursued by a long line of his students.[2] A hallmark of that line of research was the consideration of a typed programming language (namely, ML) which already had a rich language of types with strong theoretical guarantees and

[1]A point Benjamin Pierce makes in the introduction of *Types and Programming Languages*.

[2]Recently, Frank made some remarks about the history of the term in response to a question by Michael Greenberg, see "A refinement type by any other name" (March 16, 2015), `http://www.weaselhat.com/2015/03/16/a-refinement-type-by-any-other-name/`.

tool support, making it all the more important that any type system layered on top be conservative in a strong sense. That is, not only should these type refinement systems keep the semantics of the underlying language unchanged, but ideally they should also retain the good theoretical and practical properties of the original type system, such as modular and effective type checking. These difficult constraints led to the rise of a number of useful general principles for the construction of type refinement systems.

Among the principles that emerged, a fundamental one was the idea that type refinement should be distinguished from (and in a sense comes logically prior to) the more familiar concept of "subtyping". Intuitively, the refinement relationship

$$R \sqsubset A \qquad \text{“}R \text{ refines } A\text{”}$$

asserts that R is a property (which may or may not hold) of As. For example,

$$pos, even, odd \sqsubset nat$$

asserts that being positive, even, or odd are different properties of natural numbers, while

$$red, tasty, banana \sqsubset fruit$$

asserts that being red, tasty, or a banana are different properties of fruits. On the other hand, a subtyping relationship

$$R_1 \leq_A R_2 \qquad \text{“}R_1 \text{ is a subtype of } R_2 \text{ (at } A\text{)”}$$

asserts that for all As, property R_1 implies property R_2. For example, the assertion $odd \leq_{nat} pos$ is valid since every odd natural number is positive. Although we might leave off the index from a subtyping judgment when we're relaxed, really it only makes sense to compare properties of the same kind of object (we would get funny looks if we asked if being even implies being a banana), and in that sense the refinement relation comes prior to subtyping.

A related insight that emerged was that this hierarchical view of typing helps clarify the old distinction in type theory between "types à la Church" and "types à la Curry" (also called the *intrinsic* and the *extrinsic* views of typing). Recall that in the Church (intrinsic) view, every well-formed term has a uniquely associated type, and there is no reasonable meaning to be assigned to untyped terms. In the Curry (extrinsic) view, on the other hand, types express properties of terms, and so a single term can have many different types (or no type at all), but it also has an underlying (computational) meaning which is independent of these types. A more practical aspect of this distinction is that determining the type of a term under the Church view is usually a trivial task, whereas the problem of deciding whether a term has a given type under the Curry view can have high computational complexity, or even be undecidable.

Viewed from the perspective of type refinement systems, the distinction between types à la Church and types à la Curry can be summarized in the slogan that "Curry refines Church":

$$\text{Curry} \sqsubset \text{Church}$$

In other words, what this view emphasizes is that Curry-style types should not be considered as properties of terms "in a vacuum", but rather as properties of terms constructed using some more primitive, intrinsic typing mechanism. In many historical instances (not to mention modern ones) of Curry-style typing these underlying Church-types may initially be difficult to spot (typically because they involve the solution to a recursive equation like $D \cong D \to D$), but being aware of this underlying refinement structure can be very useful in getting a better handle on the type systems of interest.

The first half of these lectures (Chapter 2) is an introduction to the fundamentals of type refinement systems organized around the "Curry $\sqsubset$ Church" slogan, which we'll take quite literally by using a series of refinements of Church's simply typed lambda calculus to navigate topics such as subtyping, intersection types, and bidirectional typing. The material and presentation are heavily-inspired by the aforementioned line of work by Frank Pfenning and collaborators (especially [33]), and more or less follows the standard proof-theoretic tradition of type theory in terms of overall methodology.

The second half of the lectures (Chapter 3) examines type refinement from a categorical perspective, based on the idea that a type refinement system should be seen as an *erasure functor* from typing derivations to terms. This leads to some interesting interactions with the theory of bifibrations, and also ties with other ideas in logic and computer science, such as separation logic. The material in this half of the lectures is based on joint work with Paul-André Melliès, with whom I have been collaborating on this subject for a number of years.

Chapter 2

Refining the simply typed lambda calculus

Despite its relatively limited expressive power, the simply typed lambda calculus $\lambda_{\to}$ plays an important role in programming languages and type theory, serving as the basis for more powerful extensions such as ML or System F. In this chapter, rather than extending the language, we will instead consider a series of *refinements* of the simply typed lambda calculus, as different type systems constructed over $\lambda_{\to}$. The main motivation is that certain general principles of type refinement systems can already be articulated in this basic setting, and we'll see that various interesting and illustrative phenomena arise.

2.1 Preliminaries on $\lambda_{\to}$

We assume that the reader is already familiar with the standard lambda calculus notions of free and bound variables, α-conversion, capture-avoiding substitution, β-reduction, and η-expansion. Here we recall some basic definitions for the simply typed lambda calculus, and establish some notation.

Definition 2.1.1 (Simple types)**.** Given a set $\mathcal{P}$ of *atomic types*, the **simple type hierarchy generated over** $\mathcal{P}$ is the least set $\mathcal{T}[\mathcal{P}]$ containing $\mathcal{P}$ and which is closed under the operation of building *function types*:

1. if $p \in \mathcal{P}$ then $p \in \mathcal{T}[\mathcal{P}]$. (atomic types)
2. if $A \in \mathcal{T}[\mathcal{P}]$ and $B \in \mathcal{T}[\mathcal{P}]$ then $A \to B \in \mathcal{T}[\mathcal{P}]$. (function types)

In order to define the simply typed terms, we suppose given a collection of variables $x, y, z, \ldots$, and moreover we suppose that each variable is uniquely associated with a simple type. Diverging a bit from Church's original presentation, we do not indicate the type of x explicitly (Church used subscript annotations x_A), instead viewing this information as determined implicitly by the context.

Definition 2.1.2 (Simply typed terms)**.** The set of **(simply typed) terms of type** $A \in \mathcal{T}[\mathcal{P}]$ is defined inductively by the following rules:

1. If x is a variable of type A then x is a term of type A. (variables)
2. If t is a term of type $A \to B$ and u is a term of type A then $t(u)$ is a term of type B. (applications)
3. If x is a variable of type A and t is a term of type B then $\lambda x.t$ is a term of type $A \to B$. (abstractions)

Notation. We write $x_1 : A_1, \dots, x_k : A_k \vdash t : B$ to indicate that t is a simply typed term of type t with free variables $x_1, \dots, x_k$ of types $A_1, \dots, A_k$. Here, the list $\Gamma = x_1 : A_1, \dots, x_k : A_k$ is called the **context** of t.

Sometimes we also write $t : A$, leaving the context Γ implicit.

In the following sections, we will take "$\lambda_\to$" to mean the simply typed lambda calculus generated over a *single* atomic type ι. For example,

$$\lambda x.x : \iota \to \iota$$
$$\lambda x.\lambda y.x : \iota \to \iota \to \iota$$
$$\lambda f.\lambda g.\lambda x.g(f(x)) : (\iota \to \iota) \to (\iota \to \iota) \to (\iota \to \iota)$$

are a few different terms of $\lambda_\to$.

2.2 Refinement, subtyping, and typing in $\lambda_\to^\leq$

2.2.1 Introducing $\lambda_\to^\leq$

For Church, the atomic type ι represented an abstract type of "individuals". (He probably got this from Bertrand Russell.) Our first type refinement system, which we call $\lambda_\to^\leq$, will be based on the simple idea of replacing ι by a set of atomic types $\mathcal{P}$, which we'd like to think of as giving more precise information about "what kind of individual". For example, we might take $\mathcal{P}$ as the following collection of properties:

$$\mathcal{P}_\mathbf{w} = \{\,\text{animal}, \text{mammal}, \text{marine}, \text{llama}, \text{dolphin}, \text{jellyfish}, \text{vegetable}, \text{carrot}\,\}$$

Moreover, we can express the fact that some of these properties imply others (every dolphin is a mammal, etc.) by asking that the set $\mathcal{P}_\mathbf{w}$ be equipped with a preorder $p \leq q$ $(p, q \in \mathcal{P})$, indicated here by a Hasse diagram:

$$\begin{tikzcd}
 & & \text{animal} \arrow[dl, dash] \arrow[dr, dash] & & \text{vegetable} \arrow[d, dash] \\
 & \text{mammal} \arrow[d, dash] \arrow[dr, dash] & & \text{marine} \arrow[dl, dash] \arrow[d, dash] & \text{carrot} \\
 & \text{llama} & \text{dolphin} & \text{jellyfish} &
\end{tikzcd}$$

In order to define the type refinement system $\lambda^{\leq}_{\rightarrow}$, we will assume given such an arbitrary preordered set $(\mathcal{P}, \leq)$ as a parameter. But before discussing the typing relation, we first introduce two simpler relations, respectively called *refinement* and *subtyping*.

Definition 2.2.1 (Refinement, $\lambda^{\leq}_{\rightarrow}$)**.** The **refinement relation** $R \sqsubset A$ ($R \in \mathcal{T}[\mathcal{P}]$, $A \in \mathcal{T}[\iota]$) (pronounced "R refines A") for $\lambda^{\leq}_{\rightarrow}$ is defined inductively by the following rules:

$$\frac{p \in \mathcal{P}}{p \sqsubset \iota} \qquad \frac{R \sqsubset A \quad S \sqsubset B}{R \rightarrow S \sqsubset A \rightarrow B}$$

Terminology. When $R \sqsubset A$ holds for some A, I will sometimes refer to R as a **type refinement**, or (by transposition) as a **refinement type**, or (for short) as a **refinement**. Types are types, though (whether they be in $\mathcal{T}[\mathcal{P}]$ or in $\mathcal{T}[\iota]$), and so I will also continue to refer to a type refinement as a **type**. Hopefully, this mix of terminology should not be too confusing.

Definition 2.2.2 (Subtyping, $\lambda^{\leq}_{\rightarrow}$)**.** The **subtyping relation** $R \leq_A S$ ($R, S \sqsubset A$) (pronounced "R is a subtype of S (at type A)") for $\lambda^{\leq}_{\rightarrow}$ is defined inductively by the following rules:

$$\frac{p \leq q}{p \leq_\iota q}\ \leq_\iota \qquad \frac{R_2 \leq_A R_1 \quad S_1 \leq_B S_2}{R_1 \rightarrow S_1 \leq_{A \rightarrow B} R_2 \rightarrow S_2}\ \leq_\rightarrow$$

Notation. We sometimes omit the subscript, writing $R \leq S$ instead of $R \leq_A S$ when A is clear from context or not relevant.

Notation. Most often we leave the choice of preordered set $(\mathcal{P}, \leq)$ as an implicit parameter of $\lambda^{\leq}_{\rightarrow}$, but if we want to make very clear that we are considering refinement, subtyping, etc., with respect to a given preorder $\mathcal{P} = (\mathcal{P}, \leq)$, we indicate it by a superscript, writing $R \sqsubset^{\mathcal{P}} A$, $R \leq^{\mathcal{P}} S$, etc.

The difference between the refinement and subtyping relations might at first be difficult to fully understand, but let me emphasize that the difference is really important! Formally, in the definitions above, refinement relates a type in $\mathcal{T}[\mathcal{P}]$ to a type in $\mathcal{T}[\iota]$, while subtyping relates two types in $\mathcal{T}[\mathcal{P}]$ which both refine the same type in $\mathcal{T}[\iota]$. Moreover, we have the following properties:

Proposition 2.2.3 ($\sqsubset$ functional)**.** *Refinement is a functional relation, i.e., for every $R \in \mathcal{T}[\mathcal{P}]$, there exists exactly one $A \in \mathcal{T}[\iota]$ such that $R \sqsubset A$.*

Proposition 2.2.4 ($\leq$ preorder)**.** *Subtyping is a preorder, i.e., it is reflexive ($R \leq_A R$ for all $R \sqsubset A$) and transitive (if $R \leq_A S$ and $S \leq_A T$ then $R \leq_A T$ for all $R, S, T \sqsubset A$).*

The differences between refinement and subtyping should hopefully become more clear as we progress. One of the reasons why they are sometimes confused is that if one thinks of types as sets, both the "is a refinement of" relation and the "is a subtype of" relation look superficially similar to the "is a subset of" relation. Prop. 2.2.3 gives us a hint that the refinement relation $\sqsubset$ behaves a bit differently from the usual subset relation $\subseteq$ of set theory, though, and in Section 2.2.3, we will make this precise by giving a set-theoretic semantics to $\lambda_{\to}^{\leq}$.

Example 2.2.5. Take $(\mathcal{P}_{\mathbf{w}}, \preceq)$ as introduced above, and consider the simple type

$$A = (\iota \to \iota) \to \iota.$$

It can be refined in many different ways, such as:

$$\begin{aligned} R &= (\text{mammal} \to \text{mammal}) \to \text{mammal} \\ S &= (\text{dolphin} \to \text{animal}) \to \text{animal} \\ T &= (\text{animal} \to \text{dolphin}) \to \text{animal} \end{aligned}$$

Among these three refinements $R, S, T \sqsubset A$, the subtyping relationships $R \leq T$ and $S \leq T$ hold, but neither $R \leq S$ nor $S \leq R$.

$$\begin{tikzcd} & T \arrow[dl, dash] \arrow[dr, dash] & \\ R & & S \end{tikzcd}$$

$\square$

We now turn to typing, which we will express in the concise and familiar notation of inference rules.

Definition 2.2.6 (Context refinement)**.** Let Γ and Π be two contexts mentioning the same collection of variables. We say that Π is a **context refinement** of Γ (written $\Pi \sqsubset \Gamma$) if the types of the variables in Π refine the types of the variables in Γ, in other words, if $\Gamma = x_1 : A_1, \ldots, x_k : A_k$ and $\Pi = x_1 : R_1, \ldots, x_k : R_k$ for some $R_1 \sqsubset A_1, \ldots, R_k \sqsubset A_k$.

Definition 2.2.7 (Typing, $\lambda_{\to}^{\leq}$)**.** The **typing relation** $\Pi \vdash t : R$ ($\Pi \sqsubset \Gamma$ and $\Gamma \vdash t : A$ and $R \sqsubset A$) for $\lambda_{\to}^{\leq}$ is defined inductively by the following rules:

$$\frac{x : R \in \Pi}{\Pi \vdash x : R}\ \mathit{var} \qquad \frac{\Pi \vdash t : R \to S \quad \Pi \vdash u : R}{\Pi \vdash t(u) : S}\ \mathit{app} \qquad \frac{\Pi, x : R \vdash t : S}{\Pi \vdash \lambda x.t : R \to S}\ \mathit{abs}$$

$$\frac{\Pi \vdash t : R \quad R \leq S}{\Pi \vdash t : S}\ \mathit{sub}_{\leq}$$

Clarification. Our convention is to not state explicitly all of the various conditions which are implicitly required for the judgments in the above rules to be well-formed. For example, it is implicit in the $\mathit{sub}_{\leq}$ rule that $\Pi \sqsubset \Gamma$ and $R, S \sqsubset A$ for the uniquely determined context Γ and type A such that $\Gamma \vdash t : A$.

Remark. The typing rules *var*, *app*, and *abs* look suspiciously like rules (1)–(3) of Defn. 2.1.2, especially if we translate the latter back from the prose to their more usual modern presentation as inference rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (1) \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B}\ (2) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ (3)$$

These two sets of rules have a quite different interpretation, though. Whereas previously we defined the language of simply typed terms indexed together with their types (in the original style of Church), here we are not extending the language but rather taking the terms of $\lambda_{\to}$ as given, and explaining how to ascribe them additional, more refined types. This distinction becomes sharper if we look through the lens of the Curry-Howard correspondence, where a simply typed term $\Gamma \vdash t : A$ corresponds to a proof π_t of A from assumptions Γ. Viewed in that light, rules (1)–(3) of Defn. 2.1.2 really would be better translated as the following standard rules of natural deduction:

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ hyp \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ {\to}E \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}I$$

In particular, these rules do not mention anything about terms...which makes sense, since via Curry-Howard, terms of $\lambda_{\to}$ correspond to the proofs themselves! On the other hand, if we have in our hands a proof π_t of $\Gamma \vdash A$, corresponding to a simply typed term $\Gamma \vdash t : A$, then we can still ask about the *properties* of that particular proof. That's the role of the system we have just introduced: a derivation α of $\Pi \vdash t : R$ in $\lambda^{\leq}_{\to}$ can be seen as a proof π_α in a slightly more powerful logic, saying something *about* the original proof π_t. □

Example 2.2.8. Consider the sequential composition combinator

$$seq = \lambda f.\lambda g.\lambda x.g(f(x)) : (\iota \to \iota) \to (\iota \to \iota) \to (\iota \to \iota).$$

Using the typing and subtyping rules of $\lambda^{\leq}_{\to}$, we can, for instance, give it the following more precise type:

$$seq : (\text{animal} \to \text{llama}) \to (\text{mammal} \to \text{dolphin}) \to (\text{jellyfish} \to \text{dolphin})$$

This might be glossed as, "If I have a procedure for turning any animal into a llama, and a procedure for turning any mammal into a dolphin, then their sequential composition gives me a procedure which will turn any jellyfish into a dolphin, since every jellyfish is an animal, and every llama is a mammal." On the other hand, the following type ascription is invalid:

$$seq : (\text{llama} \to \text{llama}) \to (\text{vegetable} \to \text{vegetable}) \to (\text{llama} \to \text{vegetable})$$

Indeed, if we try to compose a llama endomorphism with a vegetable endomorphism, we shouldn't expect any promising results. □

Besides its *interpretation* as a type refinement system over $\lambda_{\rightarrow}$, the main novelty of $\lambda_{\rightarrow}^{\leq}$ relative to standard simply typed lambda calculus is the $sub_{\leq}$ rule, which is commonly called **subsumption**. The subsumption rule also has a contravariant form,

$$\frac{\Omega \leq \Pi \quad \Pi \vdash t : R}{\Omega \vdash t : R}\ sub_{\leq}^{op}$$

where the judgment $\Omega \leq \Pi$ is interpreted in the obvious way for two context refinements $\Omega, \Pi \sqsubset \Gamma$, namely, that the type assigned to each variable in Ω is a subtype of the type assigned to the same variable in Π. Although I did not include $sub_{\leq}^{op}$ explicitly as part of Defn. 2.2.7, it is easily seen to be admissible.

Proposition 2.2.9 (Contravariant subsumption)**.** *$sub_{\leq}^{op}$ is admissible in $\lambda_{\rightarrow}^{\leq}$.*

Proof. By induction on typing derivations. □

You can warmup to the subsumption rule by working through the following sequence of exercises, where we'll restrict $(\mathcal{P}, \preceq)$ to be the right-hand side of the preorder introduced at the beginning of this section,

$$\begin{tikzcd}\text{vegetable} \arrow[d, dash] \\ \text{carrot}\end{tikzcd}$$

which if you prefer you can also treat (more blandly) as the two-element preorder $\mathcal{P}_2 = \bot \preceq \top$.

Exercise 2.2.10. Let $\mathbb{B} = \iota \rightarrow \iota \rightarrow \iota$ be the type of the Church booleans, where $true = \lambda x.\lambda y.x$ and $false = \lambda x.\lambda y.y$. Find all refinements $R \sqsubset \mathbb{B}$ such that

a) $\vdash true : R$ and $\vdash false : R$

b) $\vdash true : R$ but $\nvdash false : R$

c) $\vdash false : R$ but $\nvdash true : R$

d) $\nvdash true : R$ and $\nvdash false : R$

Exercise 2.2.11. Let $\mathbb{N} = (\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$ be the type of the Church numerals, where

$$\begin{aligned} c_0 &= \lambda f.\lambda x.x \\ c_1 &= \lambda f.\lambda x.f(x) \\ c_2 &= \lambda f.\lambda x.f(f(x)) \end{aligned}$$

How many different refinements $R \sqsubset \mathbb{N}$ can you assign each Church numeral c_i, for $0 \leq i \leq 2$? What about for $i > 2$?

Exercise 2.2.12. It is often the case in $\lambda^{\leq}_{\to}$ that one typing judgment can have multiple typing derivations. Indeed, if we allow unrestricted applications of the subsumption rule, then any derivable judgment has infinitely many derivations, since given a derivation α of $\Pi \vdash t : R$ we can always construct a new derivation

$$\alpha' = \cfrac{\overset{\alpha}{\Pi \vdash t : R} \quad R \leq R}{\Pi \vdash t : R}\ sub_{\leq}$$

where the second premise is discharged by reflexivity. For the purpose of this exercise, though, you may assume that there are no such no trivial uses of $sub_{\leq}$. Under that assumption, how many different derivations are there of...

a) $\vdash \lambda x.x : \text{carrot} \to \text{vegetable}$?

b) $\vdash \lambda x.\lambda y.x : \text{carrot} \to \text{vegetable} \to \text{vegetable}$?

c) $\vdash \lambda f.\lambda x.\lambda y.fyx : (\bot \to \top \to \bot) \to (\top \to \bot \to \bot)$?

Exercise 2.2.13. Let's say that a closed term $t : A$ has a *principal type* if there is a $R \sqsubset A$ such that $\vdash t : R$, and moreover such that for any other $S \sqsubset A$, if $\vdash t : S$ then $R \leq S$. Which of the terms from Exercise 2.2.12 have principal types in this sense? Would this change if we replaced the preorder of refinements $\mathcal{P}_2$ by a different preorder?

2.2.2 Subtyping and η-expansion

I'd like to take a moment to briefly discuss an important general phenomenon of type refinement systems that already shows up in $\lambda^{\leq}_{\to}$, and which will be useful later when we turn to intersection types. There's a sense in which the subsumption rule $sub_{\leq}$ can be seen as "hiding" implicit η-expansion. Suppose that we restrict the rule to only its atomic instances:

$$\frac{\Pi \vdash t : p \quad p \leq q}{\Pi \vdash t : q}\ sub_{\leq}$$

I'll refer to this restriction of $\lambda^{\leq}_{\to}$ as $\lambda^{\leq}_{\to\eta}$, motivated by the following:

Proposition 2.2.14 (HO subsumption = atomic subsumption + η-expansion). *The higher-order subsumption rule*

$$\frac{\Pi \vdash t : R \quad R \leq_A S}{\Pi \vdash \eta_A[t] : S}\ sub_{\leq\eta}$$

is derivable in $\lambda^{\leq}_{\to\eta}$, where $\eta_A[t]$ is the <u>*iterated η-expansion*</u> *of t defined by:*

$$\eta_\iota[t] = t$$
$$\eta_{A\to B}[t] = \lambda x.\eta_B[t(\eta_B[x])]$$

Proof. By induction on A. □

One immediate corollary is that any higher-order subtyping relation can be realized as *typing of an identity coercion*.

Proposition 2.2.15. *If $R \leq_A S$ then $x : R \vdash \eta_A[x] : S$ in $\lambda^{\leq}_{\to\eta}$.*

Indeed the converse of Prop. 2.2.15 is also true, as you will be asked to prove in Exercise 2.2.38. So in that sense subtyping can be seen as a "derived" notion, less primitive than typing, even though we chose to introduce the subtyping relation first when we originally defined $\lambda^{\leq}_{\to}$.

