Semantic Set-theoretic Types and Occurrence Typing [currently in submission]

ANDREW M. KENT, Indiana University
SAM TOBIN-HOCHSTADT, Indiana University

We show that the combination of semantic subtyping, as popularized in languages such as ℂduce, and occurrence typing, as found in languages such as Typed Racket and TypeScript, combine in a natural way that increases the expressiveness and reduces the complexity of existing occurrence typing systems. The expressive type operations allowed by set-theoretic types enable us to simplify the presentation of occurrence typing, removing ad-hoc syntactic approaches in favor of complete semantic ones. We also characterize the idea of predicate functions in occurrence typing semantically and give an algorithm for computing the most precise such types.

Additionally, we present a full algorithmic account of a language with semantic set-theoretic types, automatically tested using PLT Redex. To our knowledge, this is the first such account in the literature and enables further experimentation with this powerful and expressive technology.

ACM Reference Format:
Andrew M. Kent and Sam Tobin-Hochstadt. 2018. Semantic Set-theoretic Types and Occurrence Typing [currently in submission]. 1, 1 (April 2018), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For almost every untyped programming language that has gained popularity, a type system has sprung up in its wake. This is true for early systems such as Lisp (Cartwright 1976) and Smalltalk (Bracha and Griswold 1993), but is now widely appreciated in the context of JavaScript (Chaudhuri et al. 2017; Microsoft 2017), Racket, Lua, PHP, Python, and many other languages. In each of these systems, type system designers face a central challenge: **accommodating the idioms of the existing language in a sound, statically-typed fashion**.

Consider the following program, adapted from the TypeScript online documentation:

```
function move(pet : Bird | Fish) {
  if (isFish(pet)) pet.swim();
  else             pet.fly();
}
```

This program shows two features found in TypeScript as well as many other type systems designed for existing untyped languages: ad-hoc union types, and occurrence typing. Union types such as `Bird | Fish` include any values that are *either* a `Bird` or a `Fish`. To distinguish among the possibilities, the programmer has written a function `isFish(pet)`, which not only checks what kind of pet is provided, but is *known to the type system to do so*. Therefore, the occurrence of `pet` in the then branch of the `if` has the type `Fish`, whereas

Authors' addresses: Andrew M. KentIndiana University, andmkent@iu.edu; Sam Tobin-HochstadtIndiana University, samth@iu.edu.

the type in the else branch is `Bird`. This ability to give distinct types to the same variable based on dynamic type tests is known as *occurrence typing*.

To accommodate these idiomatic patterns, union types and occurrence typing[1] are now featured in numerous languages, ranging from the logical types found in Typed Racket (Tobin-Hochstadt and Felleisen 2010) to simple syntactic patterns (Ceylon Project 2017; JetBrains 2017; Groovy 2017) to flow-analysis driven approaches (Chaudhuri et al. 2017; Guha et al. 2010) to TypeScript's expressive but unsound type predicates.

However, almost all of these systems suffer from two serious flaws. First, they fail to reason completely in the presence of union types, meaning that programs that seem obviously correct may fail to type check. And second, they are unable to reason about types as sets of values in ways besides unions; ways that can feel just as natural to programmers and find their way into existing idioms. An example of this is seen in the overloaded type for `hash-ref` found in Typed Racket (slightly simplified here):

```
(: hash-ref (All (a b c) (case-> (-> (HashTable a b) a False (U False b))
                                 (-> (HashTable a b) a (-> c) (U b c)))))
```

This type says that `hash-ref`'s first argument is a table, second argument is a key, and third argument—which specifies what should happen if the key has no entry in the table—has two options: either the value `#false` can be used in which case a `b` or `#false` is returned, or a function producing some type `c` can be used in which case a `b` or `c` is returned. Unfortunately, this type significantly restricts the *actual* behavior of the `hash-ref`—in fact, *any non-function value* is allowed for the third argument. The type given instead reflects the limitations of Typed Racket's type language, which has no way to express the actual semantics of the function (i.e. that upon failing to find a key, the third argument is called with no arguments if it is a procedure, otherwise it is simply returned).

Fortunately, a framework already exists for solving both of these problems—that of semantic subtyping (Frisch et al. 2008), which supports complete reasoning over union types, negation types (needed to describe `hash-ref`'s type), and intersection types (also featured in incomplete form in languages such as TypeScript and Typed Racket).

In this work we show that the combination of semantic subtyping for set-theoretic types and occurrence typing can solve all of these problems at once, while substantially simplifying the theory of occurrence typing, previously reliant on a limited form of dependent types. By combining these two powerful features, we have the opportunity to add expressiveness to existing type systems while eliminating hard-to-understand corner cases for programmers.

The remainder of the paper describes in detail how the combination works at both intuitive and formal levels. One key ingredient is a novel algorithm for determining input types from output value types for function application with set-theoretic types, which follows the existing design patterns for set-theoretic types rather than relying on ad-hoc syntactic rules. Additionally, we give the first *complete* presentation of how to implement semantic subtyping and other key semantic metafunctions for base, product, and function types, with detailed figures generated from an extensively tested PLT Redex model (Klein et al. 2012).[2] We hope that this presentation will inspire others to build on this powerful foundation without the need to reconstruct it themselves.