2.2.3 The subset interpretation

This section is meant to provide some additional intuition for the meaning of the refinement, subtyping, and typing relations, by explaining how to interpret them in a simple class of concrete models, where refinements are interpreted as subsets, and subtyping by inclusion. Let's begin by recalling the standard set-theoretic interpretation of $\lambda_{\to}$, where types are interpreted as sets and terms as functions. The interpretation of types is given inductively by

$$\begin{aligned} [\![\iota]\!] &= D \\ [\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \end{aligned}$$

where D is some fixed set given as a parameter of the model, and where $[\![A]\!] \to [\![B]\!] = [\![B]\!]^{[\![A]\!]}$ denotes the set of all functions from $[\![A]\!]$ to $[\![B]\!]$. Contexts are interpreted as cartesian products of sets,

$$\begin{aligned} [\![\cdot]\!] &= 1 \\ [\![\Gamma, x : A]\!] &= [\![\Gamma]\!] \times [\![A]\!] \end{aligned}$$

and every term $\Gamma \vdash t : A$ is interpreted as a function

$$[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$$

defined by induction on t:

$$\begin{aligned} [\![x_i]\!]\,\gamma &= a_i \quad \text{where} \quad \Gamma = (x_1, \dots, x_k), \gamma = (a_1, \dots, a_k) \\ [\![(\lambda x.t)]\!]\,\gamma &= a \mapsto [\![t]\!]\,(\gamma, a) \\ [\![t(u)]\!]\,\gamma &= ([\![t]\!]\,\gamma)([\![u]\!]\,\gamma) \end{aligned}$$

This standard set-theoretic interpretation of the simply typed lambda calculus may be extended to an interpretation of the type refinement system $\lambda^{\leq}_{\to}$ as follows. First, we assume given an *order-preserving function*

$$i[-] : \mathcal{P} \to 2^D$$

from the preordered set of atoms to the lattice of subsets of D, in the sense that each atom $p \in \mathcal{P}$ is assigned a subset $i[p] \subseteq D$ such that $p \leq q$ implies $i[p] \subseteq i[q]$ for all $p, q \in \mathcal{P}$. This interpretation of the atoms may then be extended to an interpretation of arbitrary refinements $R \sqsubset A$ as subsets $[\![R]\!] \subseteq [\![A]\!]$, which we define (by their characteristic functions) as follows:

$$d \in [\![p]\!] \Leftrightarrow d \in i[p]$$
$$f \in [\![R \to S]\!] \Leftrightarrow \forall a.\, a \in R \text{ implies } f(a) \in S$$

Similarly, the refinement $\Pi \sqsubset \Gamma$ of a context can be interpreted as a subset $[\![\Pi]\!] \subseteq [\![\Gamma]\!]$, defined by a conjunction of constraints on the components of the cartesian product:

$$* \in [\![\cdot]\!] \Leftrightarrow \text{true}$$
$$(\gamma, a) \in [\![\Pi, x : R]\!] \Leftrightarrow \gamma \in [\![\Pi]\!] \text{ and } a \in [\![R]\!]$$

We then have the following easy result:

Proposition 2.2.16 (Soundness of subset interpretation)**.** *(i) if $R \leq_A S$ then $[\![R]\!] \subseteq [\![S]\!]$ (i.e., for all $a : [\![A]\!]$, if $a \in [\![R]\!]$ then $a \in [\![S]\!]$), and (ii) if $\Pi \vdash t : R$ then $[\![t]\!]([\![\Pi]\!]) \subseteq [\![R]\!]$ (i.e., for all $\gamma : [\![\Gamma]\!]$, if $\gamma \in [\![\Pi]\!]$ then $[\![t]\!](\gamma) \in [\![R]\!]$).*

Proof. By induction on subtyping and typing derivations. □

Now, if you have some background in category theory, you probably know that the soundness of the standard set-theoretic interpretation of $\lambda_{\rightarrow}$ can be said to rely on the fact that **Set**, the category of sets and functions, is cartesian closed. So, you might be wondering if the subset interpretation and Prop. 2.2.16 may likewise be justified by some similarly slick categorical gadget. In fact, subsets can be arranged into another category **Subset** whose objects are *pairs* (A, R) of a set equipped with a subset of that set ($R \subseteq A$), and whose morphisms

$$(A, R) \longrightarrow (B, S)$$

are functions $f : A \to B$ sending elements of R to elements of S. Then what is crucial is that this category **Subset** is cartesian closed, and moreover that the forgetful functor **Subset** $\to$ **Set** defined by projection

$$(A, R) \mapsto A$$

strictly preserves this cartesian closed structure. We will return to this idea and explore it much more deeply in Chapter 3.

Finally, in some situations it might be argued that it is better to interpret types as partially-ordered sets (posets) and lambda terms as order-preserving functions. The subset interpretation of $\lambda^{\leq}_{\rightarrow}$ can be extended to this situation, but now rather than denoting arbitrary subsets, a refinement $R \sqsubset A$ is interpreted as a *downwards-closed* subset of $[\![A]\!]$, in the sense that

$$\text{if } a' \in [\![R]\!] \text{ and } a \leq a' \text{ then } a \in [\![R]\!]$$

for all $a, a' : [\![A]\!]$. As with the more basic subset interpretation, the order-theoretic subset interpretation can be justified on abstract categorical grounds, relying on the fact that downwards closed subsets of posets can be arranged into a cartesian closed category **Downset**, together with a strict cartesian closed functor **Downset** $\to$ **Poset** projecting back to the category of posets and order-preserving functions.

2.2.4 Principal types, type schemes, and bidirectional typing

So far we have studied $\lambda^{\leq}_{\to}$ as a "type assignment" system, in the sense that in Section 2.2.1 we gave inductive definitions of the subtyping and typing relations, and in Section 2.2.3 we showed that these relations admit a natural interpretation wherein simple types denote sets and refinements denote subsets of those sets. What we have not done yet, however, is give an explicit algorithm for deciding whether the subtyping and/or typing relations hold in any given instance. In fact, the way we defined subtyping, it's pretty easy to see it is decidable:

Proposition 2.2.17 ($\leq$ decidable). *The subtyping relation $R \leq_A S$ of $\lambda^{\leq}_{\to}$ is decidable, provided the underlying preorder $p \leq q$ is decidable.*

Proof. We greedily attempt to construct a derivation of $R \leq_A S$ bottom-up, by induction on A. In each case, only one possible rule applies (either $\leq_\iota$ or $\leq_\to$), so we apply the appropriate rule until we are left with only premises of the form $p \leq q$, which we can decide by the assumption. □

No such simple argument works for deciding the typing relation, though. You already got a hint of this if you did Exercise 2.2.12, which explored the fact that in $\lambda^{\leq}_{\to}$ a typing judgment can often have more than one derivation. Besides the problem that the subsumption rule can be applied at any time, what really gets in the way of a naive bottom-up interpretation of the typing rules is the form of *app* and *sub*$_\leq$,

$$\frac{\Pi \vdash t : R \to S \quad \Pi \vdash u : R}{\Pi \vdash t(u) : S}\ app \qquad \frac{\Pi \vdash t : R \quad R \leq S}{\Pi \vdash t : S}\ sub_\leq$$

where the type checking algorithm seemingly has to pull the type R out of thin air. For example, it happens to be the case that the typing judgment

$$\vdash (\lambda f.\lambda x.\lambda y.fyx)(\lambda z.\lambda w.z) : \top \to \bot \to \top \tag{2.1}$$

is valid (where $\bot \leq \top$), but how would a type checking algorithm figure out that it should first verify

$$\vdash \lambda f.\lambda x.\lambda y.fyx : (\bot \to \top \to \bot) \to (\top \to \bot \to \top) \tag{2.2}$$

in order to verify (2.1)?

If you have done any programming in languages like ML or Haskell, then you know that it is sometimes possible to answer such questions through *type inference*, which attempts to compute the *principal type* of a term. Exercise 2.2.13 above recalled the usual definition of what it means to be a principal type for a closed term, and more generally one can speak of principal types with respect to a given typing context.

Definition 2.2.18 (Π-principal types). Let $\Gamma \vdash t : A$ be a term and $\Pi \sqsubset \Gamma$ a refinement of its context. A **principal type under** Π (or Π**-principal type**) of t, when it exists, is a refinement $R \sqsubset A$ such that $\Pi \vdash t : R$, and moreover such that for any other $S \sqsubset A$, if $\Pi \vdash t : S$ then $R \leq S$.

As suggested by Exercise 2.2.13, principal types in this sense do not always exist in $\lambda^{\leq}_{\to}$.

Counterexample 2.2.19. Let $(\mathcal{P}_{\mathrm{w}}, \leq)$ be the preorder fixed at the beginning of Section 2.2.1. Then the identity term $\lambda x.x : \iota \to \iota$ does not have a principal type (with respect to any context), since it can be assigned both types

$$\vdash \lambda x.x : \text{animal} \to \text{animal} \quad \text{and} \quad \vdash \lambda x.x : \text{vegetable} \to \text{vegetable},$$

neither of which is a subtype of the other. □

Although the lack of principal types might seem like a strong barrier to devising a type inference algorithm for $\lambda^{\leq}_{\to}$, it's actually a bit of a red herring, and there at least two different ways of addressing it.

Response 1: Principal type schemes.

One possible response is to say that rather than looking for principal types in $\lambda^{\leq}_{\to}$, we should really be looking for *principal type schemes*. Indeed, the well-known Hindley-Milner algorithm (based on unification) can be seen as computing principal type schemes for simply typed lambda terms without the $sub_{\leq}$ rule. Something similar will work for $\lambda^{\leq}_{\to}$, though we should first make precise exactly what we mean by a principal type scheme.

Definition 2.2.20 (Atomic type substitutions)**.** Let $\mathcal{P} = (\mathcal{P}, \leq^{\mathcal{P}})$ and $\mathcal{Q} = (\mathcal{Q}, \leq^{\mathcal{Q}})$ be preordered sets. An **atomic type substitution** from $\mathcal{P}$ to $\mathcal{Q}$ is an order-preserving function $\sigma : \mathcal{P} \to \mathcal{Q}$.

Notation. If $\sigma : \mathcal{P} \to \mathcal{Q}$ is an atomic type substitution, we write $[\sigma]$ for the homomorphic extension of σ to type refinements defined by

$$[\sigma]p = \sigma(p) \qquad [\sigma](R \to S) = [\sigma]R \to [\sigma]S$$

as well as for its homomorphic extension to context refinements defined by

$$[\sigma]\cdot = \cdot \qquad [\sigma](\Gamma, x : R) = [\sigma]\Gamma, x : [\sigma]R$$

Proposition 2.2.21 (Preservation of typing under type substitution)**.** *If* $\Pi \vdash^{\mathcal{P}} t : R$*, and* $\sigma : \mathcal{P} \to \mathcal{Q}$ *is an atomic type substitution, then* $[\sigma]\Pi \vdash^{\mathcal{Q}} t : [\sigma]R$.

Proof. By induction on typing derivations. □

Definition 2.2.22 (Principal type schemes)**.** Let $\Gamma \vdash t : A$ be a term. A **type scheme** for t is a triple of a preordered set $\mathcal{P} = (\mathcal{P}, \leq)$, a context refinement $\Pi \sqsubset^{\mathcal{P}} \Gamma$ and a type refinement $R \sqsubset^{\mathcal{P}} A$, such that $\Pi \vdash^{\mathcal{P}} t : R$. It is said to be a **principal type scheme** if for any other type scheme $((\mathcal{Q}, \leq), \Omega, S)$ for t, there exists an atomic type substitution $\sigma : \mathcal{P} \to \mathcal{Q}$ such that $\Omega \leq^{\mathcal{Q}} [\sigma]\Pi$ and $[\sigma]R \leq^{\mathcal{Q}} S$.

Although some terms may fail to have principal types in the sense of Defn. 2.2.18, it turns out that all terms of $\lambda_\to$ have principal type schemes in the sense of Defn. 2.2.22. For example,

$$\vdash \lambda x.x : p \to q \quad [p \preceq q] \tag{2.3}$$

is a principal type scheme for the identity term $\lambda x.x : \iota \to \iota$, while

$$\vdash \lambda f.\lambda x.\lambda y.fyx : (p \to q \to r) \to (s \to t \to u) \quad [s \preceq q, t \preceq p, r \preceq u] \tag{2.4}$$

is a principal type scheme for $\lambda f.\lambda x.\lambda y.fyx : (\iota \to \iota \to \iota) \to (\iota \to \iota \to \iota)$. Once we have principal type schemes, it is pretty easy to decide typing. For example, to check (2.2) using (2.4), it suffices to perform the substitution

$$p, r, t \mapsto \bot \quad q, s, u \mapsto \top$$

and verify the three inequalities $\top \preceq \top, \bot \preceq \bot, \bot \preceq \top$.

Exercise 2.2.23. Find principal type schemes for each of the Church numerals (cf. Exercise 2.2.11).

Exercise 2.2.24. Show that every term $\Gamma \vdash t : A$ has a type scheme where $\mathcal{P} = \{\iota\}$ is the one-element set. Is this a principal type scheme? □

An algorithm computing principal type schemes for a system more or less equivalent to $\lambda^{\preceq}_{\to}$ was given by John Mitchell [31] (although there are small technical differences between our definitions and his). Mitchell's algorithm ("Algorithm GA") is very similar in spirit to the Hindley-Milner algorithm, and you are asked to reconstruct it in the next two exercises.

Exercise 2.2.25. In the programming language of your choice, write a procedure which inputs a simply typed lambda term $\Gamma \vdash t : A$, and returns a $\lambda^{\preceq}_{\to}$ principal type scheme for t, in the form of a triple of a preorder $\mathcal{P} = (\mathcal{P}, \preceq^{\mathcal{P}})$, a context refinement $\Pi \sqsubset^{\mathcal{P}} \Gamma$, and a type refinement $R \sqsubset^{\mathcal{P}} A$ such that $\Pi \vdash^{\mathcal{P}} t : R$.

Exercise 2.2.26. Like the usual presentation of the Hindley-Milner algorithm, Mitchell's Algorithm GA was originally expressed as a type inference procedure on untyped terms t, which might fail in the case that t does not have a simple type. Modify your solution from Exercise 2.2.25 to work on untyped terms, then figure out the right way of adapting Definitions 2.2.20 and 2.2.22 so that your modified algorithm has the correct *specification*.

Response 2: Bidirectional typing.

Another possible response to the apparent difficulty we noticed before is to say that maybe instead of being worried that not every term has a principal type in $\lambda^{\preceq}_{\to}$, we should just pay closer attention to the ones that *do* have principal types. This gives rise to the idea of *bidirectional typing* (also called "local type inference"), which combines type inference on a select class of terms with type checking for the rest. As a general purpose technique, bidirectional typing has many practical advantages over computing principal type schemes, especially as one considers richer type systems. Moreover, it happens to have deep links to the underlying logical structure and symmetry of the lambda calculus, making it interesting from a purely theoretical perspective as well.

In order to explain bidirectional typing, let us first recall the classical lambda calculus notions of *normal* and *neutral* terms.

Definition 2.2.27 (Neutral and normal terms)**.** Being **neutral** or **normal** are properties of a lambda term, defined by mutual induction as follows:

1. If x is a variable then x is neutral.
2. If t is neutral and u is normal then $t(u)$ is neutral.
3. If t is neutral then t is normal.
4. If x is a variable and t is normal then $\lambda x.t$ is normal.

Proposition 2.2.28 (β-normal terms are normal)**.** *A term t is normal if and only if it does not contain a β-redex $(\lambda x.t')(u)$ as a subterm.*

The lucky observation which motivates bidirectional typing is that there is a very simple procedure for checking whether a normal term has a given type, by inferring principal types for its neutral subterms.

Notation. We use the letters e and m to range over neutral and normal terms, respectively. More compactly, the definition of neutral and normal terms can be expressed by the following grammar:

$$\begin{aligned} e &::= x \mid e(m) \\ m &::= e \mid \lambda x.m \end{aligned}$$

Definition 2.2.29 (Bidirectional typing, $\lambda^{\leq}_{\to}$, normal fragment)**.** Bidirectional typing for (the normal fragment of) $\lambda^{\leq}_{\to}$ consists of a **checking relation** $\Pi \vdash m \Leftarrow R$ together with a **synthesis relation** $\Pi \vdash e \Rightarrow R$, which are defined inductively by the following rules:

$$\frac{x : R \in \Pi}{\Pi \vdash x \Rightarrow R}\ var^{\Rightarrow} \qquad \frac{\Pi \vdash e \Rightarrow R \to S \quad \Pi \vdash m \Leftarrow R}{\Pi \vdash e(m) \Rightarrow S}\ app^{\Rightarrow}$$

$$\frac{\Pi \vdash e \Rightarrow R \quad R \leq S}{\Pi \vdash e \Leftarrow S}\ sub^{\Leftarrow}_{\leq} \qquad \frac{\Pi, x : R \vdash m \Leftarrow S}{\Pi \vdash \lambda x.m \Leftarrow R \to S}\ abs^{\Leftarrow}$$

What is remarkable about the four rules in Defn. 2.2.29 is that they are almost identical to the four rules in Defn. 2.2.7, yet they have an immediate algorithmic interpretation. In the terminology of logic programming, the definitions of the synthesis and checking relations correspond to *well-moded logic programs*:

$$\overset{\text{input}}{\Pi} \ \vdash \ \overset{\text{input}}{e} \ \Rightarrow \ \underset{\text{output}}{R} \qquad\qquad \overset{\text{input}}{\Pi} \ \vdash \ \overset{\text{input}}{m} \ \Leftarrow \ \overset{\text{input}}{R}$$

That is, the synthesis relation can be realized as a procedure which takes a context refinement and a neutral term as input and attempts to compute a type refinement as output, while checking can be realized as a procedure which takes a triple of a context refinement, a normal term, and a type refinement as input, and either succeeds or fails. In particular, the problem we ran into before with the *app* and $sub_{\leq}$ rules disappears for their bidirectional versions $app^{\Rightarrow}$ and $sub_{\leq}^{\Leftarrow}$, since the synthesis relation uniquely determines the refinement which we had to somehow mysteriously "guess" before.

Proposition 2.2.30 ($\Rightarrow$ partial functional)**.** *For any $\Pi \sqsubset \Gamma$ and $\Gamma \vdash e : A$, there exists at most one $R \sqsubset A$ such that $\Pi \vdash e \Rightarrow R$, and moreover R must already occur as a subformula*[1] *in Π.*

Proposition 2.2.31 ($\Leftarrow$ decidable)**.** *For any $\Pi \sqsubset \Gamma$ and $\Gamma \vdash m : A$ and $R \sqsubset A$, it is decidable whether $\Pi \vdash m \Leftarrow R$.*

The rules of bidirectional typing are obviously sound for the typing relation (we just replace "$\Rightarrow$" and "$\Leftarrow$" by ":" to obtain the rules in Defn. 2.2.7), but what makes them interesting is that they are also *complete* for the typing relation, at least as restricted to β-normal terms:

Theorem 2.2.32 (Completeness of bidirectional typing, normal fragment)**.** *Let $\Gamma \vdash t : A$, and suppose $\Pi \vdash t : R$ for some $\Pi \sqsubset \Gamma$ and $R \sqsubset A$.*

a) If t is neutral, then there exists an $R_0 \sqsubset A$ such that $\Pi \vdash t \Rightarrow R_0$ and $R_0 \leq R$.

b) If t is normal, then $\Pi \vdash t \Leftarrow R$.

To prove completeness, we rely on a lemma establishing covariant and contravariant subsumption principles for the bidirectional system.

Lemma 2.2.33 (Subsumption, bidirectional version)**.** *(Subsumption for synthesis:) If $\Omega \leq \Pi$ and $\Pi \vdash e \Rightarrow R$, then $\Omega \vdash e \Rightarrow R_0$ for some $R_0 \leq R$. (Subsumption for checking:) If $\Omega \leq \Pi$ and $\Pi \vdash m \Leftarrow R$ and $R \leq S$, then $\Omega \vdash m \Leftarrow S$.*

Proof. Straightforward by induction on derivations. □

Proof of Thm. 2.2.32. By induction on the derivation of $\Pi \vdash t : R$, with the two statements ordered so we can appeal to (a) from (b) with the same derivation. For (a) we have the following cases:

Case $\dfrac{x : R \in \Pi}{\Pi \vdash x : R}\,var$**:** Take $R_0 = R$, then $\Pi \vdash x \Rightarrow R$ by the $var^{\Rightarrow}$ rule and $R \leq R$ by reflexivity of subtyping.

[1] By "occur as a subformula" I just mean that R must already appear syntactically somewhere in Π. This relation can be defined inductively, beginning with the subformula relation $R \lhd S$ between types, which satisfies $R \lhd R$ for all R, and $R \lhd S_1 \to S_2$ iff $R \lhd S_1$ or $R \lhd S_2$.

Case $\dfrac{\Pi \vdash t' : R \to S \quad \Pi \vdash u : R}{\Pi \vdash t'(u) : S}\, app$**:** By assumption, t' is neutral and u is normal. By the induction hypothesis we obtain $\Pi \vdash t' \Rightarrow R' \to S'$ for some $R' \to S' \leq R \to S$, as well as $\Pi \vdash u \Leftarrow R$. By inversion of the subtyping relation, we have $R \leq R'$ and $S' \leq S$, hence $\Pi \vdash u \Leftarrow R'$ by Lemma 2.2.33, and $\Pi \vdash t'(u) \Rightarrow S'$ by the $app^{\Rightarrow}$ rule.

Case $\dfrac{\Pi \vdash t : R \quad R \leq R'}{\Pi \vdash t : R'}\, sub_{\leq}$**:** Immediate by the i.h. and transitivity of $\leq$.

Case (*lam*)**:** Impossible by assumption that t is neutral.

For (b) we have the following cases:

Case t **neutral:** Immediate by i.h. (a) and application of the $sub^{\Leftarrow}_{\leq}$ rule.

Case $\dfrac{\Pi, x : R \vdash t : S}{\Pi \vdash \lambda x.t' : R \to S}\, lam$**:** Immediate by i.h. (b) and application of $lam^{\Leftarrow}$.

Case $\dfrac{\Pi \vdash t : R \quad R \leq R'}{\Pi \vdash t : R'}\, sub_{\leq}$ **with** t **not neutral:** Since t is normal, it is necessarily of the form $t = \lambda x.t'$, while R and R' are of the form $R = R_1 \to S_1$, $R' = R_2 \to S_2$. By the i.h., $\Pi \vdash \lambda x.t' \Leftarrow R_1 \to S_1$, which implies $\Pi, x : R_1 \vdash t' \Leftarrow S_1$ by inversion of the checking relation. Since $R_2 \leq R_1$ and $S_1 \leq S_2$, this implies $\Pi, x : R_2 \vdash t' \Leftarrow S_2$ by Lemma 2.2.33, hence $\Pi \vdash \lambda x.t' \Leftarrow R_2 \to S_2$ by $lam^{\Leftarrow}$.

□

Corollary 2.2.34. *If $\Pi \vdash e \Rightarrow R$, then R is a Π-principal type for e.*

Since bidirectional typing seems to give such a simple answer to the question of typing β-normal terms, an obvious followup question is, "What about non-normal terms?" The short answer is that every β-redex should be accompanied by a type annotation.

Definition 2.2.35 (Bidirectional typing, $\lambda^{\leq}_{\rightarrow}$, annotations)**.** A **(type refinement) annotation** is a pair of a term $m : A$ together with a type refinement $R \sqsubset A$, written "$m : R$". Annotations are typed by the rule of **annotation synthesis**,

$$\frac{\Pi \vdash m \Leftarrow R}{\Pi \vdash (m : R) \Rightarrow R}\, ann^{\Rightarrow}$$

which says that $m : R$ synthesizes type R just in case m checks against R.

The completeness theorem can now be extended to arbitrary terms, at least in the more limited sense that it is always possible to place enough annotations on a term so that it checks.

Proposition 2.2.36 (Annotatability)**.** *If $\Pi \vdash t : R$, then there exists an annotated term m such that $|m| = t$ and $\Pi \vdash m \Leftarrow R$, where $|m|$ erases all annotations from m.*

Note we also need a minor adjustment to Prop. 2.2.30: whenever $\Pi \vdash e \Rightarrow R$, the type R must *either* occur as a subformula in Π or else as a subformula in (an annotation occurring in) e.

One way to think of type annotations is as a sort of virtual coercion from normal terms to neutral terms,

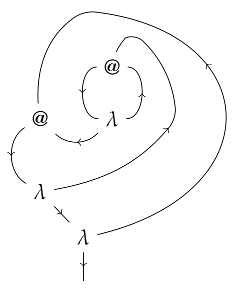
$$e ::= \cdots \mid (m : R)$$

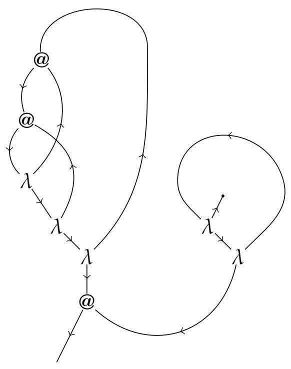
dual to the inclusion of neutral terms into normal terms. These coercions are "virtual", in the sense that the extended grammar

$$\begin{aligned} e &::= x \mid e(m) \mid (m : R) \\ m &::= e \mid \lambda x.m \end{aligned}$$

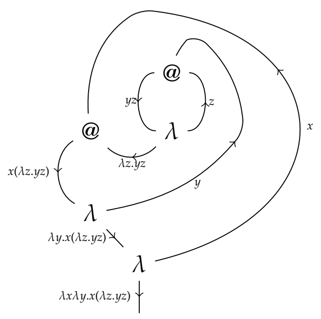
strictly speaking no longer describes the classical properties of being neutral or normal (Defn. 2.2.27), but rather can be seen as a way of "polarizing" the subterms of an arbitrary term, marking them as being amenable to either type synthesis or type checking.

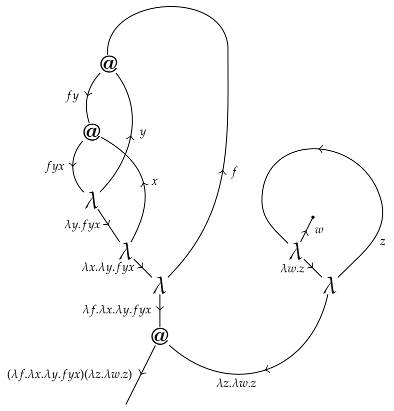
There's actually a way of visualizing this bidirectional flow of type information, which I'm fond of, so please forgive a momentary digression. This visualization technique builds on a folklore representation of lambda terms as certain kinds of labelled graphs. In particular, the operations of application and abstraction can be represented diagrammatically by considering graphs with two different types of trivalent vertex: a "@-vertex" having two inputs (t and u) and one output ($t(u)$), and a "λ-vertex" having one input (t) and two outputs (x and $\lambda x.t$). For example, below are diagrams representing the terms $\lambda x.\lambda y.x(\lambda z.yz)$ and $(\lambda f.\lambda y.\lambda x.fyx)(\lambda z.\lambda w.z)$:

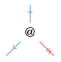
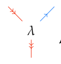
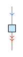
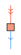


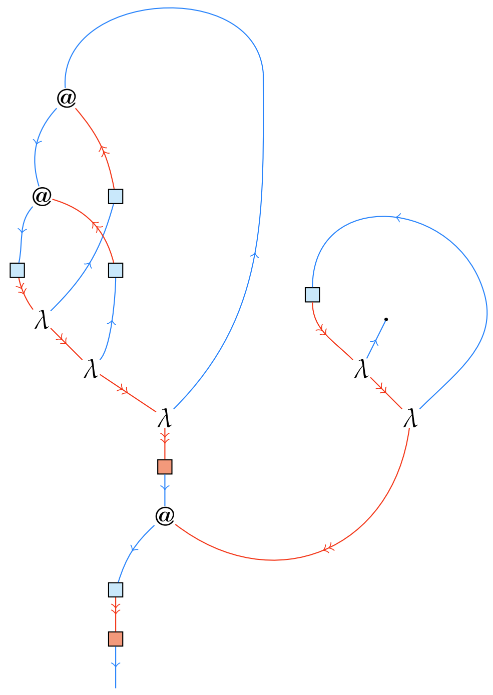


(This is a bit more obvious if we annotate wires with subterms:)

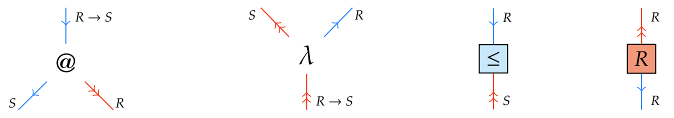




The "neutral vs. normal" distinction can then be incorporated into these diagrams by coloring the wires, say, in blue and red. So, @-vertices get colored as  and λ-vertices as , while we use explicit "converter boxes"  and  to mediate between the two types of wires. For example, here is a bicolored version of the diagram for $(\lambda f.\lambda y.\lambda x.fyx)(\lambda z.\lambda w.z)$:



To now get to bidirectional typing, all we have to do is reverse the red wires! The idea is we view blue wires as carrying *positive* information about the principal types of the corresponding subterms, while we view red wires as carrying *negative* information in the form of type checking obligations. The bidirectional typing algorithm can be described very concisely by four local conditions,

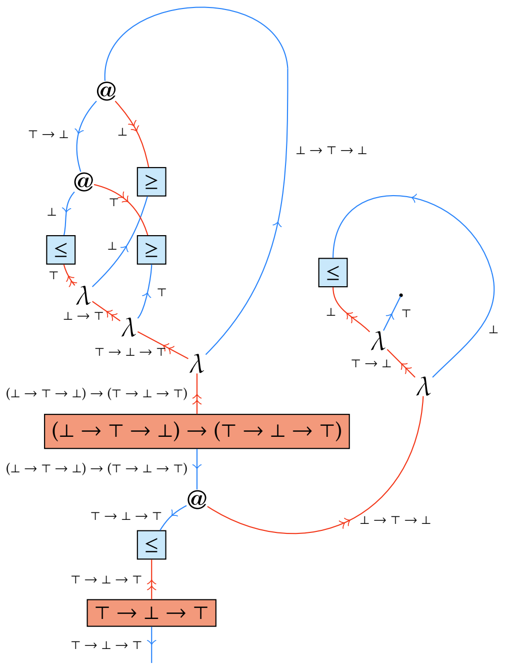
 (†)

which explain how to push this positive and negative information along the diagram of a term (containing some type annotations), and which correspond directly to the four typing rules *app*$^{\Rightarrow}$, *lam*$^{\Leftarrow}$, *sub*$^{\Leftarrow}_{\leq}$, and *ann*$^{\Rightarrow}$, respectively.[2]

Turning back to the motivating example at the beginning of our discussion, in order to verify that

$$\vdash (\lambda f.\lambda y.\lambda x.fyx)(\lambda z.\lambda w.z) : \top \to \bot \to \top$$

it suffices to annotate the subterm $\lambda f.\lambda y.\lambda x.fyx$ with an appropriate type, and bidirectional typing will accomplish the rest:



Exercise 2.2.37. In the language of your choice, write a bidirectional typechecker for $\lambda^{\leq}_{\to}$, and use it to check your answers to Exercises 2.2.10 and 2.2.11. (You

[2]The *var*$^{\Rightarrow}$ rule comes "for free" in the graphical language, by physical connection of wires.

may either work with a fixed preorder, or else write a modular typechecker that takes a decidable preorder as a parameter.)

Exercise 2.2.38. Prove that for all $R, S \sqsubset A$, if $x : R \vdash \eta_A[x] \Leftarrow S$ then $R \leq_A S$ (Hint: generalize the induction hypothesis), and combine this with Thm. 2.2.32 to derive a converse to Prop. 2.2.15. Finally, apply this fact to refactor your typechecker from Exercise 2.2.37, so that it answers questions about higher-order subtyping by type checking an appropriate identity coercion.

2.3 Intersection types

2.3.1 The intersection type refinement system $\lambda^{\leq\wedge}_{\to}$

Even though the type refinement system $\lambda^{\leq}_{\to}$ is rather limited in its expressive power, it already gave us a useful illustration of some important principles of type refinement. The most basic of these principles might just be the simple idea that a term can have more than one (refinement) type. One way of motivating *intersection type systems* (which in general have significantly greater expressive power than $\lambda^{\leq}_{\to}$) is that they give a way to "internalize" the fact that a term has multiple types. There are quite a few different variations of intersection typing in the literature, but he we will study intersection types in another well-behaved refinement of $\lambda_{\to}$. Like $\lambda^{\leq}_{\to}$, the intersection type refinement system $\lambda^{\leq\wedge}_{\to}$ does not increase the class of expressible programs – these remain limited to the simply-typed lambda terms – but it increases their class of *typings*.

In keeping with the approach we took to $\lambda^{\leq}_{\to}$, the definition of $\lambda^{\leq\wedge}_{\to}$ will be parameterized with respect to any preordered set of atoms with *finite meets*, so let's first recall what that means.

Definition 2.3.1 (Finite meets)**.** A **meet** of a pair of elements p_1, p_2 of a preordered set is an element $p_1 \wedge p_2$ such that $p_1 \wedge p_2 \preceq p_1$ and $p_1 \wedge p_2 \preceq p_2$, and moreover such that for any other element q, if $q \preceq p_1$ and $q \preceq p_2$ then $q \preceq p_1 \wedge p_2$. A preordered set $(\mathcal{P}, \preceq)$ is said to have **finite meets** if every pair of elements $p_1, p_2 \in \mathcal{P}$ has a meet $p_1 \wedge p_2 \in \mathcal{P}$ and there is also a **top** element $\top \in \mathcal{P}$, where $q \preceq \top$ for all $q \in \mathcal{P}$. (A partial order with finite meets is commonly called a *meet semilattice*.)

In a sense, all that intersection types do in $\lambda^{\leq\wedge}_{\to}$ is explain how to extend the concept of finite meet from atomic elements to arbitrary refinements of simple types, with the only subtlety being the emergence of some non-trivial subtyping relations. Let's begin by studying the refinement and typing rules, before we get to the subtlety.

For binary intersections we have:

$$\frac{R_1 \sqsubset A \quad R_2 \sqsubset A}{R_1 \wedge R_2 \sqsubset A} \qquad \frac{\Pi \vdash t : R_1 \quad \Pi \vdash t : R_2}{\Pi \vdash t : R_1 \wedge R_2}\wedge I \qquad \frac{\Pi \vdash t : R_1 \wedge R_2}{\Pi \vdash t : R_i}\wedge E_i$$

The refinement rule says that it it is possible to form the intersection of any pair of refinements of the same type, while the typing rules just formalize the idea of combining multiple typings of the same term (NB: the index i in $\wedge E_i$ stands for either 1 or 2, so this is really a pair of rules). The "0-ary" versions of these rules are even simpler:

$$\frac{}{\top_A \sqsubset A} \qquad \frac{}{\Pi \vdash t : \top_A}\ \top I \qquad \text{(no elimination rule for } \top\text{)}$$

Note we index the "top" refinement by the type A it refines, to avoid ambiguity, although sometimes we can omit this index when it is clear from context.

Now as for subtyping, clearly all of the following subtyping relations and rules should be admissible:

$$R_1 \wedge R_2 \leq_A R_1 \quad R_1 \wedge R_2 \leq_A R_2 \qquad \frac{S \leq_A R_1 \quad S \leq_A R_2}{S \leq_A R_1 \wedge R_2} \qquad S \leq_A \top$$

Essentially, these just express that the preorder of refinements of a type A has finite meets. It turns out, though, that we should also admit the following subtyping relations between refinements of function types:

$$(S \to R_1) \wedge (S \to R_2) \leq_{A\to B} S \to (R_1 \wedge R_2) \qquad \text{(dist-}\wedge\text{)}$$
$$\top \leq_{A\to B} S \to \top \qquad \text{(dist-}\top\text{)}$$

One way you might justify these distributivity principles to yourself is by thinking about the subset interpretation of refinement types, and reading intersection types as intersection of subsets (see Section 2.3.2). You can also justify them by recalling the discussion of Section 2.2.2, and trying to realize (dist-$\wedge$) and (dist-$\top$) as typings of an identity coercion (see Section 2.3.3). For now, though, for the sake of further analysis, let's gather all of these rules into one formal definition.

Definition 2.3.2 (System $\lambda^{\leq\wedge}_{\to\eta}$)**.** Let $(\mathcal{P}, \leq)$ be a preordered set with finite meets. The refinement, subtyping, and typing relations for $\lambda^{\leq\wedge}_{\to}$ are defined inductively by the union of the rules of $\lambda^{\leq}_{\to}$ (Definitions 2.2.1, 2.2.2 and 2.2.7) together with:

- Refinement:

$$\frac{R_1 \sqsubset A \quad R_2 \sqsubset A}{R_1 \wedge R_2 \sqsubset A} \qquad \frac{}{\top_A \sqsubset A}$$

- Subtyping:

$$\frac{}{R_1 \wedge R_2 \leq_A R_1} \quad \frac{}{R_1 \wedge R_2 \leq_A R_2} \qquad \frac{S \leq_A R_1 \quad S \leq_A R_2}{S \leq_A R_1 \wedge R_2} \qquad \frac{}{S \leq_A \top}$$

$$\frac{}{(S \to R_1) \wedge (S \to R_2) \leq_{A\to B} S \to (R_1 \wedge R_2)} \qquad \frac{}{\top \leq_{A\to B} S \to \top}$$

$$\frac{}{R \leq_A R} \qquad \frac{R \leq_A S \quad S \leq_A T}{R \leq_A T}$$

- Typing:

$$\frac{\Pi \vdash t : R_1 \quad \Pi \vdash t : R_2}{\Pi \vdash t : R_1 \wedge R_2} \wedge I \qquad \frac{\Pi \vdash t : R_1 \wedge R_2}{\Pi \vdash t : R_i} \wedge E_i \qquad \frac{}{\Pi \vdash t : \top_A} \top I$$

Exercise 2.3.3. In this exercise and the next one, take $(\mathcal{P}, \leq)$ to be the two-element preorder $\mathcal{P}_2 = \bot \leq \top$, which is also a trivial meet semilattice with

$$\bot \wedge \bot = \bot \wedge \top = \top \wedge \bot = \bot \quad \text{and} \quad \top \wedge \top = \top$$

and top element $\top$. Let's say that two refinements $R, S \sqsubset A$ are in the same *equivalence class* if $R \leq S$ and $S \leq R$. Verify that the refinements of $\mathbb{B} = \iota \to \iota \to \iota$ are partitioned into five equivalence classes in $\lambda^{\leq \wedge}_{\to}$, suggestively labeled by the vertices of the following Hasse diagram:

$$\begin{tikzcd}
 & \top_{\mathbb{B}} \arrow[dl, dash] \arrow[dr, dash] & \\
true \arrow[dr, dash] & & false \arrow[dl, dash] \\
 & fuzzy \arrow[d, dash] & \\
 & \bot_{\mathbb{B}} &
\end{tikzcd}$$

Exercise 2.3.4. Recall the standard definitions of conjunction, disjunction, and negation on the Church booleans:

$$\begin{aligned}
and, or &: \mathbb{B} \to \mathbb{B} \to \mathbb{B} \\
and &= \lambda p.\lambda q.\lambda x.\lambda y.p\,(q\,x\,y)\,y \\
or &= \lambda p.\lambda q.\lambda x.\lambda y.p\,x\,(q\,x\,y) \\
not &: \mathbb{B} \to \mathbb{B} \\
not &= \lambda p.\lambda x.\lambda y.p\,y\,x
\end{aligned}$$

Find principal types for *and*, *or*, and *not* with respect to $\mathcal{P}_2$. (Hint: beware the excluded middle!)

Exercise 2.3.5 (cf. Hindley [16], Dunfield [11]). The typing rules for intersection types look similar to the standard typing rules for product types,

$$\frac{\Pi \vdash t_1 : R_1 \quad \Pi \vdash t_2 : R_2}{\Pi \vdash (t_1, t_2) : R_1 \times R_2} \times I \qquad \frac{\Pi \vdash t : R_1 \times R_2}{\Pi \vdash \pi_i\, t : R_i} \times E_i$$

except that $\times I$ and $\times E_i$ come with explicit pairing and projection operations on terms, whereas the term t remains fixed in rules $I\wedge$ and $E_i\wedge$. Suppose that we define an "elaboration" of typing derivations in an intersection type system into the simply typed lambda calculus with products, by a translation whose effect on types is to map $\wedge$ to $\times$ and leave everything else unchanged:

$$p^* = p \qquad (R \to S)^* = R^* \to S^* \qquad (R \wedge S)^* = R^* \times S^*$$

Give an example of a (refinement) type $R \sqsubset A$, such that R is uninhabited in $\lambda^{\leq\wedge}_{\to}$ (i.e., there is no closed term $t : A$ such that $t : R$), but its translation R^* is inhabited in the simply typed lambda calculus with products.

2.3.2 Extending the subset interpretation

The interpretation of refinements as subsets that we discussed in Section 2.2.3 extends in a straightforward way to cover $\lambda^{\leq\wedge}_{\to}$, with intersection of refinements ($\wedge$) interpreted as intersection of subsets ($\cap$), and "top" refinements interpreted as the maximal subset:

$$[\![R_1 \wedge R_2]\!] = [\![R_1]\!] \cap [\![R_2]\!] \qquad [\![\top_A]\!] = [\![A]\!]$$

The only subtlety is that for this semantics to be coherent, we must also require the interpretation function

$$i[-] : \mathcal{P} \to 2^D$$

to preserve finite meets in addition to being order-preserving. Under that assumption, the soundness result (Prop. 2.2.16) continues to hold. For example, the distributivity principle

$$(S \to R_1) \wedge (S \to R_2) \leq_{A \to B} S \to (R_1 \wedge R_2)$$

has the following reading as an inclusion of subsets: if $f : A \to B$ is a function which maps elements of $[\![S]\!]$ to elements of $[\![R_1]\!]$ and which *also* maps elements of $[\![S]\!]$ to elements of $[\![R_2]\!]$, then f will map elements of $[\![S]\!]$ to elements of the intersection $[\![R_1]\!] \cap [\![R_2]\!]$.

2.3.3 A further analysis of subtyping

Looking at Defn. 2.3.2, there is some incongruity between the concise statement of the refinement and typing rules for intersection types, and the definition of the subtyping relation in $\lambda^{\leq\wedge}_{\to}$, which seems a lot more complex. This is largely an artifact of the presentation, though, and there are equivalent but more elegant ways of defining subtyping.

The quickest way of defining subtyping is to reduce it to typing and η-expansion, like we discussed in Section 2.2.2. For example, the following (partial) typing derivation shows how one can derive the distributivity principle (dist-$\wedge$) in this manner:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots \vdash x : (S \to R_1) \wedge (S \to R_1)}{\cdots \vdash x : S \to R_1}\,\wedge E_1 \qquad \ldots, y : S \vdash y : S}{\ldots, y : S \vdash x(y) : R_1}\,\mathit{app} \qquad \begin{matrix}\vdots \\ \ldots, y : S \vdash x(y) : R_2\end{matrix}}{\ldots, y : S \vdash x(y) : R_1 \wedge R_2}\,\wedge I}{x : (S \to R_1) \wedge (S \to R_2) \vdash \lambda y.x(y) : S \to (R_1 \wedge R_2)}\,\mathit{abs}$$

We can derive (dist-$\top$) similarly (but even more quickly):

$$\dfrac{\dfrac{}{x : \top, y : S \vdash x(y) : \top}\,\top I}{x : \top \vdash \lambda y.x(y) : S \to \top}\,\mathit{abs}$$

Another approach inspired by sequent calculus is to generalize the subtyping judgment to take a *list* of refinements on the left,

$$R_1, \ldots, R_n \ll_A S$$

with the intended interpretation that

$$R_1, \ldots, R_n \ll_A S \qquad \text{iff} \qquad R_1 \wedge \cdots \wedge R_n \leq_A S.$$

The following does the trick:

Notation. We write $\vec{R} \sqsubset A$ to stand for a list of refinements of the same type $\vec{R} = R_1, \ldots, R_n \sqsubset A$. We write $i \in \binom{n}{k}$ (where k, n are natural numbers) to indicate that i is a sequence of k distinct indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

Definition 2.3.6 (Sequent subtyping)**.** The **sequent subtyping relation** $\vec{R} \ll_A S$ ($\vec{R}, S \sqsubset A$) is defined inductively by the following rules:

$$\dfrac{\vec{R_0}, R_1, R_2, \vec{R} \ll_A S}{\vec{R_0}, R_1 \wedge R_2, \vec{R} \ll_A S} \qquad \dfrac{\vec{R} \ll_A S_1 \quad \vec{R} \ll_A S_2}{\vec{R} \ll_A S_1 \wedge S_2} \qquad \dfrac{\vec{R_0}, \vec{R} \ll_A S}{\vec{R_0}, \top, \vec{R} \ll_A S} \qquad \dfrac{}{\vec{R} \ll_A \top}$$

$$\dfrac{p_1 \wedge \cdots \wedge p_n \leq q}{p_1, \ldots, p_n \ll_\iota q} \qquad \dfrac{i \in \binom{n}{k} \quad R \ll_A R_{i_1} \quad \cdots \quad R \ll_A R_{i_k} \quad S_{i_1}, \ldots, S_{i_k} \ll_B S}{R_1 \to S_1, \ldots, R_n \to S_n \ll_{A \to B} R \to S}$$

Like our original definition of subtyping in $\lambda^{\leq}_{\to}$, this definition has the advantage of admitting an immediate algorithmic interpretation, by reading the rules bottom-up. (Defn. 2.3.2 does not admit a bottom-up interpretation due to the explicit inclusion of transitivity.) On the other hand, this algorithm is not particularly efficient: to apply the subtyping rule for function types requires choosing a k-element subset of the n refinements on the left, and there are a total of $\sum_k \binom{n}{k} = 2^n$ possible such choices! We will return to the more general question of the complexity of type checking intersection types in Section 2.3.6. For now, let's record the fact that these various definitions of subtyping are equivalent.