[1]Occurrence typing can be seen as an elimination form for union types.
[2]An unpublished manuscript by Castagna (2015) outlines a basic framework for implementing semantic set-theoretic types. We build on this, explaining in full detail what is necessary to get such a system off the ground with intuition for how it works and why it is correct.

2 A BRIEF TOUR OF THE OCCURRENCE TYPING AND SEMANTIC SET-THEORETIC TYPES

In this section we review the key ideas behind occurrence typing and semantic set-theoretic types before describing intuitively how they can work together.

2.1 Occurrence Typing Review

Occurrence typing is a general term which describes a flow-sensitive typing system which is capable of checking different occurrences of the same variable at different types.[3] Occurrence typing is well suited for "unityped" languages where idiomatic type-tag tests often appear in control-flow statements. For example, this TypeScript (Microsoft 2017) function `padLeft` pads the left side of a string with either a particular string or a certain number of spaces, depending on the value passed for `padding`:

```
function padLeft(value : string, padding : string | number) {
    if (typeof padding === "number")
        return Array(padding + 1).join(" ") + value;
    if (typeof padding === "string")
        return padding + value;
    throw new Error("Expected string or number, got " + String(padding));
}
```

Here TypeScript's type checker recognizes that in the body of the first `if` the identifier `padding` can safely used as a number and that in the second `if` it can safely be used as a string. Because this style of programming is ubiquitous in unityped languages, an increasing number of type systems support occurrence typing to some degree.

2.2 Semantic Set-theoretic Types Review

Languages with occurrence typing often also feature set-theoretic type constructors such as unions. E.g., the `padLeft` function (from section 2.1) uses the union type `string | number` for the `padding` parameter, which indicates `padding` can be either a `string` *or* a `number`.

Although set-theoretic types offer a flexible and expressive means for describing program invariants, most type systems only reason about them syntactically and therefore feature sound but incomplete subtyping. In other words, they use well understood, valid subtyping rules which cannot prove some obviously true subtyping claims. To illustrate, consider these standard syntactic rules describing reflexivity, union, and product subtyping:

$$\frac{}{\tau <: \tau} \qquad \frac{\tau <: \sigma_1}{\tau <: \sigma_1 | \sigma_2} \qquad \frac{\tau <: \sigma_2}{\tau <: \sigma_1 | \sigma_2} \qquad \frac{\tau_1 <: \sigma \quad \tau_2 <: \sigma}{\tau_1 | \tau_2 <: \sigma} \qquad \frac{\tau_1 <: \sigma_1 \quad \tau_2 <: \sigma_2}{\tau_1 * \tau_2 <: \sigma_1 * \sigma_2}$$

With these rules, it is impossible to prove $(\textsc{Int}|\textsc{Str})*\textsc{Str} <: (\textsc{Int}*\textsc{Str})|(\textsc{Str}*\textsc{Str})$ even though both types describe the *exact same set of values*: pairs with an integer or string in the first field and a string in the second field.

To resolve this, we can instead use a semantic interpretation in which the meaning of a type *is* the set of values it denotes. In the semantic view, subtyping does not depend on a set of syntactic rules, but literally is just a question of set-containment. I.e., a type τ is a subtype of type σ if the values denoted by τ are a *subset* of the values denoted by σ.

[3]To our knowledge, the term "occurrence typing" was first used by Komondoor et al. (2005) to describe a system which reasoned about type-tests in Cobol programs and independently later introduced by Tobin-Hochstadt and Felleisen (2008) to describe a system for type checking Scheme programs.

Although the semantic treatment in some sense is simpler in its handling of set-theoretic type constructors (e.g. type union *really is* equivalent to set union), it is often considered more complex for at least two important reasons: (1) instead of requiring faith in a few (relatively) simple syntactic subtyping axioms, the semantic approach is justified by a series of non-trivial set-theoretic interpretations, and (2) unlike the syntactic approach, an implementation strategy does not obviously follow from the description.

Regarding (1), fortunately Frisch et al. (2008) have provided a *thorough* accounting for why the natural semantic approach for set-theoretic types "just works" (i.e. why it is sound and formally justified). For (2), some unpublished work (Castagna 2015) has described basic implementation strategies which have proven effective in the context of the ℂduce project. In this work we hope to build on those strategies, providing further intuition and raw implementation details so more curious language practitioners can experiment with and enjoy the expressiveness and flexibility semantic set-theoretic types offer.

2.3 Combining Occurrence Typing with Semantic Set-theoretic Types

Our contributions revolve around a novel approach for supporting occurrence typing in which we semantically analyze arbitrary set-theoretic function types to infer argument types based on return values. To illustrate, consider these two λ_{TR} occurrence typing examples from Tobin-Hochstadt and Felleisen (2010):

λ_{TR} example 1:

```
(λ ([x : Any])
  (if (int? x)
      (add1 x)
      0))
```

λ_{TR} example 14:

```
(λ ([y : (U Int Str)] [p : (Pair Any Any)])
  (cond
    [(and (int? y) (int? (proj 1 p)))
     (+ input (proj 1 p))]
    [(int? (proj 1 p))
     (+ (strlen y) (proj 1 p))]
    [else 0]))
```

In order to type check such programs, a type system must be able to (1) recognize that the result of an expression such as `(int? x)` witnesses something about the type of its argument `x` (this allows example 1 to type check) and (2) reason logically about control flow and how it relates to the witnessed type information for in-scope identifiers (which, with (1), is how example 14 type checks).

To achieve (2), we use a flow-sensitive typing judgment like λ_{TR}'s which propagates logical information properly w.r.t. control flow. For (1)—i.e. determining when a function's return value witnesses something about the type of a term—we propose a novel semantic analysis at function application sites which answers the following questions:

- what must be true about a function's argument if a non-`false` value is returned?
- what must be true about a function's argument if `false` is returned?

Actually, the technique—which we describe in detail in section 4.4—is more general and can reason about an arbitrary output type (i.e. not just whether or not the returned value is `false`). Using this approach, we show that semantic set-theoretic types themselves offer a rich, compositional foundation for occurrence typing capable of type checking *all* of the example occurrence typing programs proposed by Tobin-Hochstadt and Felleisen (2010).

bop ::=	`+` | `-` | `×`	**Binary Operators**
e ::=		**Expressions**
	| x	Variable
	| c	Constant
	| (e e)	Application
	| (bop e e)	Binary Op. Application
	| (λ\{τ, ...\}(x) e)	Abstraction
	| (`proj` i e)	Pair Projection (i ∈ \{1,2\})
	| (`pair` e e)	Pair Construction
	| (`if` e e e)	Conditional
	| (`let` (x e) e)	Local Binding
π ::=		**Paths**
	| x	Variable Path
	| (*proj* i π)	Path Projection (i ∈ \{1,2\})
o ::=		**Objects**
	| $\top^o$	Trivial Object
	| $\bot^o$	Absurd Object
	| π	Path
$\mathbb{B}$ ::=	#t | #f	**Meta Booleans**

τ, σ ::=		**Types**
	| ι	Base Type
	| `τ*τ`	Product
	| `τ->τ`	Function
	| `τ|τ`	Union
	| `τ&τ`	Intersection
	| `!τ`	Complement
	| Any	Universal Type
	| Empty	Empty Type
p, q ::=		**Propositions**
	| $\pi \in \tau$	Type Membership
	| $p \wedge p$	Conjunction
	| $p \vee p$	Disjunction
	| $x \rightsquigarrow \pi$	Alias
	| $\mathbb{tt}$	Trivial
	| $\mathbb{ff}$	Absurd
Γ ::=	$\emptyset$ | Γ, p	**Contexts**
ρ ::=	ϵ | ρ[x $\mapsto$ v]	**Environments**
R ::=	$\langle\tau$, p, q, o$\rangle$	**Type Results**
c ::=	*int* | *str* | uop | `true` | `false`	**Constants**
ι ::=	Int | Str | True | False	**Base Types**
uop ::=	`not` | `strlen` | `zero?` | `int?` | `str?` | `pair?` | `fun?`	**Unary Operators**
v ::=	c | (`pair` v v) | [ρ, (λ\{τ, ...\}(x) e)]	**Values**

Bool $\equiv$ True|False   Any* $\equiv$ Any*Any   Any$^{->}$ $\equiv$ Empty->Any   Any$^\iota$ $\equiv$!(Any*|Any$^{->}$)

Figure 1: λ_{so} syntax and abbreviations

3 THE CALCULUS

λ_{so} is a minimal typed lambda calculus rich enough to illustrate how semantic set-theoretic types and occurrence typing coalesce. The figures we present are generated directly from a heavily tested PLT Redex (Klein et al. 2012) prototype of the language.

3.1 Syntax

The syntax of λ_{so} is enumerated in figure 1; judgments and metafunctions definitions describing the static and dynamic semantics of λ_{so} appear in subsequent sections. Regarding these definitions, an ellipsis to the right of a pattern indicates repetition of that pattern, and metafunction definitions are ordered from top-to-bottom (i.e. a particular case is only reached if the preceding cases' patterns or constraints did not match).

3.1.1 Constants (c) *and Primitive Operators* (uop *and* bop)*.* λ_{so} has integer, string, unary operator, and boolean constants. *int* and *str* are metavariables ranging over integer and string literals respectively; `true` and `false` are the boolean literals. Note that semantically in λ_{so}, all non-`false` values are considered 'true' when tested in conditionals. Unary operator semantics are enumerated in figure 11 and binary operators correspond to the respective mathematical operations on integers.

3.1.2 Expressions (e) *and Values* (v). λ_{so} expression syntax is mostly standard. Pair projections ($\texttt{proj}\ \mathrm{i}\ \mathrm{e}$) use a literal index (i.e. the metavariable i) to specify which pair field is being projected. Abstractions are annotated with a set of types which describes the intended domains of the function. The enumeration of domain types allows a user to indicate an abstraction's type should be an intersection of function types with the specified domains, e.g. $(\lambda\{\textsc{Bool}\}(\mathrm{x})\ (\texttt{not}\ \mathrm{x}))$ is a $\textsc{Bool}\texttt{->}\textsc{Bool}$ whereas $(\lambda\{\textsc{True}, \textsc{False}\}(\mathrm{x})\ (\texttt{not}\ \mathrm{x}))$ is assigned the more specific type $\textsc{True}\texttt{->}\textsc{False}\texttt{\&}\textsc{False}\texttt{->}\textsc{True}$.