Theorem 2.3.7. *Let $R, S \sqsubset A$. The following are equivalent in $\lambda^{\leq\wedge}_{\rightarrow}$:*

1. *$R \leq_A S$ is derivable*
2. *$x : R \vdash \eta_A[x] : S$ is derivable with subsumption restricted to atomic instances.*
3. *$R \ll_A S$ is derivable*

Proof sketch. The implication (3) $\Rightarrow$ (1) is an easy induction applying the intended interpretation of the sequent subtyping judgment described above, while we have given some indications of the implication (1) $\Rightarrow$ (2), although there is some work to do in explaining how to do to deal with reflexivity and transitivity. The implication (2) $\Rightarrow$ (3) becomes a lot easier after we've introduced a bidirectional type system for intersection types in Section 2.3.5. For a full proof of an analogous (but more sophisticated) result, see Lovas and Pfenning [24, §4]. $\square$

2.3.4 Subject expansion and the complexity of type inference

From exercise Exercise 2.3.4 you know that it is sometimes possible to compute principal types for terms in $\lambda^{\leq\wedge}_{\rightarrow}$, essentially because we can use intersection types to list all of the distinct possible behaviors of a term (assuming there are only finitely many atomic refinements). For various reasons, though, this observation does not necessarily yield the best path towards a practical type system. One reason is that these principal types can get rather large. For example, if we keep the same two-element lattice of atomic refinements $\mathcal{P}_2$ as in the exercises, then the term

$$\begin{aligned}
&\mathit{test} : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \\
&\mathit{test} = \lambda p.\lambda q.\lambda r.\mathit{and}\, p\, (\mathit{or}\, (\mathit{not}\, q)\, r)
\end{aligned}$$

has the following principal type:

$$\begin{aligned}
\mathit{test} : \;& \mathit{true} \rightarrow \mathit{true} \rightarrow \mathit{true} \rightarrow \mathit{true} \\
&\wedge \mathit{true} \rightarrow \mathit{true} \rightarrow \mathit{false} \rightarrow \mathit{false} \\
&\wedge \mathit{true} \rightarrow \mathit{true} \rightarrow \mathit{fuzzy} \rightarrow \mathit{fuzzy} \\
&\wedge \mathit{true} \rightarrow \mathit{true} \rightarrow \bot_{\mathbb{B}} \rightarrow \bot_{\mathbb{B}} \\
&\wedge \mathit{true} \rightarrow \mathit{false} \rightarrow \top_{\mathbb{B}} \rightarrow \mathit{true} \\
&\wedge \mathit{true} \rightarrow \mathit{fuzzy} \rightarrow \mathit{false} \rightarrow \mathit{fuzzy} \\
&\wedge \mathit{true} \rightarrow \bot_{\mathbb{B}} \rightarrow \top_{\mathbb{B}} \rightarrow \bot_{\mathbb{B}} \\
&\wedge \mathit{false} \rightarrow \top_{\mathbb{B}} \rightarrow \top_{\mathbb{B}} \rightarrow \mathit{false} \\
&\wedge \mathit{fuzzy} \rightarrow \mathit{true} \rightarrow \mathit{true} \rightarrow \mathit{fuzzy} \\
&\wedge \mathit{fuzzy} \rightarrow \mathit{false} \rightarrow \top_{\mathbb{B}} \rightarrow \mathit{fuzzy} \\
&\wedge \bot_{\mathbb{B}} \rightarrow \top_{\mathbb{B}} \rightarrow \top_{\mathbb{B}} \rightarrow \bot_{\mathbb{B}}
\end{aligned}$$

Of course, if we replace $\mathcal{P}_2$ by something else then the principal type could get even bigger! Moreover, besides being verbose, oftentimes such principal types are not particularly useful, because they describe the behavior of a program on inputs which will never be generated – for example, you may have noticed that there are no closed Church booleans of type *fuzzy* or $\bot_{\mathbb{B}}$. Rather than attempting to compute principal types for arbitrary terms, a much more practical solution to type checking with intersection types is to use bidirectional typing, as we will discuss in Section 2.3.5. Before we get there, though, I'd like to discuss another theoretical barrier to pure type inference for $\lambda^{\leq\wedge}_{\to}$, which is closely related to some of the historical motivations for studying intersection type systems.

Like many other intersection type systems, $\lambda^{\leq\wedge}_{\to}$ satisfies both the usual subject reduction (a.k.a. type preservation) property as well as a dual "subject expansion" property.

Definition 2.3.8 (Subject reduction/expansion)**.** A type system with typing relation $\Pi \vdash t : R$ and rewriting relation $t \to t'$ is said to satisfy...

- **subject reduction**: if $\Pi \vdash t : R$ and $t \to t'$ implies $\Pi \vdash t' : R$.
- **subject expansion**: if $t \to t'$ and $\Pi \vdash t' : R$ implies $\Pi \vdash t : R$.

As usual, subject reduction (with respect to β-conversion) relies on a substitution lemma, which is proved by an easy induction.

Lemma 2.3.9 (Substitution, $\lambda^{\leq\wedge}_{\to}$)**.** *If* $\Pi \vdash u : R$ *and* $\Pi, x : R \vdash t : S$ *then* $\Pi \vdash t[u/x] : S$.

Dually, subject expansion relies on a converse to the substitution lemma, typical of intersection type systems.

Lemma 2.3.10 (Converse substitution, $\lambda^{\leq\wedge}_{\to}$)**.** *If* $\Pi \vdash t[u/x] : S$ *then there exists an* R *such that* $\Pi \vdash u : R$ *and* $\Pi, x : R \vdash t : S$.

Proof. By induction on the derivation of $\Pi \vdash t[u/x]$. We show some representative cases:

Case $\dfrac{\Pi \vdash t[u/x] : S_1 \quad \Pi \vdash t[u/x] : S_2}{\Pi \vdash t[u/x] : S_1 \wedge S_2}\wedge I$**:** By the i.h. there exists an R_1 such that $\Pi \vdash u : R_1$ and $\Pi, x : R_1 \vdash t[u/x] : S_1$, as well as an R_2 such that $\Pi \vdash u : R_2$ and $\Pi, x : R_2 \vdash t[u/x] : S_2$. The two hypothetical derivations can be weakened to $\Pi, x : R_1 \wedge R_2 \vdash t[u/x] : S_1$ and $\Pi, x : R_1 \wedge R_2 \vdash t[u/x] : S_2$ by contravariant subsumption. Then by application of $I\wedge$ twice we obtain $\Pi \vdash u : R_1 \wedge R_2$ and $\Pi, x : R_1 \wedge R_2 \vdash t[u/x] : S_1 \wedge S_2$.

Case $\dfrac{}{\Pi \vdash t[u/x] : \top}\top I$**:** By $I\top$ twice we have $\Pi \vdash u : \top$ and $\Pi, x : \top \vdash t : \top$.

Case $\wedge E$**:** immediate by the i.h. and reapplying $\wedge E$.

Case $\dfrac{}{\Pi, y : S \vdash y[u/x] : S}var$**:** Take $R = \top$.

Case $\dfrac{\Pi \vdash t_1[u/x] : S' \to S \quad \Pi \vdash t_2[u/x] : S'}{\Pi \vdash t_1(t_2)[u/x] : S}\ app$**:** Similar to $I\wedge$.

Case $\dfrac{\Pi, y : S' \vdash t[u/x] : S}{\Pi \vdash \lambda y.t[u/x] : S' \to S}\ lam$**:** By applying the i.h. and a strengthening lemma.

□

Corollary 2.3.11 (Subject reduction + expansion, $\lambda^{\leq\wedge}_{\to}$). *$\lambda^{\leq\wedge}_{\to}$ satisfies both the subject reduction and subject expansion properties with respect to β-conversion.*

Proof. By induction on typing derivations, applying the substitution and converse substitution lemmas. (Incidentally, the subject reduction and expansion properties also hold with respect to η-conversion, which is related to our discussion of subtyping in Section 2.3.3.) □

Corollary 2.3.12 (Complexity of typing, $\lambda^{\leq\wedge}_{\to}$). *Deciding the typing relation in $\lambda^{\leq\wedge}_{\to}$ requires non-elementary time (i.e., is not bounded by any fixed tower of exponentials), for any preorder of atomic refinements with at least two inequivalent elements $\bot < \top$.*

Proof. There are two closed normal terms of type $\iota \to \iota \to \iota$ in $\lambda_{\to}$, namely the Church booleans $true = \lambda x.\lambda y.x$ and $false = \lambda x.\lambda y.y$. As a consequence of subject reduction + expansion, we can decide whether a closed term $t : \iota \to \iota \to \iota$ normalizes to *true* by asking whether $t : \bot \to \top \to \bot$. But by Statman's theorem [44, 26], deciding whether t normalizes to *true* requires non-elementary time. □

In addition to the practical issue we already discussed, this complexity issue imposes a strong theoretical barrier to any approach to type checking $\lambda^{\leq\wedge}_{\to}$ based on pure inference. Moreover, the fact that the typing relation is decidable at all relies on a special property of simply typed lambda calculus, that all terms eventually reduce to a normal form. In contrast, intersection type refinement systems built on top of Turing-complete languages (such as the original intersection type systems refining pure lambda calculus) often have an undecidable typing relation.

2.3.5 Bidirectional typing for intersection types

The bidirectional typing algorithm we discussed in Section 2.2.4 extends in a natural way to deal with intersection types. The introduction and elimination rules for intersection types divide smoothly into checking and synthesis:

$$\frac{}{\Pi \vdash m \Leftarrow \top_A}\ \top I^{\Leftarrow} \qquad \frac{\Pi \vdash m \Leftarrow R_1 \quad \Pi \vdash m \Leftarrow R_2}{\Pi \vdash m \Leftarrow R_1 \wedge R_2}\ \wedge I^{\Leftarrow} \qquad \frac{\Pi \vdash e \Rightarrow R_1 \wedge R_2}{\Pi \vdash e \Rightarrow R_i}\ \wedge E_i^{\Rightarrow}$$

Limiting to the case of normal forms for the moment, the biggest difference with bidirectional typing for $\lambda^{\leq}_{\to}$ is that the synthesis relation is nondeterministic, in other words, it denotes a *multifunction* from pairs of a context Π and neutral term e to types R, rather than a partial function.

Proposition 2.3.13 ($\Rightarrow$ multifunctional)**.** *For any $\Pi \sqsubset \Gamma$ and $\Gamma \vdash e : A$, there exist a finite number (possibly empty) of types $R_1, \ldots, R_n \sqsubset A$ such that $\Pi \vdash e \Rightarrow R_i$, and moreover each of the R_i must already occur as a subformula in Π (or as a subformula of an annotation in e, in the non-normal case).*

The completeness theorem for the normal fragment (Thm. 2.2.32) continues to hold, with a slightly adjusted statement.

Theorem 2.3.14 (Completeness of bidirectional typing for $\lambda^{\leq\wedge}_{\rightarrow}$, normal fragment)**.** *Let $\Gamma \vdash t : A$, and suppose $\Pi \vdash t : R$ for some $\Pi \sqsubset \Gamma$ and $R \sqsubset A$.*

a) If t is neutral, then there exist $R_1, \ldots, R_n \sqsubset A$ such that

$$\Pi \vdash t \Rightarrow R_1 \quad \cdots \quad \Pi \vdash t \Rightarrow R_n$$

and $R_1 \wedge \cdots \wedge R_n \leq R$.

b) If t is normal, then $\Pi \vdash t \Leftarrow R$.

The main issue for dealing with non-normal terms is that since intersection typing involves checking the same term against different types, we might end up needing a different set of annotations depending on which type the term is being checked against. For example, suppose that our context contains the binding

$$+ : (int \to int \to int) \wedge (real \to real \to real)$$

for an infix operator +, and we want to verify that

$$\lambda x.(\lambda z.x + z)\, x : (int \to int) \wedge (real \to real).$$

To derive the left branch of the intersection, we need to show that

$$\lambda z.x + z : int \to int$$

holds in a context extended by $x : int$, while to derive the right branch we need to show

$$\lambda z.x + z : real \to real$$

in a context extended by $x : real$. So what annotation should we place on the subterm $\lambda z.x + z$? Taking the intersection of these two types

$$\lambda z.x + z : (int \to int) \wedge (real \to real)$$

will not work, because in a given typing context, $\lambda z.x + z$ does not have *both* types $int \to int$ and $real \to real$, rather it just has one or the other depending on the type of x. (This example is not so contrived, because similar issues arise whenever one wants to define a function depending on local parameters and apply it later.)

A tentative solution to this problem is to specify a list of annotations, and ask the typechecker to choose the right one as needed in a given context. Such behavior can be neatly specified by the following rule:

$$\frac{\Pi \vdash m \Leftarrow R_i}{\Pi \vdash (m : R_1, \dots, R_n) \Rightarrow R_i}\; ann_i^{\Rightarrow}$$

For example, using this rule we can now derive

$$\lambda x.(\lambda z.x + z : int \to int, real \to real)\, x \Leftarrow (int \to int) \wedge (real \to real)$$

in the bidirectional type system.

Exercise 2.3.15. Extend the bidirectional typechecker you wrote in Exercise 2.2.37 with intersection types. To check subtyping relations, you can either try to implement the sequent subtyping algorithm described in Section 2.3.3, or else reuse your solution to Exercise 2.2.38 and save yourself some work.

2.3.6 PSPACE-completeness of bidirectional typing

Since we saw earlier that deciding the $\lambda^{\leq\wedge}_{\to}$ typing relation for general, unannotated terms requires non-elementary time, one might wonder about the situation for normal terms, or for terms with type annotations. It turns out that John Reynolds answered this question, showing that intersection typing is PSPACE-hard even in this situation. Since his proof is conceptual and cute, it seems like a nice way to conclude our discussion of intersection types.[3]

You already prepared for understanding this encoding with Exercise 2.3.4. Besides the operations of conjunction, disjunction, and negation, one can also define quantifiers over the Church booleans:

$$\begin{aligned} \mathit{forall}, \mathit{exists} &: (\mathbb{B} \to \mathbb{B}) \to \mathbb{B} \\ \mathit{forall} &= \lambda p.\mathit{and}\,(p\,(\lambda x.\lambda y.x))\,(p\,(\lambda x.\lambda y.y)) \\ \mathit{exists} &= \lambda p.\mathit{or}\,(p\,(\lambda x.\lambda y.x))\,(p\,(\lambda x.\lambda y.y)) \end{aligned}$$

These five operations correspond exactly to the language of *quantified boolean formulas* (QBF),

$$\phi ::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall p.\phi \mid \exists p.\phi$$

and it is clear how any QBF ϕ can be represented as a term t_ϕ composed out of *and*, *or*, *not*, *forall*, and *exists*.

Once again, let's consider refinements of the Church booleans $\mathbb{B} = \iota \to \iota \to \iota$ with respect to the two-element lattice of atomic refinements $\mathcal{P}_2 = \bot \leq \top$, but now limiting our attention to just these two refinements:

$$\mathit{true} \overset{\text{def}}{=} \bot \to \top \to \bot$$

$$\mathit{false} \overset{\text{def}}{=} \top \to \bot \to \bot$$

[3]Also, Reynolds' PSPACE-hardness result seems to be relatively unsung – it appears in an unpublished note [37], and later in the appendix of a technical report [38]. I learned about the existence of this result from reading William Lovas' thesis [23].

Then it is easy to verify that the operations *and*, *or*, *not*, *forall*, *exists* can be assigned the following refinement types:[4]

$$\begin{aligned}
\mathit{and} : {} & \mathit{true} \to \mathit{true} \to \mathit{true} \\
& \wedge \mathit{true} \to \mathit{false} \to \mathit{false} \\
& \wedge \mathit{false} \to \mathit{true} \to \mathit{false} \\
& \wedge \mathit{false} \to \mathit{false} \to \mathit{false} \\
\mathit{or} : {} & \mathit{true} \to \mathit{true} \to \mathit{true} \\
& \wedge \mathit{true} \to \mathit{false} \to \mathit{true} \\
& \wedge \mathit{false} \to \mathit{true} \to \mathit{true} \\
& \wedge \mathit{false} \to \mathit{false} \to \mathit{false} \\
\mathit{not} : {} & \mathit{true} \to \mathit{false} \\
& \wedge \mathit{false} \to \mathit{true} \\
\mathit{forall} : {} & ((\mathit{true} \to \mathit{true}) \wedge (\mathit{false} \to \mathit{true})) \to \mathit{true} \\
& \wedge ((\mathit{true} \to \mathit{true}) \wedge (\mathit{false} \to \mathit{false})) \to \mathit{false} \\
& \wedge ((\mathit{true} \to \mathit{false}) \wedge (\mathit{false} \to \mathit{true})) \to \mathit{false} \\
& \wedge ((\mathit{true} \to \mathit{false}) \wedge (\mathit{false} \to \mathit{false})) \to \mathit{false} \\
\mathit{exists} : {} & ((\mathit{true} \to \mathit{true}) \wedge (\mathit{false} \to \mathit{true})) \to \mathit{true} \\
& \wedge ((\mathit{true} \to \mathit{true}) \wedge (\mathit{false} \to \mathit{false})) \to \mathit{true} \\
& \wedge ((\mathit{true} \to \mathit{false}) \wedge (\mathit{false} \to \mathit{true})) \to \mathit{true} \\
& \wedge ((\mathit{true} \to \mathit{false}) \wedge (\mathit{false} \to \mathit{false})) \to \mathit{false}
\end{aligned}$$

But since these types are just representations of the truth tables for the corresponding QBF operations, we have that for any QBF ϕ,

$$\phi \text{ true} \qquad \text{iff} \qquad m_\phi \Leftarrow \mathit{true}$$

where m_ϕ is the term t_ϕ annotated by the signatures above. Alternatively, we can also represent ϕ as an *open* β-normal term m_ϕ, just by postulating a context with operations of the above type. In either case, since evaluation of quantified boolean expressions is PSPACE-complete, this implies PSPACE-hardness.

Theorem 2.3.16 (Reynolds [37, 38])**.** *Deciding the relation* $\Pi \vdash m \Leftarrow R$ *of* $\lambda^{\leq\wedge}_{\to}$ *is PSPACE-hard.*

Conversely, it is not difficult to see that the naive logic programming interpretation of the bidirectional typing rules gives an algorithm in PSPACE. This is just because whenever there is any existential or universal branching, it always only involves quantification over types and terms which already appear in the original input (cf. Prop. 2.3.13). Therefore, like the QBF problem itself, bidirectional typing with intersection types is PSPACE-complete.

Corollary 2.3.17. *Deciding the relation* $\Pi \vdash m \Leftarrow R$ *of* $\lambda^{\leq\wedge}_{\to}$ *is PSPACE-complete.*

[4]Note these are *not* their principal types!

2.4 Refining ML-like languages

Before ending this chapter, I want to give a very brief overview of some of the issues involved in the design of type refinement systems, when passing from $\lambda_\to$ to an ML-like language.

2.4.1 Refining datatypes: datasorts and index refinements

One of the perks of programming in languages like ML is the ability to define functions over algebraic datatypes via pattern-matching. A common source of frustration, though, is that often the types are too imprecise to rule out inputs which the programmer knows to be impossible, leading to spurious warnings from the compiler about missing cases (SML/NJ's infamous "Warning: match nonexhaustive").

The idea of refining user-defined datatypes to capture more precise invariants of programs was proposed by Freeman and Pfenning in "Refinement Types for ML" [13]. That paper described a particularly natural and effective way of specifying such refinements recursively, using a syntax analogous to ML's datatype mechanism itself. In the followup literature these have been referred to as *datasort refinements* beginning with Rowan Davies' work [7, 8].

For a demonstration of datasort refinements in action, let's look at some simple examples in Davies' CIDRE system.[5] In ML, we know how to define the natural numbers:

```
datatype nat = Z | S of nat
```

Well, in CIDRE, we can also define, say, the *positive* natural numbers, or the *even* and *odd* natural numbers:

```
datasort pos = S of nat
datasort even = Z | S of odd
     and odd = S of even
```

For example, we can write the predecessor function by pattern-matching, and have CIDRE verify that it sends positive natural numbers to arbitrary natural numbers:

```
(*[ pred <: pos -> nat ]*)
fun pred (S n) = n
```

In particular, here the type checker can statically rule out the `Z` constructor as an input to the predecessor function, so there is no need for a warning. Similarly, we can write addition and multiplication on natural numbers as usual, and ask CIDRE to check that they have the expected behavior on even/odd numbers:

```
(*[ plus <: (even -> even -> even) & (even -> odd -> odd)
          & (odd -> even -> odd) & (odd -> odd -> even) ]*)
```

[5]Available at `https://github.com/rowandavies/sml-cidre`.

```
fun plus Z n = n
  | plus (S m) n = S (plus m n)

(*[ times <: (even -> even -> even) & (even -> odd -> even)
           & (odd -> even -> even) & (odd -> odd -> odd) ]*)
fun times Z n = Z
  | times (S m) n = plus n (times m n)
```

These more refined type signatures make it easier to catch bugs. For example, if we had made a mistake in writing `times`, say,

```
  | times (S m) n = plus m (times m n)
```

then the program would not have passed the refinement typechecker (although it would still define a valid ML function of type `nat -> nat -> nat` which we could compile and run). Freeman and Pfenning's original implementation of datasort refinements was based on pure type inference, but this was found to be impractical – essentially for the reasons we discussed in Section 2.3.4 – and the CIDRE system is based on bidirectional typing.

Another approach to refining ML datatypes is known as *index refinements* [46, 45]. Typical examples are refining lists by their length, natural numbers by an upper bound, or booleans by their value. Many of these examples are familiar from dependently-typed programming, but the characteristic of approaches based on refinement is that usually the indices are drawn from some decidable constraint domain, meaning that in principle there is no additional proof burden for the programmer besides finding the right type annotations.

Joshua Dunfield [10] has a prototype implementation of a system that combines datasort refinements with index refinements, called Stardust.[6] In Stardust, it is possible for example to verify the correctness of a procedure for red-black tree insertion, maintaining the color invariants using datasort refinements, and the height invariants using index refinements.

2.4.2 The value restriction

Davies and Pfenning [9] observed that the standard introduction rule for intersection types (Section 2.3.1) is unsound in the presence of ML-style effects. The reasons are very similar to the reason unrestricted let-polymorphism is unsound in ML. For example, consider the following ML program, annotated in CIDRE:

```
(*[ x <: pos ref & nat ref ]*)
val x = ref (S Z)
(*[ unsound <: pos ]*)
val unsound = (x := Z; !x)
```

[6] Available at `http://www.mpi-sws.org/~joshua/stardust/`.

Clearly the value of `unsound` = `Z` is not a positive natural number, so something is wrong with these type ascriptions. In fact, CIDRE's typechecker rejects this program because it imposes a *value restriction* on intersection introduction, corresponding to the following rule:

$$\frac{\Pi \vdash v : R_1 \quad \Pi \vdash v : R_2}{\Pi \vdash v : R_1 \wedge R_2}\ \wedge I_v$$

Here the letter v ranges over values, that is, expressions which are fully evaluated. Since the expression `ref (S Z)` is not a value (at run time, it allocates a fresh reference to a cell containing the value `S Z`), it cannot be given an intersection type. For similar reasons, the subtyping distributivity principle (dist-$\wedge$)

$$(S \to R_1) \wedge (S \to R_2) \leq S \to (R_1 \wedge R_2)$$

is unsound for call-by-value languages in the presence of effects, as is its 0-ary version (dist-$\top$), and the introduction rule ($I\top$) also requires a value restriction.

2.4.3 Union types and "tridirectional" typechecking

Union types are dual to intersection types. They arise naturally when considering datasort refinements. For example, from the datasort declarations we might expect that the subtyping relation

$$nat \leq even \vee odd$$

should hold, i.e., that every value of natural number type is either even or odd. At an even more basic level, we could imagine *decomposing* datasort declarations in terms of unions and *constructor refinements*. Intuitively, for example, the above datasort declarations for `even` and `odd` say that an even number is either zero or the successor of an odd number, and an odd number is the successor of an even number. We could equivalently express these definitions in terms of unions and constructor refinements:

$$\begin{aligned} even, odd &\sqsubset nat \\ even &= z \vee s(odd) \\ odd &= s(even) \end{aligned}$$

We will have more to say about this kind of decomposition in Chapter 3.

The refinement rules for union types (both in binary and 0-ary form) are just like the refinement rules for intersection:

$$\frac{R_1 \sqsubset A \quad R_2 \sqsubset A}{R_1 \vee R_2 \sqsubset A} \qquad \frac{}{\bot_A \sqsubset A}$$

Likewise, the introduction rules for $\vee$ and $\bot$ are dual to the elimination rules for $\wedge$ and $\top$:

$$\frac{\Pi \vdash t : R_i}{\Pi \vdash t : R_1 \vee R_2}\ \vee I_i \qquad \text{(no introduction rule for } \bot)$$

The elimination rules for union types are where it gets more interesting. It turns out that in the presence of effects, a sort of dual to the value restriction is needed:

$$\frac{\Pi \vdash t : R_1 \vee R_2 \quad \Pi, x : R_1 \vdash C[x] : S \quad \Pi, x : R_2 \vdash C[x] : S}{\Pi \vdash C[t] : S} \vee E_v$$

Here the letter C ranges over *evaluation contexts*, so that $C[t]$ stands for a term with an occurrence of t in evaluation position. The rule says that if t has union type $R_1 \vee R_2$, and the evaluation context $C[x]$ can be assigned type S assuming that the hole x has either type R_1 or type R_2, then $C[t]$ has type S. A potential generalization of this rule that one might consider is to eliminate arbitrarily many occurrences of a term of union type, which are not necessarily in evaluation position:

$$\frac{\Pi \vdash t : R_1 \vee R_2 \quad \Pi, x : R_1 \vdash t' : S \quad \Pi, x : R_2 \vdash t' : S}{\Pi \vdash t'[t/x] : S} \vee E$$

Indeed, such a rule has been proposed for union types in the setting of pure lambda calculus [1]. However, Dunfield and Pfenning [12] observed that the ($\vee E$) rule is unsound in the presence of effects. As a simple counterexample, we might imagine having an operation

$$\mathit{flip} : [0,1] \to \mathit{true} \vee \mathit{false}$$

taking as input a real number $0 \le p \le 1$, such that $\mathit{flip}(p)$ evaluates to *true* with probability p and to *false* with probability $1-p$. Well, using the unrestricted $\vee E$ rule we can derive the typing judgment

$$\mathit{flip}(0.5) == \mathit{flip}(0.5) : \mathit{true} \tag{2.5}$$

from the pair of typing judgments

$$x : \mathit{true} \vdash x == x : \mathit{true} \tag{2.6}$$

$$x : \mathit{false} \vdash x == x : \mathit{true} \tag{2.7}$$

after first factoring the original expression as $t' = x == x[\mathit{flip}(0.5)/x]$. Yet, although (2.6) and (2.7) are sound (since the types *true* and *false* are singletons), the typing judgment (2.5) is unsound, since the two occurrences of $\mathit{flip}(0.5)$ might evaluate to different booleans. The restricted elimination rule ($E_v\vee$) avoids this issue, since only one occurrence of $\mathit{flip}(0.5)$ can be eliminated at a time, in the order of evaluation. For similar reasons, the elimination rule for 0-ary unions also requires an evaluation context restriction in the presence of effects:

$$\frac{\Pi \vdash t : \bot_A}{\Pi \vdash C[t] : S} \bot E_v$$

Dunfield and Pfenning's paper [12] combined order-of-evaluation dependent typing rules with bidirectional typechecking, hence the origin of the tongue-in-cheek "tridirectional" typechecking.

2.5 Notes

The presentation of $\lambda^{\leq}_{\rightarrow}$ and $\lambda^{\leq\wedge}_{\rightarrow}$ I've given here owes much to Pfenning's expository essay [33], which considered simply typed lambda calculus refined with an atomic subtyping relationship and intersection types (I've ignored the interesting subject of *hereditary substitution*, which Pfenning's article reviews in depth). Type systems essentially equivalent to $\lambda^{\leq}_{\rightarrow}$ have been considered earlier, albeit usually without an explicit recognition of the refinement relationship: this is essentially what Benjamin Pierce [34] calls $\lambda_{\leq}$, and which he attributes to Luca Cardelli [4] as a "core calculus of subtyping". As already mentioned in Section 2.2.4, Mitchell [31] also looked at a system essentially equivalent to $\lambda^{\leq}_{\rightarrow}$ under the guise of "typing with atomic coercions", and came up with an algorithm for computing principal type schemes. Pierce has some more discussion of the difference between principal types and principal type schemes ("principal typings") in Chapter 22 of *Types and Programming Languages* [35]. (Some older sources are [30, 22].)

The use of bidirectional typing for intersection type systems was an innovation of Rowan Davies' thesis work [8]. He also introduced the $ann^{\Rightarrow}_{i}$ rule, adapting the idea of using multi-type annotations from Reynolds' older work on the Forsythe language [38]. Something which is perhaps not completely satisfactory about $ann^{\Rightarrow}_{i}$ is that it seems to introduce new nondeterminism into the type checker; to reduce the amount of nondeterminism, Dunfield and Pfenning proposed "contextual annotations" [12].

Intersection types were originally invented by Coppo and Dezani in the study of normalization of pure lambda terms [6, 2], in particular to give type-theoretic characterizations of different normalization properties. This has the consequence that pure type inference is usually undecidable. The idea of using intersection types in a practical type system to express more precise properties of programs was pioneered by Reynolds [38].

The idea of viewing types à la Curry as refinements of types à la Church is advocated explicitly by Pfenning in [33], and is also mentioned by Rowan Davies in the introduction to [8]. No doubt traces of this idea can be found earlier. For example, it is suggested reasonably explicitly by Sørensen and Urzyczyn in *Lectures on the Curry-Howard Isomorphism* [43, §11.4].

Chapter 3

A categorical perspective on type refinement

3.1 Introduction: type theory as an axiomatic theory

Type theory is a very formal branch of mathematics. We're used to thinking of type systems and programming languages as deductive systems, and via propositions-as-types, using them to prove formal theorems – whether they be simple tautologies of propositional logic derived in simply typed lambda calculus, or complex mathematical statements such as the four color theorem proved using Coq. In the past couple decades it has also become common to use formal proof assistants when establishing meta-theoretic properties of particular type systems and programming languages, such as type safety or strong normalization theorems.

That said, type theory as a "theory" is a bit different from other mathematical theories such as, say, group theory or graph theory, in that it doesn't follow the typical sociological pattern of a growing body of open problems, theorems, examples, and counterexamples built over a collection of widely-accepted (if evolving) definitions. Instead, type theory has many "one-shot theorems" about individual formal systems – it's true that often the proofs of these theorems can be retooled to prove similar properties of similar systems, but only rarely are they reused directly to obtain deeper results. In that sense, although type theory is characterized by a high degree of formal rigour, it is not an *axiomatic theory* in the same way that fields such as group theory and graph theory are (or at least pretend to be). This is not necessarily always a disadvantage, but it does make it more difficult for type theory to communicate with the rest of mathematics, and in general makes the field less accessible to outsiders.

To some extent this issue is addressed by the categorical approach to type theory pioneered in the 1970s and '80s, of which a paradigmatic example is Lambek's connection between simply typed lambda calculus with products and cartesian closed categories [20, 21]. In general, typed languages can be modelled axiomatically as categories with certain additional structure, which makes it possible to abstract away from particular presentations and prove general theorems about wide classes of languages. Still, despite many notable successes, the adoption of categorical methods within type theory (and programming languages more generally) has met with some resistance...and not always for bad reasons! In fact, this resistance is understandable, since there are many important aspects of type theory (and programming languages) which have so far eluded the categorical perspective.

The material that follows describes a categorical perspective on type refinement that I have been developing for several years in collaboration with Paul-André Melliès, some of which has been previously published [27, 28, 29]. One of the long-term motivations for this work is to build better mathematical tools that are adapted to the concerns of researchers in type theory and programming languages, and something specifically which seems to have been missing from standard categorical approaches was a clear articulation of the extrinsic view of typing. The basic idea here is to generalize Lambek's account by replacing categories with *functors*: a type refinement system can be modelled as a functor from a category whose objects are "types à la Curry" and morphisms are typing derivations to a category whose objects are "types à la Church" and morphisms are terms.

Experts may recognize that this approach has a very "fibrational" flavor, and in many ways our treatment of type refinement aligns with classical ideas in categorical logic – although the careful reader will observe that there are various subtle differences, stemming from the fact that we consider general functors rather than fibrations as the basic object of study. For people without a background in categorical logic, I hope that the interpretations provided here might also provide some alternative sources of intuition for this fascinating branch of mathematics.

3.2 Modelling type refinement systems as functors

3.2.1 Type refinement systems are not just categories

Before jumping to our study of type refinement systems as functors, it might be a good idea to explain what makes it difficult to study them directly as categories. Also, before everything else, let's recall the definition of a category and fix some notation:

Definition 3.2.1 (Categories)**.** A **category** C consists of:

- a collection C_0 of *objects* $(A, B, \dots)$;
- a collection C_1 of *morphisms* $(f, g, \dots)$, together with operations $\mathrm{src}, \mathrm{tgt} : C_1 \to C_0$ assigning to each morphism a unique source and target;

- for every pair $f, g \in C_1$ such that $\mathrm{src}(g) = \mathrm{tgt}(f)$, a *composition* $g \circ f \in C_1$ such that $\mathrm{src}(g \circ f) = \mathrm{src}(f)$ and $\mathrm{tgt}(g \circ f) = \mathrm{tgt}(g)$;
- for every $A \in C_0$, an *identity* $\mathrm{id}_A \in C_1$ such that $\mathrm{src}(A) = \mathrm{tgt}(_A) = A$;
- such that the associativity and unit laws hold whenever they make sense:

$$\begin{aligned} h \circ (g \circ f) &= (h \circ g) \circ f \\ f \circ \mathrm{id}_A &= f = \mathrm{id}_B \circ f \end{aligned}$$

Notation. We write $f : A \to B$ or $A \xrightarrow{f} B$ to indicate that f is a morphism such that $\mathrm{src}(f) = A$ and $\mathrm{tgt}(f) = B$. Given $f : A \to B$ and $g : B \to C$, we write either $(g \circ f) : A \to C$ or $(f; g) : A \to C$ to denote their composition. We sometimes write A or id instead of id_A for an identity morphism, the former when it is clear that we are speaking about a morphism, and the latter when the object A is clear from context or not important.

Now, the standard analogy between type systems and categories begins in saying that a term with one free variable

$$x : A \vdash t : B$$

can be interpreted as a morphism

$$\begin{tikzcd} A \arrow[r, "f"] & B \end{tikzcd}$$

in a category with sufficient structure. More generally this extends to an interpretation of terms with any number of free variables by considering categories with products (or multicategories), but the basic interpretation is already sufficient for illustrating the following problem.

Thinking back to the type refinement systems we considered in the previous chapter, what are we supposed to make of, say, the subsumption rule, or the introduction rule for intersection types?

$$\frac{\Pi \vdash t : R \quad R \leq S}{\Pi \vdash t : S} \qquad \frac{\Pi \vdash t : R_1 \quad \Pi \vdash t : R_2}{\Pi \vdash t : R_1 \wedge R_2}$$

In both cases we have multiple typing judgments made about the same term. But in a category, it does not really make sense to make multiple assertions

$$\begin{tikzcd} A \arrow[r, "f"] & B_1 \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "f"] & B_2 \end{tikzcd}$$

about the source and target of a single morphism. Or at least one can't make *distinct* assertions: in this case, if $\mathrm{tgt}(f) = B_1$ and $\mathrm{tgt}(f) = B_2$, then $B_1 = B_2$ by symmetry and transitivity. In other words, the interpretation of type systems as categories is inherently biased towards the intrinsic view of typing, which makes no distinction between terms and typing derivations.

3.2.2 Reading a functor as a type refinement system

Our solution is to incorporate the distinction between terms and typing derivations directly into the model, by viewing a type refinement system not as a category but as a *functor* from one category to another. Again, for reference let's recall:

Definition 3.2.2 (Functors)**.** Let $\mathcal{D}$ and $\mathcal{T}$ be categories. A **functor** $r : \mathcal{D} \to \mathcal{T}$ consists of:

- a pair of mappings $r : \mathcal{D}_0 \to \mathcal{T}_0$ and $r : \mathcal{D}_1 \to \mathcal{T}_1$, which are compatible in the sense that $\alpha : R \to S$ in $\mathcal{D}$ is sent to $r(\alpha) : r(R) \to r(S)$ in $\mathcal{T}$;
- such that composition and identity are preserved:

$$\begin{aligned} r(\beta \circ \alpha) &= r(\beta) \circ r(\alpha) \\ r(\mathrm{id}_R) &= \mathrm{id}_{r(R)} \end{aligned}$$

The idea is that a type refinement system can be modelled as the "erasure" functor (or "forgetful" functor) which maps a type refinement to the type that it refines, and a typing derivation to the term that it types. In fact, for the remainder of this chapter we will find it convenient to take this as a *definition* of "type refinement system" (or "refinement system" for short), as being in the most general case simply an arbitrary functor.

Definition 3.2.3. A **(type) refinement system** is a functor $r : \mathcal{D} \to \mathcal{T}$.

Although this definition may seem a bit austere, there is a legitimate sense in which any functor $r : \mathcal{D} \to \mathcal{T}$ can be interpreted as a very elementary type refinement system.

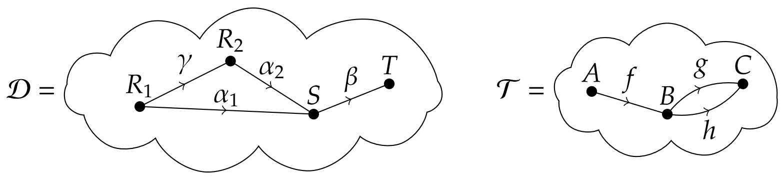
Definition 3.2.4 (Refinement)**.** We say that an object $R \in \mathcal{D}$ **refines** an object $A \in \mathcal{T}$ (notated $R \sqsubset A$) if $r(R) = A$.

Definition 3.2.5 (Judgments)**.** A **typing judgment** is a triple (R, f, S), where f is a morphism of $\mathcal{T}$ such that $R \sqsubset \mathrm{src}(f)$ and $S \sqsubset \mathrm{tgt}(f)$ (notated $R \underset{f}{\Longrightarrow} S$). In the special case where R and S refine the same object $R, S \sqsubset A$ and f is the identity morphism $f = \mathrm{id}_A$, the typing judgment (R, f, S) is also called a **subtyping judgment** (notated $R \underset{A}{\Longrightarrow} S$).

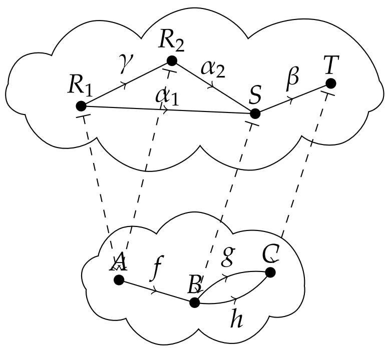
Definition 3.2.6 (Derivations)**.** A **derivation** of a (sub)typing judgment (R, f, S) is a morphism $\alpha : R \to S$ in $\mathcal{D}$ such that $r(\alpha) = f$ (notated $R \overset{\alpha}{\underset{f}{\Longrightarrow}} S$).

Since these definitions are all parameterized with respect to an arbitrary functor r, to be completely precise we should speak of r-judgments, r-derivations, etc., but usually the functor will be clear from context and we can leave off the prefix.

Example 3.2.7. Consider a pair of categories $\mathcal{D}$ and $\mathcal{T}$ freely generated from a pair of graphs:



Let's define a functor $r : \mathcal{D} \to \mathcal{T}$ with the action indicated on the left on objects, and the action indicated on the right on morphisms:



$$\begin{aligned} \gamma &\mapsto \mathrm{id}_A \\ \alpha_1 &\mapsto f \\ \alpha_2 &\mapsto f \\ \beta &\mapsto g \end{aligned}$$

For this refinement system r, we have that $R_1, R_2 \sqsubset A$ and $S \sqsubset B$ and $T \sqsubset C$ in the sense of Defn. 3.2.4, that

$$R_1 \underset{A}{\Longrightarrow} R_2 \qquad R_1 \underset{f}{\Longrightarrow} S \qquad R_2 \underset{f}{\Longrightarrow} S \qquad S \underset{g}{\Longrightarrow} T \qquad S \underset{h}{\Longrightarrow} T$$

are typing (and subtyping) judgments in the sense of Defn. 3.2.5, and that

$$\overset{\gamma}{R_1 \underset{A}{\Longrightarrow} R_2} \qquad \overset{\alpha_1}{R_1 \underset{f}{\Longrightarrow} S} \qquad \overset{\alpha_2}{R_2 \underset{f}{\Longrightarrow} S} \qquad \overset{\beta}{S \underset{g}{\Longrightarrow} T}$$

are typing derivations in the sense of Defn. 3.2.6. Observe that the typing judgment $S \underset{h}{\Longrightarrow} T$ is not derivable in r. □

We will also find it helpful to adapt the proof-theoretic notation of inference rules. In general, we say that a typing rule

$$\frac{S_1 \underset{f_1}{\Longrightarrow} T_1 \quad \dots \quad S_n \underset{f_n}{\Longrightarrow} T_n}{S \underset{f}{\Longrightarrow} T}$$

is admissible for a type refinement system (in the sense of Defn. 3.2.3) if, given derivations of the premises (in the sense of Defn. 3.2.6), we can construct a derivation of the conclusion. (When we write down a rule, it is always left implicit that the typing judgments are well-formed in the sense of Defn. 3.2.5.)

Proposition 3.2.8. *The following rules are admissible for any refinement system:*

$$\frac{R \underset{f}{\Longrightarrow} S \quad S \underset{g}{\Longrightarrow} T}{R \underset{f;g}{\Longrightarrow} T}\ ; \qquad\qquad \frac{}{R \underset{A}{\Longrightarrow} R}\ \mathrm{id}$$

Proof. This is essentially a restatement of functoriality. For example, suppose α is a derivation of (R, f, S) and β is a derivation of (S, g, T). By definition, this means that $\alpha : R \to S$ and $r(\alpha) = f$, and $\beta : S \to T$ and $r(\beta) = g$. But then $(\alpha;\beta)$ is a derivation of $(R, (f;g), T)$, since $(\alpha;\beta) : R \to T$ and $r(\alpha;\beta) = (r(\alpha); r(\beta)) = (f;g)$. Similarly, assuming $R \sqsubset A$, then $\mathrm{id}_R : R \to R$ is a derivation of (R, id_A, R), since $r(\mathrm{id}_R) = \mathrm{id}_{r(R)} = \mathrm{id}_A$. □

Proposition 3.2.9. *Subtyping is reflexive and transitive, and admits rules of covariant and contravariant subsumption:*

$$\frac{}{R \underset{A}{\Longrightarrow} R} \qquad \frac{R \underset{A}{\Longrightarrow} S \quad S \underset{A}{\Longrightarrow} T}{R \underset{A}{\Longrightarrow} T} \qquad \frac{R \underset{f}{\Longrightarrow} S_1 \quad S_1 \underset{B}{\Longrightarrow} S_2}{R \underset{f}{\Longrightarrow} S_2} \qquad \frac{R_1 \underset{A}{\Longrightarrow} R_2 \quad R_2 \underset{g}{\Longrightarrow} T}{R_1 \underset{g}{\Longrightarrow} S}$$

Proof. Reflexivity of subtyping is by definition just another name for the id typing rule of Prop. 3.2.8, while transitivity and subsumption are all special cases of ";" with one or both of f and g set to the identity term. □

Remark. The proof of Prop. 3.2.9 illustrates that constructing a typing derivation sometimes involves reasoning about equality of terms (i.e., morphisms in $\mathcal{T}$). For clarity it is sometimes useful to mark applications of an equality by an explicit *conversion step*:

$$\frac{R \underset{f}{\Longrightarrow} S}{R \underset{g}{\Longrightarrow} S}\ \sim$$

For example, the covariant subsumption rule can be more explicitly derived as follows:

$$\cfrac{\cfrac{R \underset{f}{\Longrightarrow} S_1 \quad S_1 \underset{B}{\Longrightarrow} S_2}{R \underset{f;B}{\Longrightarrow} S_2}\ ;}{R \underset{f}{\Longrightarrow} S_2}\ \sim$$

□

Remark. Although we often use inference rule notation, we should keep in mind that this is just a notation, and that formally, a derivation of a typing judgment is just a particular kind of morphism in the category $\mathcal{D}$. In particular, the categorical axioms imply that various ways of composing inference rules must end up naming the same typing derivation. For example, the associativity axioms imply that

$$\dfrac{\dfrac{R \overset{\alpha}{\underset{f}{\Longrightarrow}} S \quad S \overset{\beta}{\underset{g}{\Longrightarrow}} T}{R \underset{f;g}{\Longrightarrow} T}\,; \qquad T \overset{\gamma}{\underset{h}{\Longrightarrow}} T'}{R \underset{(f;g);h}{\Longrightarrow} T'}\,; \quad = \quad \dfrac{R \overset{\alpha}{\underset{f}{\Longrightarrow}} S \qquad \dfrac{S \overset{\beta}{\underset{g}{\Longrightarrow}} T \quad T \overset{\gamma}{\underset{h}{\Longrightarrow}} T'}{S \underset{g;h}{\Longrightarrow} T'}\,;}{R \underset{f;(g;h)}{\Longrightarrow} T'}\,;$$

while the unit laws imply that

$$\dfrac{R \overset{\alpha}{\underset{f}{\Longrightarrow}} S \quad \dfrac{}{S \underset{B}{\Longrightarrow} S}\,\text{id}}{R \underset{f;B}{\Longrightarrow} S}\,; \quad = \quad R \overset{\alpha}{\underset{f}{\Longrightarrow}} S \quad = \quad \dfrac{\dfrac{}{R \underset{A}{\Longrightarrow} R}\,\text{id} \quad R \overset{\alpha}{\underset{f}{\Longrightarrow}} S}{R \underset{A;f}{\Longrightarrow} S}\,;$$

One case where checking such equations is particularly easy is when the type refinement system is **proof-irrelevant**, in the sense that there is at most one derivation of any typing judgment. The refinement system constructed in Example 3.2.7 happens to be proof-irrelevant, but many other refinement systems which we will be interested in are not. □

Remark. Finally, the definition of a type refinement system as an arbitrary functor does not, a priori, give us any hint of how to decide whether there exists a derivation of a given typing judgment, and in that sense it can be seen as an abstract specification of a "type assignment" system. For the most part, in this chapter we will avoid dealing directly with questions of type checking – although this is not to suggest that such questions are unimportant, and we should hope to eventually be able to speak about such issues within the framework. □

3.2.3 Some examples of type refinement systems

One of the nice things about having an abstract mathematical definition of a serious scientific concept is that you can come up with silly examples. The following pair of degenerate examples of type refinement systems may be silly, but they nonetheless come in handy sometimes, since they help in relating constructions on refinement systems back to standard constructions on categories.

Example 3.2.10. For any category $\mathcal{C}$, there is a refinement system $! : \mathcal{C} \to 1$ mapping every object and morphism of $\mathcal{C}$ to the unique object and morphism of the terminal category 1. In terms of the definitions of the previous section, this means that $A \sqsubset *$ for every object $A \in \mathcal{C}$ (where $*$ denotes the unique object of 1), and a (sub)typing judgment

$$A \underset{*}{\Longrightarrow} B$$

is derivable just in case there exists a morphism $A \to B$ in $\mathcal{C}$. □

Example 3.2.11. For any category C, there is a refinement system $\mathrm{id} : C \to C$ corresponding to the identity functor on C. This means that $A \sqsubset A$ for all objects A, and that *every* typing judgment

$$A \underset{f}{\Longrightarrow} B$$

has a derivation (provided, as usual, that it is well-formed), in particular given by f itself. □

Now let's discuss some actual reasonable examples.

Example 3.2.12. A natural class of examples of type refinement systems comes from the Floyd-Hoare approach to program verification. In abstract terms, these refinement systems can be described as follows:

- Take $\mathcal{T}$ to be a category with one object W (representing the state space) and a morphism $c : W \to W$ for every possible command (viewed as a state transformer), noting that we need to include the identity command ("skip") and sequential composition to ensure that we get a category.

- Take $\mathcal{D}$ to be a category whose objects P, Q are predicates over the state space, and whose morphisms $(c, \alpha) : P \to Q$ are pairs of a command c equipped with a verification α that it will take any state satisfying P to a state satisfying Q.

- Take $r : \mathcal{D} \to \mathcal{T}$ to be the evident forgetful functor, which sends every predicate P to the object W, and every "verified" command (c, α) to the underlying command c.