For values we use reuse constant, unary operator, and pair syntax from expressions and use closures to represent function values.

3.1.3 Base Types (ι) *and Types* (τ *or* σ). A type in λ_{so} intuitively denotes a set of values:

- $\textsc{Int}$ denotes the set of all integers;
- $\textsc{Str}$ denotes the set of all strings;
- $\textsc{True}$ denotes the singleton set $\{\texttt{true}\}$;
- $\textsc{False}$ denotes the singleton set $\{\texttt{false}\}$.
- $\tau\texttt{*}\sigma$ denotes the set of pairs whose first element is a τ and second element is a σ;
- $\tau\texttt{->}\sigma$ denotes the set of all functions which when given a value of type τ produce a value of type σ;[4]
- $\tau\texttt{|}\sigma$ denotes the union of τ and σ (i.e. values of type τ *or* σ);
- $\tau\texttt{\&}\sigma$ denotes the intersection τ and σ (i.e. values of type τ *and* σ);
- $\texttt{!}\tau$ denotes the complement of the set denoted by τ (i.e. values *not* of type τ);
- $\textsc{Any}$ denotes the set of all values; and
- $\textsc{Empty}$ denotes the empty set of values (i.e. the uninhabited type).

Figure 1 also contains some abbreviations for common types: $\textsc{Bool}$ (the type for booleans) is a union of the individual boolean types, while $\textsc{Any}^{\iota}$, $\textsc{Any}^{\texttt{*}}$, and $\textsc{Any}^{\texttt{->}}$ are the top types describing all base values, all pair values, and all function values.

3.1.4 Environments (ρ)*, Paths* (π)*, and Objects* (o). An environment is a simple mapping of variables to values and is either empty (ϵ) or is an environment extended with a single mapping from a variable to its value ($\rho[\mathrm{x} \mapsto \mathrm{v}]$).

A path is a named address which can be looked up in the current environment. E.g., if ρ is the current runtime environment, then the path x refers to the entry for x in ρ, the path $(\mathit{proj}\ 1\ \mathrm{x})$ refers to the first projection of the entry for x in ρ, the path $(\mathit{proj}\ 2\ (\mathit{proj}\ 1\ \mathrm{x}))$ refers to the second projection of the first projection of the entry for x in ρ, etc. We write $\rho(\pi)$ to refer to the value described by π in ρ. Although some paths may be nonsensical (i.e. a projection from a non-pair), at any point where such a path arises during type checking, that code would be considered dead since we could derive $\mathbb{ff}$.

Objects are used to describe what (if any) path will correspond to the result of evaluating an expression. For any given expression, either (1) that expression's value will correspond to some path π in ρ, (2) an expression's value has no known correlation to a value in ρ—denoted by the trivial object $\top^{o}$—or (3) an expression is provably unreachable (i.e. dead code) and thus corresponds with the absurd object $\perp^{o}$.

[4]A language with recursion would instead say that $\tau\texttt{->}\sigma$ denotes the set of all functions which when given a value of type τ, *if they produce a value*, then that value is of type σ.

3.1.5 Propositions (p *or* q*) and Contexts (*Γ*).* λ_{so} propositions include standard components from propositional logic, type membership propositions $\pi \in \tau$ which state that the value found at path π is of type τ, and equality statements $x \rightsquigarrow \pi$ which indicate that the variable x has a value equal to π's value. A context Γ is just a collection of propositions.

3.1.6 Type Results (R*).* Type results allow the type system to ascribe additional information to an expression beyond merely its type. A type result $\langle \tau, \mathrm{p}, \mathrm{q}, o \rangle$ describes an expression of type τ which, if used in a conditional test, allows us to assume p in the then-branch and q in the else-branch, and whose value corresponds to the object o. These additional pieces of information are key to occurrence typing since they allow the typing judgment to propagate information learned from control-flow tests.

3.1.7 Miscellaneous Syntax. The non-terminal 'i' is used to range over the set $\{1, 2\}$, which allows for convenient descriptions of pair operations. The non-terminal $\mathbb{B}$ ranges over the metatheoretic boolean literals #t and #f, which are generally used to define metatheoretic predicate functions or to make an otherwise partial metafunction total (i.e. returning #f to indicate failure).

3.2 Subtyping and Logic

In this section we lay out the general framework for understanding subtyping and the logic of λ_{so}. In section 4.3 we fully explore algorithmic subtyping and in section 4.1 we discuss how to implement the logical reasoning necessary for type checking.

3.2.1 Subtyping. In section 3.1.3 we gave an intuitive way to interpret a type τ as a set of values; notationally we will refer to the set a type denotes by writing $[\![\tau]\!]$. With this intuitive interpretation in mind, we proceed to define subtyping for λ_{so}:

Definition 3.1 (Semantic Subtyping). A type τ is a subtype of type σ, written $\tau <: \sigma$, if and only if $[\![\tau]\!] \subseteq [\![\sigma]\!]$, i.e. the interpretation of τ is a subset of the interpretation of σ.

For further reading regarding the formal justification for semantically dealing with set-theoretic types, we recommend perusing the work of Frisch et al. (2008); our use of semantic subtyping is a direct application of their work.

3.2.2 Logic. The logic of λ_{so} can be viewed as a standard propositional theory which reasons about simple term equality and statements of type (i.e. set) membership. Because both subtyping and inhabitation for types in λ_{so} are decidable (see section 4.3 for a discussion on how), it is not difficult to see that a minor extension to any one of the well-understood approaches to propositional reasoning suffices to make our logic decidable. We therefore leave abstract the exact choice of rules for derivations, requiring only that they conform to the following specification:

Definition 3.2 (Logic). A proposition p can be derived in context Γ, written $\Gamma \vdash \mathrm{p}$, if and only if $\forall \rho . \rho \vDash \Gamma$ implies that $\rho \vDash \mathrm{p}$.

$\rho \vDash \mathrm{p}$ here is read "ρ satisfies p" and intuitively means that the runtime environment ρ is one in which the proposition p is inherently true. A context Γ is satisfied if each proposition in it is satisfied. Satisfaction is formally defined in figure 11.

3.3 Typing Rules

The typing judgment for λ_{so} is defined in figure 2. Like some previous calculi for occurrence typing, it assigns a *type-result* (instead of merely a type) to well-typed expressions:

$$\boxed{\Gamma \vdash e : R}$$

$$\textsc{T-Const}\ \frac{}{\Gamma \vdash c : \Delta^{c}(c)} \qquad \textsc{T-BOp}\ \frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash (\mathrm{bop}\ e_1\ e_2) : \langle \texttt{Int}, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle}$$

$$\textsc{T-Var}\ \frac{\Gamma \vdash x \rightsquigarrow \pi \qquad \Gamma \vdash \pi \in \tau}{\Gamma \vdash x : \langle \tau, \pi \in \texttt{!False}, \pi \in \texttt{False}, \pi \rangle}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \langle \tau_2, \_, \_, o_2 \rangle \qquad \tau = \mathsf{apply}^{?}(\tau_1, \tau_2) \qquad \langle \sigma_+, \sigma_- \rangle = \mathsf{pred}^{?}(\tau_1)}{\Gamma \vdash (e_1\ e_2) : \langle \tau, \mathsf{is}(o_2, \sigma_+), \mathsf{is}(o_2, \sigma_-), \top^{o} \rangle}$$

$$\textsc{T-Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{pair}\ e_1\ e_2) : \langle \tau_1 \texttt{*} \tau_2, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle} \qquad \textsc{T-Proj}\ \frac{\Gamma \vdash e : \langle \tau, \_, \_, o \rangle \qquad \tau_i = \mathsf{proj}^{?}(i, \tau)}{\Gamma \vdash (\texttt{proj}\ i\ e) : \langle \tau_i, \mathbb{tt}, \mathbb{tt}, \mathsf{proj}^{o}(i, o) \rangle}$$

$$\textsc{T-Abs}\ \frac{\Gamma, x \in \tau_k \vdash e : \sigma_k \qquad \ldots}{\Gamma \vdash (\lambda\{\tau_k, \ldots\}(x)\ e) : \langle (\tau_k \texttt{->} \sigma_k) \texttt{\&} \ldots, \mathbb{tt}, \mathbb{ff}, \top^{o} \rangle} \qquad \textsc{T-If}\ \frac{\Gamma \vdash e_1 : \langle \tau_1, p_1, q_1, \_ \rangle \qquad \Gamma, p_1 \vdash e_2 : R_2 \qquad \Gamma, q_1 \vdash e_3 : R_3}{\Gamma \vdash (\texttt{if}\ e_1\ e_2\ e_3) : R_2 \vee R_3}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash e_1 : R_1 \qquad \Gamma, \mathsf{alias}(x, R_1) \vdash e_2 : R_2}{\Gamma \vdash (\texttt{let}\ (x\ e_1)\ e_2) : R_2} \qquad \textsc{T-Sub}\ \frac{\Gamma \vdash e : R_1 \qquad \Gamma \vdash R_1 <: R_2}{\Gamma \vdash e : R_2} \qquad \textsc{T-ExFalso}\ \frac{\Gamma \vdash \mathbb{ff}}{\Gamma \vdash e : \langle \texttt{Empty}, \mathbb{ff}, \mathbb{ff}, \bot^{o} \rangle}$$

Figure 2: typing judgment

$$\Gamma \vdash e : \langle \tau, p, q, o \rangle$$

This judgment states that in environment Γ

- e has type τ;
- if e evaluates to a non-`false` (i.e. treated as true) value ‘then proposition’ p holds;
- if e evaluates to `false` ‘else proposition’ q holds;
- if e evaluates to a value, it corresponds to looking up o in the runtime environment.

We occasionally write $\Gamma \vdash e : \tau$ (i.e. use a type instead of a type result) when we are only interested in the type of an expression. For simplicity of presentation and proof, we require that the free variables in R be a subset of those found in Γ and assume that programs are uniquely named. The syntactic metafunctions use by the type system—i.e. those which primarily match on the literal syntax of the language—are defined in figure 3. The semantic metafunctions $\mathsf{proj}^{?}$, $\mathsf{apply}^{?}$, and $\mathsf{pred}^{?}$—functions which must reason about arbitrary types by considering which sets of values they represent—are intuitively described in this section with their corresponding typing rule; we discuss and examine their full definitions in section 4.