In this case, a typing judgment is nothing but a *Hoare triple* $\{P\}c\{Q\}$, while the composition typing rule and rules of co- and contravariant subsumption are nothing but the rules of *sequential composition*, of *post-weakening*, and of *pre-strengthening*, respectively:

$$\frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1;c_2\{R\}} \qquad \frac{\{P\}c\{Q_1\} \quad Q_1 \supset Q_2}{\{P\}c\{Q_2\}} \qquad \frac{P_1 \supset P_2 \quad \{P_2\}c\{Q\}}{\{P_1\}c\{Q\}}$$

This is really a class of examples because we can vary the collection of commands and the language of predicates to obtain different refinement systems with different properties, as we will see soon. It is also possible to adapt this example to *separation logic*, which we will get to later. □

Example 3.2.13. The previous example can also be seen from the following slightly different perspective. Any monoid M induces a category $\mathbf{B}M$ (called the *delooping* of M) with one object $*$ and a morphism $m : * \to *$ for every monoid element $m \in M$. Then if C is an arbitrary category, a functor $p : C \to \mathbf{B}M$ can be seen as a "weighting" of the morphisms of C by elements of m, and for any pair of objects $A, B \in C$, the typing judgment

$$A \underset{m}{\Longrightarrow} B$$

is derivable (possibly with multiple derivations) just in case there exists in C a morphism from A to B of weight m.

As another instance of this pattern, any finite state automaton (possibly nondeterministic and with ϵ-transitions) can be viewed as a refinement system $\delta : \mathcal{Q} \to \mathbf{B}\Sigma^*$, where $\mathbf{B}\Sigma^*$ is the one-object category corresponding to the monoid of words on an alphabet Σ, where $\mathcal{Q}$ is a category whose objects are states and whose morphisms correspond to transitions of the automaton, and where δ is the functor that assigns the corresponding input word to each transition of the machine. A typing judgment

$$q_1 \underset{w}{\Longrightarrow} q_2$$

is derivable (again, possibly with multiple derivations) just in case there is a transition from state q_1 to state q_2 on input w. In particular, a subtyping judgment $q_1 \underset{*}{\Longrightarrow} q_2$ is derivable iff there is an ϵ-transition from q_1 to q_2. □

Example 3.2.14. We already got a preview of another natural refinement system in Section 2.2.3, where we introduced the category **Subset** whose objects are pairs $(A, R \subseteq A)$, and whose morphisms $(A, R) \longrightarrow (B, S)$ are functions $f : A \to B$ such that $a \in R$ implies $f(a) \in S$ for all a. The evident projection functor **Subset** $\to$ **Set** then defines a refinement system which seems to capture the intuitive model people often have in mind when they talk about "refinement types", although we should be aware that it is just one model.

As also alluded to at the end of Section 2.2.3, this example has a straightforward generalization where we replace sets by *partially-ordered* sets and arbitrary subsets by *downwards-closed* subsets. The refinement system **Downset** $\to$ **Poset** is again defined by a projection functor, where **Poset** is the category of posets and order-preserving maps, and **Downset** is the category whose objects are pairs of a poset together with a downwards-closed subset of that poset, and with morphisms defined as for **Subset**. (Note that both of these examples are "large" in the category-theoretic sense, since **Set**, **Poset**, etc., are all large categories; such size issues will not be important for any of what we do, though.) □

Example 3.2.15. Finally, both **Subset** $\to$ **Set** and **Downset** $\to$ **Poset** can be further generalized by replacing (po)sets with categories and subsets by *presheaves*. The refinement system **Psh** $\to$ **Cat** is defined as follows:

- **Cat** is the (large) category whose objects are categories and morphisms are functors.
- Objects of **Psh** are pairs $(\mathcal{A}, \phi)$, where $\mathcal{A}$ is a category and $\phi : \mathcal{A}^{op} \to \mathbf{Set}$ is a contravariant presheaf over that category.
- Morphisms $(\mathcal{A}, \phi) \to (\mathcal{B}, \psi)$ of **Psh** are pairs (F, θ), where $F : \mathcal{A} \to \mathcal{B}$ is a functor and $\theta : \phi \Rightarrow (\psi \circ F^{op})$ is a natural transformation.
- **Psh** $\to$ **Cat** is the evident projection.

Note that whereas the refinement systems **Subset** $\to$ **Set** and **Downset** $\to$ **Poset** are proof-irrelevant, the refinement system **Psh** $\to$ **Cat** is not. □

And of course, the list goes on! There are many examples of functors in the world, and so far all that we've done is explain how to read the data of a functor in a type-theoretic accent. The actual motivation for looking at type refinement in this way is only really apparent after we've seen how to describe various natural properties of and constructions on type refinement systems as natural properties of and constructions on functors.

3.2.4 Pulling back and pushing forward

Suppose we've fixed some refinement system $r : \mathcal{D} \to \mathcal{T}$. Given a refinement $R \sqsubset A$ and a morphism $f : A \to B$, a natural question is whether there exists an $S \sqsubset B$ such that

$$R \underset{f}{\Longrightarrow} S$$

is derivable. Moreover, if there are several such S, we might hope for there to exist a "best" one, in the sense that it is a subtype of all the others. The definition of *pushforward* refinements is a categorical formalization of this concept.

Definition 3.2.16 (Pushforward refinements)**.** Let $R \sqsubset A$ and $f : A \to B$. A **pushforward of** R **along** f, when it exists, is a refinement $f_\diamond R \sqsubset B$

$$\frac{R \sqsubset A \quad f : A \to B}{f_\diamond R \sqsubset B}$$

equipped with a pair of typing rules

$$\frac{}{R \underset{f}{\Longrightarrow} f_\diamond R}\, f_\diamond I \qquad \frac{R \underset{f;g}{\Longrightarrow} S}{f_\diamond R \underset{g}{\Longrightarrow} S}\, f_\diamond E$$

satisfying a pair of equations on typing derivations

$$\dfrac{\dfrac{}{R \underset{f}{\Longrightarrow} f_\diamond R}\, f_\diamond I \quad \dfrac{\overset{\beta}{R \underset{f;g}{\Longrightarrow} S}}{f_\diamond R \underset{g}{\Longrightarrow} S}\, f_\diamond E}{R \underset{f;g}{\Longrightarrow} S}\; ; \quad = \quad \overset{\beta}{R \underset{f;g}{\Longrightarrow} S}$$

and

$$\overset{\eta}{f_\diamond R \underset{g}{\Longrightarrow} S} \quad = \quad \dfrac{\dfrac{\dfrac{}{R \underset{f}{\Longrightarrow} f_\diamond R}\, f_\diamond I \quad \overset{\eta}{f_\diamond R \underset{g}{\Longrightarrow} S}}{R \underset{f;g}{\Longrightarrow} S}\; ;}{f_\diamond R \underset{g}{\Longrightarrow} S}\, f_\diamond E$$

Evidently, there is also a dual concept of "pulling back".

Definition 3.2.17 (Pullback refinements)**.** Let $f : A \to B$ and $S \sqsubset B$. A **pullback of** S **along** f, when it exists, is a refinement $f^{\square} S \sqsubset A$

$$\frac{f : A \to B \quad S \sqsubset B}{f^{\square} S \sqsubset A}$$

equipped with a pair of typing rules

$$\frac{}{f^{\square} S \underset{f}{\Longrightarrow} S}\, f^{\square}E \qquad \frac{R \underset{g;f}{\Longrightarrow} S}{R \underset{g}{\Longrightarrow} f^{\square} S}\, f^{\square}I$$

satisfying the pair of equations on typing derivations

$$\cfrac{\cfrac{R \overset{\beta}{\underset{g;f}{\Longrightarrow}} S}{R \underset{g}{\Longrightarrow} f^{\square} S}\, f^{\square}I \qquad \cfrac{}{f^{\square} S \underset{f}{\Longrightarrow} S}\, f^{\square}E}{R \underset{g;f}{\Longrightarrow} S}\; ; \qquad = \quad R \overset{\beta}{\underset{g;f}{\Longrightarrow}} S$$

and

$$R \overset{\eta}{\underset{g}{\Longrightarrow}} f^{\square} S \quad = \quad \cfrac{\cfrac{R \overset{\eta}{\underset{g}{\Longrightarrow}} f^{\square} S \qquad \cfrac{}{f^{\square} S \underset{f}{\Longrightarrow} S}\, f^{\square}E}{R \underset{g;f}{\Longrightarrow} S}\; ;}{R \underset{g}{\Longrightarrow} f^{\square} S}\, f^{\square}I$$

Before moving on, it might be helpful to give a few examples, beginning with the most familiar.

Example 3.2.18. For the refinement system **Subset** $\to$ **Set**, the operations of pushing forward and pulling back correspond respectively to taking the image or inverse image of a subset along a function:

$$f_{\diamond}(A, R) = (B, f(R))$$
$$f^{\square}(B, S) = (A, f^{-1}(S))$$

Given the interpretation of typing judgments, the typing rules are essentially vacuous, while the equations hold trivially since the refinement system is proof-irrelevant. □

Example 3.2.19. For the class of "Hoare logic refinement systems" introduced in Example 3.2.12, a pushforward $c_{\diamond} P$ of a predicate along a command is essentially a *strongest postcondition* $c_{\diamond} P = sp(c, P)$, while a pullback $c^{\square} Q$ is essentially a *weakest precondition* $c^{\Box} Q = wp(c, Q)$.[1] Whether strongest postconditions and/or weakest preconditions exist depends, though, on the specific set of commands of predicates. For example, suppose that the language includes three primitive assignment statements

$$x := 0 \quad , \quad x := 1 \quad , \quad x := 1 - x$$

which are closed under sequential composition (to get a category), and that we are using a very restricted logic with just two predicates $[x = 0]$ and $[x = 1]$ and nothing else. In this situation, strongest postconditions are easy to compute, corresponding to forward execution:

$$\begin{aligned}
sp(x := 0, [x = 0]) = sp(x := 0, [x = 1]) &= [x = 0] \\
sp(x := 1, [x = 0]) = sp(x := 1, [x = 1] &= [x = 1] \\
sp(x := 1 - x, [x = 0]) &= [x = 1] \\
sp(x := 1 - x, [x = 1]) &= [x = 0] \\
sp(\mathrm{skip}, P) &= P \\
sp((c_1; c_2), P) &= sp(c_2, sp(c_1, P))
\end{aligned}$$

On the other hand, with this very limited language of predicates, weakest preconditions can only be computed in certain cases, namely for the $x := 1 - x$ command and its compositions:

$$\begin{aligned}
wp(x := 1 - x, [x = 0]) &= [x = 1] \\
wp(x := 1 - x, [x = 1]) &= [x = 0] \\
wp(\mathrm{skip}, P) &= P \\
wp((c_1; c_2), P) &= wp(c_1, wp(c_2, P))
\end{aligned}$$

Intuitively, the reason why we can't compute the others is because *backwards* execution of constant assignment statements is nondeterministic and potentially aborting. We can recover these "missing" weakest preconditions, though, by adding finite disjunctions to the language of predicates:

$$\begin{aligned}
wp(x := i, [x = i]) &= [x = 0] \vee [x = 1] \\
wp(x := i, [x = 1 - 1]) &= \bot
\end{aligned}$$

$\Box$

[1] Recall that a strongest postcondition is a predicate $sp(c, P)$ such that $\{P\}c\{sp(c, P)\}$, and such that for any other Q, if $\{P\}c\{Q\}$ then $sp(c, P) \supset Q$. Dually, a weakest precondition is a predicate $wp(c, Q)$ such that $\{wp(c, Q)\}c\{Q\}$, and such that for any other P, if $\{P\}c\{Q\}$ then $P \supset wp(c, Q)$. It is easy to see that these properties are implied by Definitions 3.2.16 and 3.2.17. The converse is almost true (we can safely ignore the equations by assuming proof-irrelevance), but not quite, in the sense that for the standard definition of, e.g., weakest preconditions, the triple $\{P\}d; c\{Q\}$ does not necessarily imply the triple $\{P\}d\{wp(c, Q)\}$ for general d. It is true in many situations, though, such as when the refinement system is a fibration (see below). (In the categorical literature, this issue comes up in the question of the precise definition of "cartesian morphism", with the standard definition of $wp(c, Q)$ corresponding to what is sometimes called "weakly cartesian".)

As we see, a given refinement system may or may not have pushforwards and/or pullbacks for all refinements along all morphisms. In the lucky situation that it does, it gets a special name:

Definition 3.2.20 (Fibrations et al.)**.** A refinement system (= functor) is said to be a **fibration** if there exists a pullback $(f^{\square} S, f^{\square} E, f^{\square} I)$ for every $f : A \to B$ and $S \sqsubset B$, an **opfibration** if there exists a pushforward $(f_{\diamond} R, f_{\diamond} E, f_{\diamond} I)$ for every $f : A \to B$ and $R \sqsubset A$, and a **bifibration** if it is both a fibration and an opfibration.

These definitions go back to Grothendieck, albeit in a slightly different notation (cf. [3, 18]). Part of the original motivation for the definitions is that fibrations and opfibrations can be interpreted as *indexed categories* in a certain sense that we'll review.

Definition 3.2.21 (Subcategories of refinements)**.** Given a refinement system $r : \mathcal{D} \to \mathcal{T}$ and type $A \in \mathcal{T}$, the **subcategory of refinements of** A is defined as the category $\mathcal{D}_A$ whose objects are refinements $R \sqsubset A$ and whose morphisms $R_1 \to R_2$ are subtyping derivations, i.e., morphisms $\alpha : R_1 \to R_2$ in $\mathcal{D}$ such that $r(\alpha) = \mathrm{id}_A$. (Note: the category $\mathcal{D}_A$ is often referred to as the *fiber* of A in the categorical literature.)

Definition 3.2.22 (Strong equivalence)**.** Let $R_1, R_2 \sqsubset A$ be two refinements of the same type. We say that R_1 and R_2 are **strongly equivalent** (written $R_1 \equiv R_2$) if they are isomorphic in the subcategory of refinements of A. In other words, R_1 is strongly equivalent to R_2 just in case there exist a pair of subtyping derivations $R_1 \overset{\alpha_1}{\underset{A}{\Longrightarrow}} R_2$ and $R_2 \overset{\alpha_2}{\underset{A}{\Longrightarrow}} R_1$ which compose to the identities on R_1 and R_2. (Note: a strong equivalence is sometimes referred to as a *vertical isomorphism* in the literature on fibrations.)

Exercise 3.2.23. Show that when $f_{\diamond} R$ and $f^{\square} S$ exist, they are determined uniquely up to strong equivalence.

Given a refinement system $r : \mathcal{D} \to \mathcal{T}$, consider the mapping

$$A \mapsto \mathcal{D}_A$$

which sends any type to its subcategory of refinements. When r is a fibration or opfibration, this mapping can be extended to a (contravariant or covariant) functor from $\mathcal{T}$ to **Cat**. More generally, the operations of pushing forward and pulling back have the following properties:

Proposition 3.2.24. *For any refinement system, whenever the corresponding pushforward and/or pullback refinements exist:*

1. *the following subtyping rules are admissible:*

$$\frac{R_1 \underset{A}{\Longrightarrow} R_2}{f_{\diamond} R_1 \underset{B}{\Longrightarrow} f_{\diamond} R_2} \leq_{f_{\diamond}} \qquad \frac{S_1 \underset{B}{\Longrightarrow} S_2}{f^{\square} S_1 \underset{A}{\Longrightarrow} f^{\square} S_2} \leq_{f^{\square}}$$

2. *there are strong equivalences*

$$(g \circ f)_\diamond R \equiv g_\diamond f_\diamond R \qquad (g;f)^\square S \equiv g^\square f^\square S$$

$$\mathrm{id}_\diamond R \equiv R \qquad \mathrm{id}^\square S \equiv S$$

Proof. For (1), here is a very explicit derivation of the subtyping rule for pushforwards (the one for pullbacks can be derived symmetrically):

$$\dfrac{\dfrac{\dfrac{R_1 \underset{A}{\Longrightarrow} R_2 \quad \dfrac{}{R_2 \underset{f}{\Longrightarrow} f_\diamond R_2}\,f_\diamond I}{R_1 \underset{\mathrm{id}_A;f}{\Longrightarrow} f_\diamond R_2}\,;}{R_1 \underset{f;\mathrm{id}_B}{\Longrightarrow} f_\diamond R_2}\,\sim}{f_\diamond R_1 \underset{B}{\Longrightarrow} f_\diamond R_2}\,f_\diamond E$$

For (2), to show $(g;f)^\square S \equiv g^\square f^\square S$ for instance, we begin by exhibiting subtyping derivations in both directions:

$$\dfrac{\dfrac{\dfrac{}{(g;f)^\square S \underset{g;f}{\Longrightarrow} S}\,(g;f)^\square E}{(g;f)^\square S \underset{g}{\Longrightarrow} f^\square S}\,f^\square I}{(g;f)^\square S \Longrightarrow g^\square f^\square S}\,g^\square I \qquad \dfrac{\dfrac{\dfrac{}{g^\square f^\square S \underset{g}{\Longrightarrow} f^\square S}\,g^\square E \quad \dfrac{}{f^\square S \underset{f}{\Longrightarrow} S}\,f^\square E}{g^\square f^\square S \underset{g;f}{\Longrightarrow} S}\,;}{g^\square f^\square S \Longrightarrow (g;f)^\square S}\,(g;f)^\square I$$

Then we must also show that these two subtyping derivations compose to a pair of identities, which we can do using the equations from Defn. 3.2.17. □

Part (1) of Prop. 3.2.24 is almost enough to establish that in a refinement system which is an opfibration and/or fibration, every morphism $f : A \to B$ induces functors

$$f_\diamond : \mathcal{D}_A \to \mathcal{D}_B \qquad \text{and/or} \qquad f^\square : \mathcal{D}_B \to \mathcal{D}_A$$

between the corresponding subcategories of refinements. To guarantee that these are really functors, though, we must also check that they preserve identity and composition.

Exercise 3.2.25. Express the property that the functors $f_\diamond$ and $f^\square$ preserve identity and composition as equations on subtyping derivations constructed using $\leq_{f_\diamond}$ and $\leq_{f^\square}$ and identity and composition. Prove that these equations follow from Definitions 3.2.16 and 3.2.17.

Assuming this property, part (2) of Prop. 3.2.24 then implies that when $r : \mathcal{D} \to \mathcal{T}$ is an opfibration, the mapping $A \mapsto \mathcal{D}_A$ can be extended via the pushforward operation to a functor $\mathcal{T} \to$ **Cat**, while in the case that $r : \mathcal{D} \to \mathcal{T}$ is a fibration, the mapping can be extended via the pullback operation to a functor $\mathcal{T}^{op} \to$ **Cat**. (Technically, these functors to **Cat** are actually called "pseudofunctors", since preservation of composition and identity is only guaranteed up to strong equivalence.)

These properties also have a sort of converse, which is known as the "Grothendieck construction". Given any (pseudo)functor

$$F : \mathcal{T} \to \mathbf{Cat}$$

one can construct an opfibration

$$\pi_F : \int F \to \mathcal{T}$$

where $\int F$ is the category whose objects are pairs (A, R) of an object $A \in \mathcal{T}$ together with an object $R \in F(A)$, and whose morphisms $(A, R) \to (B, S)$ are pairs (f, α) of a morphism $f : A \to B$ in $\mathcal{T}$ together with a morphism $\alpha : F(f)(R) \to S$ in $F(B)$. The evident projection functor π_F is an opfibration, with pushforward refinements defined by the functorial action of F:

$$f_\diamond R = F(f)(R)$$

Dually, any (pseudo)functor

$$G : \mathcal{T}^{op} \to \mathbf{Cat}$$

gives rise to a fibration

$$\pi_G : \int^{op} G \to \mathcal{T}$$

where $\int^{op} G$ is the category whose objects again are pairs (A, R) of an object $A \in \mathcal{T}$ together with an object $R \in G(A)$, but now with the morphisms $(A, R) \to (B, S)$ being pairs (f, α) of a morphism $f : A \to B$ in $\mathcal{T}$ together with a morphism $\alpha : R \to G(f)(S)$ in $G(A)$. This time, the functorial action of G is used to define pullback refinements:

$$f^\square S = G(f)(S)$$

Remark. If we view functors as type refinement systems, the relationship between (op)fibrations and indexed categories can be interpreted as expressing a relationship between typing and subtyping. In a sense the Grothendieck construction says that each can be reduced to the other in the presence of sufficient pullbacks and/or pushforwards. Still, there are good reasons to give precedence to typing, and from Chapter 2 we are already familiar with some advantages of reducing subtyping to typing of the identity (see Sections 2.2.2 and 2.3.3). One disadvantage of going the other way around and reducing typing to subtyping (via either the covariant or the contravariant versions of the Grothendieck constructions) is that it forces a particular orientation on the original refinement system. This is especially flagrant in the case of a *bifibration*, where we have a three-way correspondence

$$\dfrac{\dfrac{f_\diamond R \underset{B}{\Longrightarrow} S}{R \underset{f}{\Longrightarrow} S}}{R \underset{A}{\Longrightarrow} f^\square S}$$

but it seems that none of the judgments should be privileged over the others. □

Example 3.2.26. The two trivial examples of refinement systems $! : C \to 1$ and $\mathrm{id} : C \to C$ (Examples 3.2.10 and 3.2.11) are also trivial examples of bifibrations. In the former case, there is nothing to push or pull along except for the identity on $*$ (and these must correspond to identity operations by Prop. 3.2.24), while in the latter case, pulling or pushing along a morphism $f : A \to B$ just returns the source/target of f:

$$f_\diamond A = B \qquad f^\square B = A$$

□

Example 3.2.27. Like **Subset** → **Set**, the refinement system **Downset** → **Poset** is also a bifibration, albeit with a more asymmetrical interpretation of pushforward and pullback. Let's write R as a convenient abuse-of-notation for the pair (A, R) of a (po)set A equipped with a (downwards-closed) subset $R \subseteq A$, leaving the A implicit. As already mentioned, in **Subset** → **Set** pushforward and pullbacks are computed using image and inverse image, equivalently written as follows:

$$\begin{aligned} f^\square S &= \{ a \mid f(a) \in S \} \\ f_\diamond R &= \{ f(a) \mid a \in R \} \end{aligned}$$

The definitions in **Downset** → **Poset** are almost identical, except that we have to take the *downwards-closure* of the image:

$$\begin{aligned} f^\square S &= \{ a \mid f(a) \in S \} \\ f_\diamond R &= \{ b \mid \exists a.\, b \leq_B f(a) \wedge a \in R \} \end{aligned}$$

The reason for the different treatment is that the inverse image of a downwards-closed subset along an order-preserving function is always downwards-closed, but the image is not necessarily. □

Example 3.2.28. Similarly, the refinement system **Psh** → **Cat** (Example 3.2.15) is also a bifibration, where the definitions of pullback and pushforward can be seen as a categorical generalization of the definitions for **Downset** → **Poset**. In particular, the pullback of a presheaf $\psi : \mathcal{B}^{op} \to \mathbf{Set}$ along a functor $F : \mathcal{A} \to \mathcal{B}$ is defined via precomposition:

$$\begin{aligned} F^\square \psi &: \mathcal{A}^{op} \to \mathbf{Set} \\ F^\square \psi &= a \mapsto \psi(Fa) \end{aligned}$$

And the pushforward of a presheaf $\phi : \mathcal{A}^{op} \to \mathbf{Set}$ along a functor $F : \mathcal{A} \to \mathcal{B}$ is defined as a *coend* [19]:

$$\begin{aligned} F_\diamond \phi &: \mathcal{B}^{op} \to \mathbf{Set} \\ F_\diamond \phi &= b \mapsto \int^a B(b, Fa) \times \phi(a) \end{aligned}$$

□

Finally, it is worth mentioning a relational generalization of **Subset** $\to$ **Set**.[2]

EXAMPLE 3.2.29. Let **Rel** be the category whose objects are sets A, B and whose morphisms $A \to B$ are relations between A and B. The composition of two relations

$$\begin{tikzcd} A \arrow[r, "M"] & B \arrow[r, "N"] & C \end{tikzcd}$$

is defined by relational composition

$$a\,(M;N)\,c \quad \Leftrightarrow \quad \exists b.\, a\,M\,b \wedge b\,N\,c$$

while the identities

$$\begin{tikzcd} A \arrow[r, "\mathrm{id}_A"] & A \end{tikzcd}$$

are given by the equality relation:

$$a_1\,\mathrm{id}_A\,a_2 \quad \Leftrightarrow \quad a_1 = a_2$$

Next let $\mathbf{Rel}_\bullet$ be the category whose objects are pairs $(A, R \subseteq A)$ and whose morphisms $(A, R) \to (B, S)$ are relations $M : A \to B$ such that $a \in R$ and $a\,M\,b$ implies $b \in S$ for all a and b. (The name "$\mathbf{Rel}_\bullet$" is because a subset $R \subseteq A$ is the same thing as a morphism $R : 1 \to A$ in **Rel**, i.e., $\mathbf{Rel}_\bullet$ can be seen as a "pointed" version of **Rel**.) Interestingly, $\mathbf{Rel}_\bullet \to \mathbf{Rel}$ is a bifibration where the pushforward and pullback are computed respectively by existential and universal quantification along a relation:

$$\begin{aligned} M_\diamond\,R &= \{\, b \mid \exists a.\, a\,M\,b \wedge a \in R \,\} \\ M^\square\,S &= \{\, a \mid \forall b.\, a\,M\,b \Rightarrow b \in S \,\} \end{aligned}$$

□

Exercise 3.2.30. In proof theory, an inference rule is said to be *invertible* if given a derivation of the conclusion, one can obtain a derivation of the premises. Explain how the $f_\diamond E$ and $f^\square I$ rules are invertible in the stronger sense that they witness one half of an *isomorphism* between derivations of the premise and derivations of the conclusion. Use this to prove that in a bifibration, the functors $f_\diamond : \mathcal{D}_A \to \mathcal{D}_B$ and $f^\square : \mathcal{D}_B \to \mathcal{D}_A$ form an adjoint pair $f_\diamond \dashv f^\square$.

Exercise 3.2.31. Let $\delta : Q \to B\Sigma^*$ be the refinement system associated to a finite state automaton (see Example 3.2.13). In what situations is δ an opfibration? In what situations is it a bifibration?

[2]This example comes from [29]. Be warned that there is also another well-known "relational generalization" of **Subset** $\to$ **Set**, corresponding to the pullback of **Subset** $\to$ **Set** along the cartesian product functor **Set** $\times$ **Set** $\to$ **Set**. That gives rise to another bifibration $\mathbf{Subset}_2 \to \mathbf{Set} \times \mathbf{Set}$, where $\mathbf{Subset}_2$ has as objects triples $(A, B, R \subseteq A \times B)$, and morphisms $(A, B, R) \to (C, D, S)$ given by a pair of functions $f : A \to C$, $g : B \to D$ such that $R(a, b)$ implies $S(fa, gb)$ for all a and b. Keep in mind that $\mathbf{Subset}_2$ and **Rel** are very different categories: the former has relations as objects, the latter has relations as morphisms.

3.2.5 Intersection and union types

Now that we've put some effort into understanding the operations of a bifibration, we can take another look at old friends from Chapter 2. It turns out that we can define the concept of intersection and union type refinements in a modular way for any refinement system.

Definition 3.2.32 (Union and bottom refinements)**.** Let $R_1, R_2 \sqsubset A$. A **union** of R_1 and R_2, when it exists, is a refinement $R_1 \vee R_2 \sqsubset A$

$$\frac{R_1 \sqsubset A \quad R_2 \sqsubset A}{R_1 \vee R_2 \sqsubset A}$$

equipped with typing rules

$$\frac{R_1 \underset{f}{\Longrightarrow} S \quad R_2 \underset{f}{\Longrightarrow} S}{R_1 \vee R_2 \underset{f}{\Longrightarrow} S}\,\vee E \qquad \frac{}{R_i \underset{A}{\Longrightarrow} R_1 \vee R_2}\,\vee R_i$$

satisfying the equation

$$\frac{\dfrac{}{R_i \underset{f}{\Longrightarrow} R_1 \vee R_2}\,\vee R_i \quad \dfrac{R_1 \overset{\beta_1}{\underset{f}{\Longrightarrow}} S \quad R_2 \overset{\beta_2}{\underset{f}{\Longrightarrow}} S}{R_1 \vee R_2 \underset{f}{\Longrightarrow} S}\,\vee E}{R_i \underset{f}{\Longrightarrow} S}\,; \quad = \quad R_i \overset{\beta_i}{\underset{f}{\Longrightarrow}} S$$

(for each $i \in \{1, 2\}$) as well as the equation

$$R_1 \vee R_2 \overset{\eta}{\underset{f}{\Longrightarrow}} S \;=\; \frac{\dfrac{R_1 \vee R_2 \overset{\eta}{\underset{f}{\Longrightarrow}} S \quad \dfrac{}{R_1 \underset{f}{\Longrightarrow} R_1 \vee R_2}\,\vee R_1}{R_1 \underset{f}{\Longrightarrow} S}\,; \quad \dfrac{R_1 \vee R_2 \overset{\eta}{\underset{f}{\Longrightarrow}} S \quad \dfrac{}{R_2 \underset{f}{\Longrightarrow} R_1 \vee R_2}\,\vee R_2}{R_2 \underset{f}{\Longrightarrow} S}\,;}{R_1 \vee R_2 \underset{f}{\Longrightarrow} S}\,\vee E$$

Similarly, a **bottom refinement** for a type A is a refinement $\bot_A \sqsubset A$

$$\frac{}{\bot_A \sqsubset A}$$

equipped with a typing rule

$$\frac{}{\bot_A \underset{f}{\Longrightarrow} S}\,\bot E$$

satisfying the equation

$$\bot_A \overset{\eta}{\underset{f}{\Longrightarrow}} S \;=\; \frac{}{\bot_A \underset{f}{\Longrightarrow} S}\,\bot E$$

Definition 3.2.33 (Intersection and top refinements). Let $R_1, R_2 \sqsubset A$. An **intersection** of R_1 and R_2, when it exists, is a refinement $R_1 \wedge R_2 \sqsubset A$

$$\frac{R_1 \sqsubset A \quad R_2 \sqsubset A}{R_1 \wedge R_2 \sqsubset A}$$

equipped with typing rules

$$\frac{}{R_1 \wedge R_2 \underset{A}{\Longrightarrow} R_i}\ \wedge L_i \qquad \frac{S \underset{f}{\Longrightarrow} R_1 \quad S \underset{f}{\Longrightarrow} R_2}{S \underset{f}{\Longrightarrow} R_1 \wedge R_2}\ \wedge I$$

satisfying the equation

$$\frac{\dfrac{S \overset{\beta_1}{\underset{f}{\Longrightarrow}} R_1 \quad S \overset{\beta_2}{\underset{f}{\Longrightarrow}} R_2}{S \underset{f}{\Longrightarrow} R_1 \wedge R_2}\ \wedge I \qquad \dfrac{}{R_1 \wedge R_2 \underset{f}{\Longrightarrow} R_i}\ \wedge L_i}{S \underset{f}{\Longrightarrow} R_i}\ ; \quad = \quad S \overset{\beta_i}{\underset{f}{\Longrightarrow}} R_i$$

and the equation

$$S \overset{\eta}{\underset{f}{\Longrightarrow}} R_1 \wedge R_2 \ = \ \frac{\dfrac{S \overset{\eta}{\underset{f}{\Longrightarrow}} R_1 \wedge R_2 \quad \dfrac{}{R_1 \wedge R_2 \underset{f}{\Longrightarrow} R_1}\ \wedge L_1}{S \underset{f}{\Longrightarrow} R_1}\ ; \qquad \dfrac{S \overset{\eta}{\underset{f}{\Longrightarrow}} R_1 \wedge R_2 \quad \dfrac{}{R_1 \wedge R_2 \underset{f}{\Longrightarrow} R_2}\ \wedge L_2}{S \underset{f}{\Longrightarrow} R_2}\ ;}{S \underset{f}{\Longrightarrow} R_1 \wedge R_2}\ \wedge I$$

Similarly, a **top refinement** for a type A is a refinement $\top_A \sqsubset A$

$$\frac{}{\top_A \sqsubset A}$$

equipped with a typing rule

$$\frac{}{S \underset{f}{\Longrightarrow} \top_A}\ \top I$$

satisfying the equation

$$S \overset{\eta}{\underset{f}{\Longrightarrow}} \top_A \ = \ \frac{}{S \underset{f}{\Longrightarrow} \top_A}\ \top I$$

Definition 3.2.34 (Finite intersections and unions). A refinement system is said to have **finite intersections** if there exists an intersection refinement $R \wedge S$ for all $R, S \sqsubset A$, as well as a top refinement $\top_A$ for all A. Dually it is said to have **finite unions** if there exists a union refinement $R \vee S$ for all $R, S \sqsubset A$, as well as a bottom refinement $\perp_A$ for all A. (Note: intersections and unions are essentially what are called *fibered products and coproducts* in the literature on fibrations, although our definition is more general since it applies to any functor, not just to fibrations.)

This definition of intersection and union refinements is very reminiscent of the refinement and typing rules we considered in Chapter 2 while defining $\lambda^{\leq\wedge}_{\rightarrow}$, but a key difference is that here we can apply them to reason about intersections and unions in *any* type refinement system, not just one particular inductively-defined system. The equations on typing derivations are less familiar, but are needed if we want to talk about intersections and unions in proof-relevant refinement systems.

Example 3.2.35. The refinement system **Subset** $\to$ **Set** has finite intersections and unions, computed (of course) as intersections and unions of subsets. $\square$

Example 3.2.36. In **Psh** $\to$ **Cat**, finite intersections and unions are computed "pointwise", using the cartesian product and disjoint union of sets. Like so (for any $\phi, \psi : \mathcal{A}^{op} \to$ **Set**):

$$\begin{aligned}
\phi \wedge \psi &= a \mapsto \phi(a) \times \psi(a) \\
\phi \vee \psi &= a \mapsto \phi(a) + \psi(a) \\
\top_{\mathcal{A}} &= a \mapsto \{*\} \\
\bot_{\mathcal{A}} &= a \mapsto \emptyset
\end{aligned}$$

$\square$

Exercise 3.2.37. Prove that pullbacks preserve intersections and pushforwards preserve unions, in the sense that there are strong equivalences

$$f_\diamond(R \vee S) \equiv f_\diamond R \vee f_\diamond S \tag{3.1}$$

$$g^\square(R \wedge S) \equiv g^\square R \wedge g^\square S \tag{3.2}$$

whenever the corresponding refinements exist.

Exercise 3.2.38. On the other hand, pushforwards need not preserve intersections and pullbacks need not preserve unions, in the sense that in general the following subtyping judgments are only derivable in the left-to-right direction:

$$f_\diamond(R \wedge S) \Longrightarrow f_\diamond R \wedge f_\diamond S \tag{3.3}$$

$$g^\square R \vee g^\square S \Longrightarrow g^\square(R \vee S) \tag{3.4}$$

Give explicit counterexamples to each of the right-to-left versions of (3.3) and (3.4) in some concrete bifibration with intersections and unions. (Hint: is it possible to find counterexamples to both converses in **Subset** $\to$ **Set**?)

Exercise 3.2.39. When do the trivial refinement systems $! : C \to 1$ and $\mathrm{id} : C \to C$ have finite intersections and unions?

3.2.6 Morphisms and adjunctions of refinement systems

Since we've already considered a variety of examples of different refinement systems, it might be time to put on our category theory hats and think a bit about what it means to have a morphism *between* refinement systems.

Definition 3.2.40 (Morphisms of refinement systems)**.** For any pair of refinement systems $r : \mathcal{D} \to \mathcal{T}$ and $p : \mathcal{E} \to \mathcal{B}$, a **morphism of refinement systems** from r to p consists of a pair $F = (F_{\mathcal{D}}, F_{\mathcal{T}})$ of functors $F_{\mathcal{D}} : \mathcal{D} \to \mathcal{E}$ and $F_{\mathcal{T}} : \mathcal{T} \to \mathcal{B}$ such $p \circ F_{\mathcal{D}} = F_{\mathcal{T}} \circ r$, i.e., such that the square

$$\begin{tikzcd}
\mathcal{D} \arrow[r, "F_{\mathcal{D}}"] \arrow[d, "r"'] & \mathcal{E} \arrow[d, "p"] \\
\mathcal{T} \arrow[r, "F_{\mathcal{T}}"'] & \mathcal{B}
\end{tikzcd}$$

commutes strictly.

If we keep up with our notational conventions, then this definition can be unraveled as saying that a morphism of refinement systems $F = (F_{\mathcal{D}}, F_{\mathcal{T}}) : r \to p$ consists of a refinement formation rule

$$\frac{R \sqsubset A}{F(R) \sqsubset F(A)}$$

transporting r-refinements to p-refinements, as well as a typing rule

$$\dfrac{R \underset{f}{\Longrightarrow} S}{F(R) \underset{F(f)}{\Longrightarrow} F(S)}\ F$$

transporting derivations of r-judgments to derivations of p-judgments, satisfying the functorial axioms:

$$\dfrac{\dfrac{}{R \underset{\mathrm{id}_A}{\Longrightarrow} R}\ \mathrm{id}}{F(R) \underset{F(\mathrm{id}_A)}{\Longrightarrow} F(R)}\ F \quad = \quad \dfrac{}{F(R) \underset{\mathrm{id}_{F(A)}}{\Longrightarrow} F(R)}\ \mathrm{id}$$

$$\dfrac{\dfrac{R \overset{\alpha}{\underset{f}{\Longrightarrow}} S \quad S \overset{\beta}{\underset{g}{\Longrightarrow}} T}{R \underset{f;g}{\Longrightarrow} T}\ ;}{F(R) \underset{F(f;g)}{\Longrightarrow} F(T)}\ F \quad = \quad \dfrac{\dfrac{R \overset{\alpha}{\underset{f}{\Longrightarrow}} S}{F(R) \underset{F(f)}{\Longrightarrow} F(S)}\ F \quad \dfrac{S \overset{\beta}{\underset{g}{\Longrightarrow}} T}{F(S) \underset{F(g)}{\Longrightarrow} F(T)}\ F}{F(R) \underset{F(f);F(g)}{\Longrightarrow} F(T)}\ ;$$

It's worth emphasizing that in the general definition of a morphism of refinement systems we do *not* ask that F preserves all of the additional structure that r may happen to have (such as, say, pullbacks or intersections), although there are special situations where it will.

One situation when F *must* preserve some logical structure is when it has a left or right adjoint. By an **adjunction of refinement systems**, we simply mean a picture like this

$$\begin{tikzcd}
\mathcal{D} \arrow[r, bend left, "F_{\mathcal{D}}"] \arrow[r, phantom, "\perp"] \arrow[d, "r"'] & \mathcal{E} \arrow[l, bend left, "G_{\mathcal{D}}"] \arrow[d, "p"] \\
\mathcal{T} \arrow[r, bend left, "F_{\mathcal{T}}"] \arrow[r, phantom, "\perp"] & \mathcal{B} \arrow[l, bend left, "G_{\mathcal{T}}"]
\end{tikzcd}$$

where we have a pair of morphisms of refinement systems $F = (F_{\mathcal{D}}, F_{\mathcal{T}}) : r \to p$ and $G = (G_{\mathcal{D}}, G_{\mathcal{T}}) : p \to r$, together with a pair of adjunctions of categories $F_{\mathcal{D}} \dashv G_{\mathcal{D}}$ and $F_{\mathcal{T}} \dashv G_{\mathcal{T}}$, such that the unit and counit of the adjunction $F_{\mathcal{D}} \dashv G_{\mathcal{D}}$ are mapped by r and p onto the unit and counit of $F_{\mathcal{T}} \dashv G_{\mathcal{T}}$. A generalization of the famous "RAPL" principle for adjunctions of categories (right adjoints preserve limits) also holds for adjunctions of refinement systems.

Proposition 3.2.41 (Cf. [14, 28]). *If $F \dashv G : p \to r$ is an adjunction of refinement systems, then F preserves pushforwards and finite unions, while G preserves pullbacks and finite intersections.*

A different situation where a morphism of refinement systems $F : r \to p$ typically *should* preserve some of the logical structure is when we view F as building a "model" of r in p. As an example of this situation, if we let $r : \mathcal{D} \to \mathcal{T}$ be a member of the class of Hoare logic refinement systems (Examples 3.2.12 and 3.2.19), then a morphism of refinement systems into **Subset** $\to$ **Set**

$$\begin{tikzcd}
\mathcal{D} \arrow[r, "{[\![-]\!]}"] \arrow[d, "r"'] & \mathbf{Subset} \arrow[d] \\
\mathcal{T} \arrow[r, "{[\![-]\!]}"'] & \mathbf{Set}
\end{tikzcd}$$

gives a deterministic, set-theoretic semantics, where the functor $[\![-]\!] : \mathcal{T} \to \mathbf{Set}$ sends the unique object W to some fixed set of states $[\![W]\!]$ and commands c to different endofunctions on $[\![W]\!]$, while the functor $[\![-]\!] : \mathcal{D} \to \mathbf{Subset}$ sends predicates $P \sqsubset W$ to different subsets of $[\![W]\!]$. Similarly, a morphism from r into $\mathbf{Rel}_\bullet \to \mathbf{Rel}$ gives a potentially non-deterministic semantics, where commands are interpreted as different relations on $[\![W]\!]$. In both of these situations, if r has intersections and unions, then it is reasonable to ask that $[\![-]\!]$ preserves them, in other words that it interprets them by set-theoretic intersection and union:

$$[\![P \wedge Q]\!] = [\![P]\!] \cap [\![Q]\!] \qquad [\![P \vee Q]\!] = [\![P]\!] \cup [\![Q]\!]$$

3.3 Monoidal closed refinement systems

We began this chapter by modelling type refinement systems as functors $r : \mathcal{D} \to \mathcal{T}$ between arbitrary categories $\mathcal{D}$ and $\mathcal{T}$. We recovered additional logical structure on $\mathcal{D}$ by asking whether the functor $r : \mathcal{D} \to \mathcal{T}$ happens to satisfy some additional properties (such as the property of being a bifibration, or of having finite unions and intersections), but we did not ask for any additional logical structure on the base category $\mathcal{T}$ itself. On the other hand, we opened Chapter 2 by considering a very primitive type refinement system $\lambda^{\leq}_{\to}$ where the logical structure of refinement types perfectly mirrored the logical structure of the underlying types, with the only additional information coming from the atoms. One way of describing that situation in very abstract terms is that we had a pair of categories $\mathcal{D}$ and $\mathcal{T}$ carrying some additional algebraic structure, together with a *homomorphic* erasure functor $r : \mathcal{D} \to \mathcal{T}$, that is, one which strictly preserves the algebraic structure.

In this section we look at monoidal closed refinement systems, defined as strict monoidal closed functors between monoidal closed categories. Our main reason for starting from *monoidal* closed categories rather than *cartesian* closed categories is that this captures many additional models, including refinement systems such as $\mathbf{Rel}_\bullet \to \mathbf{Rel}$ which include aspects of linearity and side-effects. Moreover, it turns out that the combination of the connectives of a monoidal closed refinement system ($\otimes$, $\multimap$, and $\multimapinv$) with the connectives of a bifibration ($f_\diamond$ and $f^\square$) already leads to some very rich logical interactions. At the end of the chapter we'll consider a few applications, including a logical decomposition of the connectives of separation logic, as well as some perspective on realizability semantics and type refinement systems built over "uni-typed" languages like the pure lambda calculus.

3.3.1 Monoidal, symmetric monoidal, and cartesian closed refinement systems

Recall (and see Mac Lane [25] or Wikipedia or the nLab for details) that a category is *monoidal* if it is equipped with a tensor product and unit operation

$$\otimes : C \times C \to C \qquad I : 1 \to C$$

which are associative and unital up to coherent isomorphism. A monoidal category is *symmetric* when the monoidal product is commutative in the strong sense that there are a coherent family of isomorphisms

$$\gamma_{X,Y} : X \otimes Y \xrightarrow{\sim} Y \otimes X$$

satisfying the identity $(\gamma_{X,Y} ; \gamma_{Y,X}) = \mathrm{id}_{X \otimes Y}$, and it is *cartesian* when the monoidal product coincides with the categorical product, this being equivalent to the existence of a family of *duplication* and *erasure* maps,

$$\delta_X : X \to X \otimes X \qquad !_X : X \to I$$

satisfying a pair of reasonable identities which say that duplication followed by erasure is the identity.

A monoidal category is said to be *closed* when it is additionally equipped with left and right residuation operations (also called left/right *implication*)

$$\multimap : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C} \qquad \multimapinv : \mathcal{C} \times \mathcal{C}^{op} \to \mathcal{C}$$

which are right adjoint to tensor product in each component:

$$\mathcal{C}(Y, X \multimap Z) \cong \mathcal{C}(X \otimes Y, Z) \cong \mathcal{C}(X, Z \multimapinv Y)$$

In the case that the tensor product is symmetric the two residuals coincide $X \multimap Y \cong Y \multimapinv X$, and in the case that it is cartesian both residuals coincide with *exponential objects* $X \multimap Y \cong Y \multimapinv X \cong Y^X$.

Definition 3.3.1 (Monoidal refinement systems)**.** A refinement system $r : \mathcal{D} \to \mathcal{T}$ is said to be **(symmetric/cartesian) monoidal (closed)** if $\mathcal{D}$ and $\mathcal{T}$ are (symmetric/cartesian) monoidal (closed) categories, and the functor r strictly preserves the (symmetric/cartesian) monoidal (closed) structure.

Again, we can try to unpack this definition using the conventions of Section 3.2. In the case of a symmetric monoidal closed refinement system, the definition implies that there are refinement and typing rules for the tensor product and unit,

$$\frac{R_1 \sqsubset A_1 \quad R_2 \sqsubset A_2}{R_1 \otimes R_2 \sqsubset A_1 \otimes A_2} \qquad \frac{}{I \sqsubset I} \qquad \frac{R_1 \underset{f_1}{\Longrightarrow} S_1 \quad R_2 \underset{f_2}{\Longrightarrow} S_2}{R_1 \otimes R_2 \underset{f_1 \otimes f_2}{\Longrightarrow} S_1 \otimes S_2}\otimes \qquad \frac{}{I \underset{I}{\Longrightarrow} I}\, I$$

as well as refinement and typing rules for the implication,

$$\frac{R \sqsubset A \quad T \sqsubset C}{R \multimap T \sqsubset A \multimap C} \qquad \frac{}{R \otimes (R \multimap T) \underset{eval}{\Longrightarrow} T}\, eval \qquad \frac{R \otimes S \underset{m}{\Longrightarrow} T}{S \underset{curry(m)}{\Longrightarrow} R \multimap T}\, curry$$

as well as typing rules for the symmetry,

$$\frac{}{R \otimes S \underset{\gamma_{A,B}}{\Longrightarrow} S \otimes R}\, \gamma$$

and such that all of these collectively satisfy a host of equations on typing derivations, corresponding to the axioms of a symmetric monoidal closed category.