3.3.1 T-Const, T-BOp, *and* T-Pair. Because constants, binary operators, and pairs do not witness any non-obvious logical information, these terms have straightforward type

$$\boxed{\Delta^{u} : \mathrm{uop} \to \tau}$$

$$\begin{aligned}
\Delta^{u}(\texttt{zero?}) &= \textsc{Int}\texttt{->}\textsc{Bool}\\
\Delta^{u}(\texttt{strlen}) &= \textsc{Str}\texttt{->}\textsc{Int}\\
\Delta^{u}(\texttt{not}) &= (\textsc{False}\texttt{->}\textsc{True})\texttt{\&}((\texttt{!}\textsc{False})\texttt{->}\textsc{False})\\
\Delta^{u}(\texttt{int?}) &= (\textsc{Int}\texttt{->}\textsc{True})\texttt{\&}((\texttt{!}\textsc{Int})\texttt{->}\textsc{False})\\
\Delta^{u}(\texttt{str?}) &= (\textsc{Str}\texttt{->}\textsc{True})\texttt{\&}((\texttt{!}\textsc{Str})\texttt{->}\textsc{False})\\
\Delta^{u}(\texttt{pair?}) &= (\textsc{Any}^{*}\texttt{->}\textsc{True})\texttt{\&}((\texttt{!}\textsc{Any}^{*})\texttt{->}\textsc{False})\\
\Delta^{u}(\texttt{fun?}) &= (\textsc{Any}^{\texttt{->}}\texttt{->}\textsc{True})\texttt{\&}((\texttt{!}\textsc{Any}^{\texttt{->}})\texttt{->}\textsc{False})
\end{aligned}$$

$$\boxed{\mathsf{alias} : \mathrm{x}\ \mathrm{R} \to \mathrm{p}}$$

$$\begin{aligned}
\mathsf{alias}(\mathrm{x}, \langle \_, \_, \_, \bot^{o}\rangle) &= \mathbb{ff}\\
\mathsf{alias}(\mathrm{x}, \langle \_, \_, \_, \pi\rangle) &= \mathrm{x} \rightsquigarrow \pi\\
\mathsf{alias}(\mathrm{x}, \langle \tau, \mathrm{p}, \mathrm{q}, \top^{o}\rangle) &= \mathrm{x} \in \tau \wedge (\mathrm{p}_{\mathrm{x}} \vee \mathrm{q}_{\mathrm{x}})\\
\text{where } \mathrm{p}_{\mathrm{x}} = \mathrm{x} \in \texttt{!}\textsc{False} \wedge \mathrm{p},\ & \mathrm{q}_{\mathrm{x}} = \mathrm{x} \in \textsc{False} \wedge \mathrm{q}
\end{aligned}$$

$$\boxed{\mathsf{is} : \mathrm{o}\ \tau \to \mathrm{p}}$$

$$\begin{aligned}
\mathsf{is}(\_, \tau) &= \mathbb{ff} \quad \text{where } \tau <: \textsc{Empty}\\
\mathsf{is}(\top^{o}, \_) &= \mathbb{tt}\\
\mathsf{is}(\bot^{o}, \tau) &= \mathbb{ff}\\
\mathsf{is}(\pi, \tau) &= \pi \in \tau
\end{aligned}$$

$$\boxed{\Delta^{c} : \mathrm{c} \to \mathrm{R}}$$

$$\begin{aligned}
\Delta^{c}(\mathit{int}) &= \langle \textsc{Int}, \mathbb{tt}, \mathbb{ff}, \top^{o}\rangle\\
\Delta^{c}(\mathit{str}) &= \langle \textsc{Str}, \mathbb{tt}, \mathbb{ff}, \top^{o}\rangle\\
\Delta^{c}(\mathrm{uop}) &= \langle \Delta^{u}(\mathrm{uop}), \mathbb{tt}, \mathbb{ff}, \top^{o}\rangle\\
\Delta^{c}(\texttt{true}) &= \langle \textsc{True}, \mathbb{tt}, \mathbb{ff}, \top^{o}\rangle\\
\Delta^{c}(\texttt{false}) &= \langle \textsc{False}, \mathbb{ff}, \mathbb{tt}, \top^{o}\rangle
\end{aligned}$$

$$\boxed{\_\mathsf{V}\_ : \mathrm{o}\ \mathrm{o} \to \mathrm{o}}$$

$$\begin{aligned}
\pi \mathbin{\mathsf{V}} \pi &= \pi\\
\bot^{o} \mathbin{\mathsf{V}} \mathrm{o} &= \mathrm{o}\\
\mathrm{o} \mathbin{\mathsf{V}} \bot^{o} &= \mathrm{o}\\
\_ \mathbin{\mathsf{V}} \_ &= \top^{o}
\end{aligned}$$

$$\boxed{\_\mathsf{V}\_ : \mathrm{R}\ \mathrm{R} \to \mathrm{R}}$$

$$\begin{aligned}
&\langle \tau_1, \mathrm{p}_1, \mathrm{q}_1, \mathrm{o}_1\rangle \mathbin{\mathsf{V}} \langle \tau_2, \mathrm{p}_2, \mathrm{q}_2, \mathrm{o}_2\rangle\\
&= \langle \tau_1 \mathbin{|} \tau_2, \mathrm{p}_1 \vee \mathrm{p}_2, \mathrm{q}_1 \vee \mathrm{q}_2, \mathrm{o}_1 \mathbin{\mathsf{V}} \mathrm{o}_2\rangle
\end{aligned}$$

$$\boxed{\mathsf{proj}^{o} : \mathrm{i}\ \mathrm{o} \to \mathrm{o}}$$

$$\begin{aligned}
\mathsf{proj}^{o}(\mathrm{i}, \pi) &= (\mathit{proj}\ \mathrm{i}\ \pi)\\
\mathsf{proj}^{o}(\mathrm{i}, \top^{o}) &= \top^{o}\\
\mathsf{proj}^{o}(\mathrm{i}, \bot^{o}) &= \bot^{o}
\end{aligned}$$

Figure 3: various type checking metafunctions

results which include the expected type, the trivial object (since their values do not necessarily correlate to any entry in ρ), and propositions declaring the self-evident fact of whether or not that expression's value can overlap with `false`. E.g., a non-`false` expression has 'then-proposition' $\mathbb{tt}$ because nothing is learned if its value is non-`false`, and 'else-proposition' $\mathbb{ff}$ since it is impossible for that expression to evaluate to `false`.

3.3.2 T-Abs. For abstractions we check that for each annotated argument type τ_k, the body is well-typed at some type σ_k. The overall type for the abstraction is the intersection of each such type τ_k`->`σ_k and the propositions and objects for abstractions indicate that abstractions are non-`false` and have no necessary correspondence to a path.

3.3.3 T-Var. When checking a variable x the type system can utilize any relevant aliasing info in Γ. E.g., if in Γ we can derive that x is an alias for (*proj* 1 p), then the type system can more-or-less treat x *as if it were* (`proj` 1 p) when checking it. If there is no alias, π can of course be chosen to be simply x since—like equality—aliasing is reflexive. Any derivable type for π in Γ can be used. The propositions indicate that testing this expression's value corresponds to testing whether π is `false` or not. The object indicates that this expression's value indeed corresponds to π in the current ρ.

3.3.4 T-Proj. When checking a projection from a pair, we first verify the underlying expression e has some type τ. Because τ can be an arbitrarily complex set-theoretic type, we then call the semantic metafunction $\mathsf{proj}^?$ (defined in section 4.5) to determine what type (if any) corresponds to the i^{th} projection from τ. The proj° metafunction determines the appropriate object for the projection.

3.3.5 T-App. For function application, after checking that the expressions e_1 and e_2 are well-typed at some types τ_1 and τ_2 there are several important tasks to perform: (1) determine if e_1 is a function which can be applied to e_2, (2) determine the *most specific* result type for this function application, and (3) decide what can be learned about the argument's type if either a non-`false` value or `false` is returned.

Because τ_1 could be an arbitrarily complicated set-theoretic type, we accomplish (1) and (2) with the help of the semantic metafunction $\mathsf{apply}^?$. It verifies that τ_1 is indeed a function type, calculates its domain τ_d, and then verifies that τ_2 <: τ_d holds. If τ_2 is indeed a valid argument, it then calculates the *smallest* type τ for which τ_1 <: τ_2->τ holds (i.e. the most precise return type for an input of type τ_2).

For (3), we use the semantic metafunction $\mathsf{pred}^?$ which, for an *arbitrary* function type, produces a pair of types $\langle\sigma_+, \sigma_-\rangle$ corresponding to precisely what input types are mapped to non-`false` values and `false` respectively. E.g., $\mathsf{pred}^?$ can determine from the *type* of `int?` that it is a predicate for integers:

$$\langle \textsc{Int}, \texttt{!}\textsc{Int}\rangle = \mathsf{pred}^?((\textsc{Int}\texttt{->}\textsc{True})\texttt{\&}(\texttt{!}\textsc{Int}\texttt{->}\textsc{False}))$$

And even for this bizzare function type, $\mathsf{pred}^?$ can determine that only integers and strings can be mapped to true values, and anything but `false` can be mapped to `false`:

$$\langle \textsc{Int}\texttt{|}\textsc{Str}, \texttt{!}\textsc{False}\rangle = \mathsf{pred}^?((\textsc{Int}\texttt{|}\textsc{Str}\texttt{->}\textsc{Bool})\texttt{\&}(\texttt{!}(\textsc{Int}\texttt{|}\textsc{Str})\texttt{->}\textsc{False})\texttt{\&}(\textsc{False}\texttt{->}\textsc{Empty}))$$

Whatever information is extracted by $\mathsf{pred}^?$ is converted into propositions by the is metafunction and used as the respective 'then' and 'else' propositions for the type result.

Finally, because the result of function application in general has no necessary correspondence to an entry in the ρ, the object of function application is $\top^o$.

3.3.6 T-If. Conditional expressions play a vital role in occurrence typing by allowing information learned from a conditional test-expression to refine Γ in the respective branches. I.e., after checking the test expression e_1 of a conditional, the 'then branch' e_2 is checked while assuming e_1's then-proposition p and the 'else branch' e_3 is checked while assuming e_1's else-proposition q. The overall type result for the conditional is the pointwise union of the two branches' type results.