Example 3.3.2. The refinement system **Subset** $\to$ **Set** is cartesian closed, as is **Downset** $\to$ **Poset**. The usual cartesian closed structure on **Set** defines the tensor product $A_1 \otimes A_2$ of sets A_1 and A_2 as their cartesian product $A_1 \times A_2$, the unit as the singleton set $1 = \{*\}$, and the residual $A \multimap B$ of B by A as the set B^A of functions from A to B. This extends to a cartesian closed structure on **Subset** by taking

$$(A_1, R_1) \otimes (A_2, R_2) \overset{\text{def}}{=} (A_1 \times A_2, R_1 \times R_2)$$
$$I \overset{\text{def}}{=} (1, 1)$$
$$(A, R) \multimap (B, S) \overset{\text{def}}{=} (B^A, R \to S)$$

where the subsets $R_1 \times R_2 \subseteq A_1 \times A_2$, $1 \subset 1$, and $R \to S \subseteq B^A$ are defined by

$$
\begin{aligned}
(a_1, a_2) \in R_1 \times R_2 &\Leftrightarrow a_1 \in R_1 \wedge a_2 \in R_2 \\
* \in 1 &\Leftrightarrow \text{true} \\
f \in R \to S &\Leftrightarrow \forall a.\, a \in R \Rightarrow f(a) \in S
\end{aligned}
$$

With these definitions, the projection functor $(A, R) \mapsto A$ manifestly preserves the cartesian closed structure. Moreover, essentially the same definitions work for **Downset** $\to$ **Poset**, except that now B^A denotes the set of order-preserving functions from A to B, with the pointwise ordering. □

Example 3.3.3. The refinement system **Psh** $\to$ **Cat** is also cartesian closed, with the cc structure on **Cat** given by forming product categories and functor categories, and the cc structure on **Psh** defined as follows:

$$
\begin{aligned}
\phi_1 \times \phi_2 &: \mathcal{A}_1 \times \mathcal{A}_2 \to \mathbf{Set} \\
\phi_1 \times \phi_2 &\overset{\text{def}}{=} (a_1, a_2) \mapsto \phi_1(a_1) \times \phi_2(a_2) \\
1 &: 1 \to \mathbf{Set} \\
1 &\overset{\text{def}}{=} * \mapsto 1 \\
\psi^\phi &: \mathcal{B}^\mathcal{A} \to \mathbf{Set} \\
\psi^\phi &\overset{\text{def}}{=} F \mapsto \int_a \phi(a) \to \psi(Fa)
\end{aligned}
$$

(Here the integral sign denotes an *end*. Equivalently, the formula for $\psi^\phi(F)$ just computes the set of natural transformations from ϕ to $\psi \circ F$.) Again, the projection functor $(\mathcal{A}, \phi) \mapsto \mathcal{A}$ manifestly preserves the cartesian closed structure. □

Example 3.3.4. The refinement system $\mathbf{Rel}_\bullet \to \mathbf{Rel}$ introduced in Example 3.2.29 is a non-cartesian symmetric monoidal closed refinement system. On objects, the tensor product in **Rel** is defined as for **Set** by taking the cartesian product of sets, but when we consider the morphisms of **Rel** we can observe that this does not give a cartesian monoidal structure. (For example, the singleton set 1 is not a terminal object in **Rel**, since there are as many morphisms $A \to 1$ in **Rel** as there are subsets of A.) Curiously, the residual $A \multimap B$ is *also* defined by the cartesian product of sets $A \multimap B = A \times B$, with the evaluation map

$$
eval : A \otimes (A \multimap B) \to B
$$

defined by the relation

$$
((a_1, (a_2, b_1)), b_2) \in eval \quad \Leftrightarrow \quad a_1 = a_2 \wedge b_1 = b_2.
$$

(Note this actually gives **Rel** the structure of a *compact closed* category, which is a special type of symmetric monoidal closed category.) To get a symmetric monoidal closed structure on **Rel**$_\bullet$, we further define

$$(A_1, R_1) \otimes (A_2, R_2) \overset{\text{def}}{=} (A_1 \times A_2, R_1 \otimes R_2)$$
$$I \overset{\text{def}}{=} (1, I)$$
$$(A, R) \multimap (B, S) \overset{\text{def}}{=} (A \times B, R \multimap S)$$

where the subsets $R_1 \otimes R_2 \subseteq A_1 \times A_2$, $I \subseteq 1$, and $R \multimap S \subseteq A \times B$ are defined by

$$(a_1, a_2) \in R_1 \otimes R_2 \Leftrightarrow a_1 \in R_1 \wedge a_2 \in R_2$$
$$* \in I \Leftrightarrow \text{true}$$
$$(a, b) \in R \multimap S \Leftrightarrow a \in R \Rightarrow b \in S$$

□

3.3.2 Refining the simply typed lambda calculus, revisited

The basic idea of a (cartesian) monoidal closed refinement system gives us a means for understanding some of what we saw in Chapter 2 from a more conceptual and elementary perspective. To a first approximation, the primitive type refinement system $\lambda^{\leq}_{\rightarrow}$ can be modelled by the forgetful functor

$$\begin{tikzcd}\textbf{free-ccc}(\mathcal{P}) \arrow[d, "\textbf{free-ccc}(!_{\mathcal{P}})"'] \\ \textbf{free-ccc}(1)\end{tikzcd}$$

where $\mathcal{P}$ can be an arbitrary category (in Section 2.2.1 we took it to be a preorder), and where **free-ccc**(C) denotes the free cartesian closed category over C.[3] One application of this analysis is that we can derive the set-theoretic semantics of Section 2.2.3 "for free". Recall that we parameterized that semantics by a set D interpreting the unique atomic type ι of $\lambda_{\rightarrow}$, as well as an order-preserving function $i[-] : \mathcal{P} \rightarrow 2^D$ mapping atomic refinements to subsets of D. Note that this is exactly the data of a morphism of refinement systems from the trivial refinement system $!_{\mathcal{P}} : \mathcal{P} \rightarrow 1$ to **Subset** $\rightarrow$ **Set**:

$$\begin{tikzcd}\mathcal{P} \arrow[r, "i[-]"] \arrow[d, "!_{\mathcal{P}}"'] & \textbf{Subset} \arrow[d] \\ 1 \arrow[r, "D"'] & \textbf{Set}\end{tikzcd}$$

[3]I say "to a first approximation", because for various well-known reasons, there is a bit of slack in the connection between $\lambda_{\rightarrow}$ and **free-ccc**(1). For example, the interpretation of terms of $\lambda_{\rightarrow}$ as morphisms of **free-ccc**(1) requires the presence of pairing (i.e., it also models $\lambda_{\rightarrow\times}$), and it does not distinguish between terms which are $\beta\eta$-equivalent. There are different approaches to smoothing over these differences, though, such as passing to multicategories (for the first problem), or to 2-categories (for the second).

But since **Subset** and **Set** are cartesian closed, the universal property of free cartesian closed categories guarantees that we can lift this to a morphism of refinement systems

$$\begin{tikzcd}
\textbf{free-ccc}(\mathcal{P}) \arrow[r, "{[\![-]\!]}"] \arrow[d, "\textbf{free-ccc}(!_{\mathcal{P}})"'] & \textbf{Subset} \arrow[d] \\
\textbf{free-ccc}(1) \arrow[r, "{[\![-]\!]}"'] & \textbf{Set}
\end{tikzcd}$$

and if you unravel the definitions you can verify that this gives precisely the set-theoretic semantics of Section 2.2.3. The same trick works identically for **Downset** $\to$ **Poset** and **Psh** $\to$ **Cat**, since these are also cartesian closed refinement systems. This won't work with $\mathbf{Rel}_\bullet \to \mathbf{Rel}$ since it is only a symmetric monoidal closed refinement system, but on the other hand, we can use the same trick to obtain a model for the *linear* version of $\lambda^{\leq}_{\to}$:

$$\begin{tikzcd}
\textbf{free-smc}(\mathcal{P}) \arrow[r, "{[\![-]\!]}"] \arrow[d, "\textbf{free-smc}(!_{\mathcal{P}})"'] & \mathbf{Rel}_\bullet \arrow[d] \\
\textbf{free-smc}(1) \arrow[r, "{[\![-]\!]}"'] & \textbf{Rel}
\end{tikzcd}$$

3.3.3 Playing with tensor, implies, and, or, push, pull

Things really start to get interesting once we start mixing the connectives $\otimes$, $\multimap$, and $\multimapinv$ of a monoidal closed refinement system with the operations $f_\diamond$ and $f^\square$ of a bifibration. For example, in any monoidal closed refinement system the following distributivity principles hold (as strong equivalences, whenever the corresponding pushforward and pullback refinements exist):

$$(f \otimes g)_\diamond(R \otimes S) \equiv f_\diamond R \otimes g_\diamond S \tag{3.5}$$

$$f_\diamond R \multimap g^\square U \equiv (f \multimap g)^\square(R \multimap U) \tag{3.6}$$

Similar distributivity principles also hold for tensor and implication interacting with unions and intersections:

$$(R_1 \otimes S) \vee (R_2 \otimes S) \equiv (R_1 \vee R_2) \otimes S \tag{3.7}$$

$$(R \multimap U_1) \wedge (R \multimap U_2) \equiv R \multimap (U_1 \wedge U_2) \tag{3.8}$$

$$(R_1 \multimap U) \wedge (R_2 \multimap U) \equiv (R_1 \vee R_2) \multimap U \tag{3.9}$$

Exercise 3.3.5. Prove equations (3.5)–(3.9) using Prop. 3.2.41, and other facts about pushforward and pullback refinements we have already established.

But that's only the beginning of the story – I'll conclude by describing a few applications of monoidal closed refinement systems, which use these connectives in the spirit of a "logical framework" for decomposing the structure of formal systems.

Encoding products and sums using intersections and unions, and vice versa

In programming we're familiar with the idea of representing datatypes by tagged unions. One way of talking about this encoding more abstractly is to observe that a *coproduct refinement*

$$R + S \sqsubset A + B$$

can be decomposed as a *union of pushforwards*,

$$R + S \equiv \mathrm{inl}_\diamond R \vee \mathrm{inr}_\diamond S \tag{3.10}$$

where

$$\begin{aligned}\mathrm{inl} &: A \to A + B\\ \mathrm{inr} &: B \to A + B\end{aligned}$$

are the injections into the underlying coproduct $A + B$. (Recall that we had a preview of this decomposition in Section 2.4.3 when talking about datasort refinements.) Dually, *product refinements*

$$R \times S \sqsubset A \times B$$

can be decomposed as an *intersection of pullbacks*,

$$R \times S \equiv \mathrm{fst}^\square R \wedge \mathrm{snd}^\square S \tag{3.11}$$

where

$$\begin{aligned}\mathrm{fst} &: A \times B \to A\\ \mathrm{snd} &: A \times B \to B\end{aligned}$$

are the projections from the underlying product $A \times B$. (Note that this kind of decomposition was used in Reynolds' Forsythe language [38].)

Exercise 3.3.6. Prove that the strong equivalences (3.10) and (3.10) hold in any monoidal refinement system where the monoidal structures are, respectively, cocartesian or cartesian.

Conversely, intersections and unions can be recovered from products and sums by pulling back or pushing forward,

$$R \wedge S \equiv \delta_A^\square(R \times S) \tag{3.12}$$

$$R \vee S \equiv \varrho_\diamond(R + S) \tag{3.13}$$

where

$$\begin{aligned}\delta_A &: A \to A \times A\\ \varrho_A &: A + A \to A\end{aligned}$$

are the duplication and "coduplication" maps.

Exercise 3.3.7. Prove (3.12) and (3.13).

Separation logic

We've already talked a few times about viewing Hoare logic as a type refinement system (Examples 3.2.12 and 3.2.19). We can also try to talk about the more sophisticated situation of *separation logic* [40], but now, rather than taking the base category $\mathcal{T}$ to be a one-object category, we should take it to be a symmetric monoidal closed category containing a *monoid object*, i.e., an object W equipped with operations

$$\begin{aligned} m &: W \otimes W \to W \\ e &: I \to W \end{aligned}$$

satisfying the monoid axioms. Intuitively, W represents the type of "heaps", with m representing the operation of combining disjoint heaps and e the empty heap.

Consider "separating conjunction" $P * Q$ and "magic wand" $P \mathrel{-\!\!*} Q$, which build predicates on the heap from a pair of predicates on the heap:

$$\frac{P \sqsubset W \quad Q \sqsubset W}{P * Q \sqsubset W} \qquad \frac{P \sqsubset W \quad Q \sqsubset W}{P \mathrel{-\!\!*} Q \sqsubset W}$$

Well, the idea is that we can decompose these "internal" tensor and implication $P * Q$ and $P \mathrel{-\!\!*} Q$ in terms of the "external" tensor and implication $P \otimes Q$ and $P \multimap Q$, by pushing along m or pulling along the currying of m:

$$\begin{aligned} P * Q &\overset{\text{def}}{=} m_\diamond(P \otimes Q) \\ P \mathrel{-\!\!*} Q &\overset{\text{def}}{=} curry(m)^\square(P \multimap Q) \end{aligned}$$

Here is how to see the construction of $P * Q \sqsubset W$ and $P \mathrel{-\!\!*} Q \sqsubset W$ step-by-step:

$$\frac{\dfrac{P \sqsubset W \quad Q \sqsubset W}{P \otimes Q \sqsubset W \otimes W} \quad m : W \otimes W \to W}{m_\diamond(P \otimes Q) \sqsubset W}$$

$$\frac{curry(m) : W \to W \multimap W \quad \dfrac{P \sqsubset W \quad Q \sqsubset W}{P \multimap Q \sqsubset W \multimap W}}{curry(m)^\square(P \multimap Q) \sqsubset W}$$

Similarly, we can define the unit of the separating conjunction $emp \sqsubset W$ by pushing the external tensor unit along the unit operation of the monoid:

$$emp \overset{\text{def}}{=} e_\diamond I$$

These decompositions give us an abstract characterization of the separation logic connectives, which we can instantiate in particular symmetric monoidal closed bifibrations to obtain different interpretations of separation logic. Notably, we can interpret the signature in $\mathbf{Rel}_\bullet \to \mathbf{Rel}$, and use the fact that any *partial commutative monoid* can be represented as a monoid object $(W, M : W \otimes W \to W, E : I \to W)$ in **Rel**:

$$\begin{aligned}(h_1, h_2)\, M\, h &\Leftrightarrow h = h_1 \uplus h_2 \\ *\, E\, h &\Leftrightarrow h = \emptyset\end{aligned}$$

Unwinding the definitions of tensor, implies, push, and pull in the refinement system $\mathbf{Rel}_\bullet \to \mathbf{Rel}$ (see Example 3.2.29 and Example 3.3.4), we recover the standard semantics of separation logic:

$$\begin{aligned}h \in P * Q &\Leftrightarrow \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge h_1 \in P \wedge h_2 \in Q \\ h \in emp &\Leftrightarrow h = \emptyset \\ h \in P \mathrel{-\!\!*} Q &\Leftrightarrow \forall h', h''.\, h'' = h' \uplus h \Rightarrow h' \in P \Rightarrow h'' \in Q\end{aligned}$$

Biorthogonality

A similar trick works for decomposing the concept of *biorthogonal closure*. In any monoidal closed refinement system $r : \mathcal{D} \to \mathcal{T}$, suppose that $\mathcal{T}$ contains some binary operation

$$plug : A \otimes B \to C$$

on arbitrary types A, B, and C, and that we've fixed a refinement

$$Obs \sqsubset C$$

as some "observation" on C. We can then define a pair of *dualization operators* as follows, by pulling back the (left or right) implications along the (left or right) currying of the *plug* operation:

$$R^\perp \overset{\text{def}}{=} lcurry(plug)^* \,(R \multimap Obs) \qquad (R \sqsubset A)$$

$$^\perp S \overset{\text{def}}{=} rcurry(plug)^* \,(Obs \multimapinv S) \qquad (S \sqsubset B)$$

In this very general situation, we automatically obtain a contravariant adjunction between the subcategories of refinements of A and B,

$$\begin{tikzcd} \mathcal{D}_A \arrow[r, bend left, "(-)^\perp"] \arrow[r, phantom, "\perp"] & \mathcal{D}_B^{op} \arrow[l, bend left, "^\perp(-)"] \end{tikzcd}$$

as witnessed by the following equivalences of typing and subtyping judgments:

$$\dfrac{\dfrac{R \otimes S \underset{plug}{\Longrightarrow} Obs}{R \underset{rcurry(plug)}{\Longrightarrow} Obs \multimapinv S}}{R \underset{A}{\Longrightarrow} {}^\perp S} \qquad \dfrac{\dfrac{R \otimes S \underset{plug}{\Longrightarrow} Obs}{S \underset{lcurry(plug)}{\Longrightarrow} R \multimap Obs}}{S \underset{B}{\Longrightarrow} R^\perp}$$

In particular, this means that we have subtyping inclusions from any $R \sqsubset A$ and $S \sqsubset B$ to their biorthogonal closures:

$$R \underset{A}{\Longrightarrow} ({}^{\perp}R)^{\perp} \qquad S \underset{B}{\Longrightarrow} {}^{\perp}(S^{\perp})$$

Refining the "uni-typed" lambda calculus à la Scott

In Chapter 2 we studied various idealized type refinement systems constructed over the simply typed lambda calculus. Many of the refinement systems that one sees "in the wild" have a much more complicated structure, however, where the underlying language has dynamically-typed aspects. Still, the powerful logical tools of monoidal closed bifibrations give us an edge even in these more subtle situations.

An illustrative example of this is pure lambda calculus itself. Scott showed how to model the pure lambda calculus as a *reflexive object* [41] in a cartesian closed category, meaning an object D equipped with operations

$$\begin{tikzcd} D \arrow[r, shift left, "@"] & D^D \arrow[l, shift left, "\lambda"] \end{tikzcd}$$

such that $@ \circ \lambda = \mathrm{id}_{D^D}$. More generally, we can relax the requirement that $@ \circ \lambda = \mathrm{id}_{D^D}$ in various ways (such as by asking for an adjunction $@ \dashv \lambda$, cf. [42]), and we can also consider the linear version of this signature in a symmetric monoidal closed category (cf. [47]):

$$@ : D \to (D \multimap D)$$
$$\lambda : (D \multimap D) \to D$$

In any case, we can then proceed to *refine* the reflexive object D. In particular, by pulling along @ or pushing along λ, we can define two different logical connectives on the refinements of D:

$$R \overset{-}{\multimap} S \overset{\text{def}}{=} @^{\square}(R \multimap S)$$
$$R \overset{+}{\multimap} S \overset{\text{def}}{=} \lambda_{\diamond}(R \multimap S)$$

Note that in general these two connectives will have slightly different interpretations, although they might coincide in certain situations (such as when we have both the β-axiom $@\circ\lambda = \mathrm{id}$ and the η-axiom $\mathrm{id} = \lambda\circ@$). Intuitively, this approach should give us as an abstract way of doing "untyped" realizability-style semantics, but guided, notably, by the presence of types and type refinements.

Bibliography

[1] Franco Barbanera, Mariongala Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119(2):202–230, 1995.

[2] Barendregt, H. P., Coppo, M., and Dezani-Ciancaglini, M.: A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48:931–940, 1983.

[3] Francis Borceux. *Handbook of Categorical Algebra 2: Categories and Structures*. Cambridge University Press, 1994.

[4] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation* 76:138–164, 1988.

[5] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5:56–68, 1940.

[6] Coppo, M. and Dezani-Ciancaglini, M. : An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame Journal of Formal Logic* 21(4):139-156, 1980.

[7] Rowan Davies. A practical refinement-type checker for Standard ML. In Algebraic Methodology and Software Tech. (AMAST'97), pages 565–566. Springer LNCS 1349, 1997.

[8] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University (CMU-CS-05-110), May 2005.

[9] Rowan Davies and Frank Pfenning. Intersection Types and Computational Effects. In *Proceedings of the Fifth International Conference on Functional Programming*, 2000.

[10] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University (CMU-CS-07-129), August 2007.

[11] Joshua Dunfield. Elaborating Intersection and Union Types. In *Proceedings of the 17th International Conference on Functional Programming*, 2012.

[12] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *Proceedings of the 31st Annual Symposium on Principles of Programming Languages*, 2004.

[13] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991, 268–277.

[14] Claudio Hermida. *Fibrations, Logical predicates and indeterminates*, Ph.D. thesis, University of Edinburgh, November 1993.

[15] R. Hindley. The principal type-scheme of an object in combinatory logic *Trans. Amer. Math. Soc.* 146:29–60, 1969.

[16] J. Roger Hindley. Coppo-Dezani types do not correspond to propositional logic. *Theoretical Computer Science*, 28:235–236, 1984.

[17] J. Roger Hindley. Types with intersection. *Formal Aspects of Computing* 4:470–486, 1992.

[18] Bart Jacobs. *Categorical Logic and Type Theory.* Studies in Logic and the Foundations of Mathematics 141. North Holland, 1999.

[19] Max Kelly. *Basic concepts in enriched category theory*. Cambridge University Press, 1982.

[20] Joachim Lambek. From lambda calculus to Cartesian closed categories, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, eds. J. P. Seldin and J. Hindley, Academic Press, 1980, pp. 376-402.

[21] Joachim Lambek and Philip Scott. Introduction to Higher-Order Categorical Logic. Cambridge Studies in Advanced Mathematics. Cambridge University Press 1986.

[22] Daniel Leivant. Polymorphic type inference. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1983.

[23] William Lovas. *Refinement types for logical frameworks*. PhD thesis, Carnegie Mellon University, September 2010.

[24] William Lovas and Frank Pfenning. Refinement types for logical frameworks and their interpretation as proof irrelevance. LMCS. 2010.

[25] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[26] Harry Mairson. A simple proof of a theorem of Statman. Theoretical Computer Science 103(2):387–394, 1992.

[27] Paul-André Melliès and Noam Zeilberger. Functors are Type Refinement Systems. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*, Mumbai, 2015.

[28] Paul-André Melliès and Noam Zeilberger. An Isbell Duality Theorem for Type Refinement Systems. July 31, 2015. arXiv:1501.05115

[29] Paul-André Melliès and Noam Zeilberger. A bifibrational reconstruction of Lawvere's presheaf hyperdoctrine. In *Proceedings of the 31st Annual IEEE Conference on Logic in Computer Science*, New York City, USA, July 2016.

[30] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17:348–375, 1978.

[31] John C. Mitchell. Type Inference with Simple Subtypes. *Journal of Functional Programming* 1(3):245–285, 1991.

[32] Frank Pfenning. Refinement Types for Logical Frameworks. Workshop on Types for Proofs and Programs, May 1993.

[33] Frank Pfenning. Church and Curry: Combining Intrinsic and Extrinsic Typing. Studies in Logic 17:303–338, 2008.

[34] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University (CMU-CS-91-205), 1991.

[35] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[36] John C. Reynolds. Preliminary Design of the Programming Language Forsythe. Report, no. CMU CS 88 159, Carnegie Mellon University, Computer Science Department, June 21, 1988.

[37] John C. Reynolds. Even normal forms can be hard to type. Unpublished, Carnegie Mellon University, December 1, 1989.

[38] John C. Reynolds. Design of the Programming Language Forsythe. Report, no. CMU CS 96 146, Carnegie Mellon University, Computer Science Department, June 1996.

[39] John C. Reynolds. The Meaning of Types: from Intrinsic to Extrinsic Semantics. BRICS Report RS-00-32, Aarhus University, December 2000.

[40] John C. Reynolds. Separation logic: A Logic for Shared Mutable Data Structures. LICS 2002.

[41] Dana S. Scott. Relating theories of the λ-calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism* (eds. Hindley and Seldin), Academic Press, 403–450, 1980.

[42] R. A. G. Seely. Modelling Computations: A 2-Categorical Framework. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, 65–71, Ithaca, NY, USA, 1987.

[43] Morten Heine Sørensen and Pawel Urzyczyn, *Lectures on the Curry-Howard Isomorphism*. Elsevier Science, 2006.

[44] Richard Statman. The typed λ-calculus is not elementary recursive. Theoretical Computer Science 9:73–81, 1979.

[45] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[46] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1998.

[47] Noam Zeilberger. Linear lambda terms as invariants of rooted trivalent maps. December 21, 2015. arXiv:1512.06751