3.3.7 T-Let. For local binding forms we first check e_1—the expression being bound to the variable x—and extend Γ with propositions to appropriately describe x (see the alias metafunction). I.e., if e_1's object corresponds to a path π already described in Γ, we can simply record this fact ($x \rightsquigarrow \pi$) and then type check the body knowing x also corresponds to this path. If e_1's object is trivial, then we need to record x's type and any information that x may witness (i.e. the type and propositions derived from checking e_1). If e_1's object is absurd, then we may assume $\mathbb{ff}$, since that implies this code is unreachable.

The overall type result for a local binding form is just the type result of the body. For the sake of simplicity in our formalism, x must not appear in the free variables of R_2 for

R_2 to be well-formed. However, with a uniquely named program, leaving occurrences of x in R_2 does not violate soundness.

3.3.8 T-SUB *and* T-EXFALSO. T-SUB allows us to either refine the type result of an expression with additional information from the current context Γ or weaken the result if, for example, we are merely checking that an expression is of a particular type. Subsumption of type results is described in more detail in section 3.4.

T-EXFALSO corresponds to the idea that if our context has become absurd then we are examining an *unreachable* branch of code and need not type check it.

3.4 Type Result Subsumption

Subsumption for type results (used in the rule T-SUB) allows a result to be either made more specific with information from Γ or less specific via weakening. Recalling from section 3.3 how type results assign meaning to expressions, we can imagine refining a type result $\Gamma \vdash e : \langle \tau, p, q, o \rangle$ by leveraging principles such as the following:

- if either $(\Gamma, p \vdash \mathbb{ff})$ or (τ`&!False` $\approx$ `Empty`), then the then-proposition can be strengthened to $\mathbb{ff}$ and the type can be made τ`&False`
- if either $(\Gamma, q \vdash \mathbb{ff})$ or (τ`&False` $\approx$ `Empty`), then the else-proposition can be strengthened to $\mathbb{ff}$ and the type can be made τ`&!False`

Put more formally, we can describe a type result as the set of values represented by its underlying type *refined* by the accompanying propositions and object:

Definition 3.3 (Type Result Subsumption). In context Γ, type result R_1 can be subsumed by R_2, written $\Gamma \vdash R_1 <: R_2$, if and only if $\forall \rho.\, \rho \models \Gamma$ implies $[\![R_1]\!]_\rho \subseteq [\![R_2]\!]_\rho$ where $[\![R]\!]_\rho$ is defined as follows:

$$\begin{aligned}
[\![\langle \tau, p, q, \bot^o \rangle]\!]_\rho &= \emptyset \\
[\![\langle \tau, p, q, \pi \rangle]\!]_\rho &= \{v \in [\![\tau]\!] \text{ s.t. } \rho(\pi) = v \text{ and if } v \neq \texttt{false} \text{ then } \rho \models p \text{ else } \rho \models q\} \\
[\![\langle \tau, p, q, \top^o \rangle]\!]_\rho &= \{v \in [\![\tau]\!] \text{ s.t. if } v \neq \texttt{false} \text{ then } \rho \models p \text{ else } \rho \models q\}
\end{aligned}$$

4 ALGORITHMIC TYPE CHECKING

In section 3 we provide a declarative description for how λ_{so} can be type checked. In this section, we aim to "bridge the gap" and provide all the missing details necessary for actually implementing such a system.

4.1 Algorithmic Logic

The following strategies—in addition to basic techniques from propositional logic—make working with the logical environment straightforward.

4.1.1 Aggressive Aliasing. Whenever a local binding binds a variable x to an expression with a non-trivial path π, the type system should *always* choose to reason about π instead of x in that local scope. Doing this consistently will create a "copy propagation"-like effect whereby the type system can more-or-less ignore local identifiers which it can already describe by some other name. This helps reduce environmental complexity and makes it easier to consistently track the most specific current type for variables.

4.1.2 Context Conjunctive Normal Form. The following steps help keep Γ in conjunctive normal form (CNF) which makes logical inquiries during type checking easier to decide.

First, reduce all atomic propositions into statements about variables by repeatedly converting $(\textit{proj}\ 1\ \pi) \in \tau$ into $\pi \in \tau$`*Any` and $(\textit{proj}\ 2\ \pi) \in \tau$ into $\pi \in$ `Any*`τ.

$t, s ::= \langle \beta, b^{*}, b^{\texttt{->}} \rangle$	**Internal Type Rep.**
$\beta ::= \langle \mathbb{B}, B \rangle$	**Base Component**
$B ::= \{\iota, ...\}$	**Base Types**
$b ::=$	**Type BDD**
$\mid \mathbb{1}$	Top Node
$\mid \mathbb{0}$	Bottom Node
$\mid n$	Type Node
$n ::= \langle a, b, b, b \rangle$	**Type Node**
$a ::= \texttt{t*t} \mid \texttt{t->t}$	**Type Atoms**
$b^{*} ::= \mathbb{1} \mid \mathbb{0} \mid \langle \texttt{t*t}, b^{*}, b^{*}, b^{*} \rangle$	**Pair BDD**
$b^{\texttt{->}} ::= \mathbb{1} \mid \mathbb{0} \mid \langle \texttt{t->t}, b^{\texttt{->}}, b^{\texttt{->}}, b^{\texttt{->}} \rangle$	**Function BDD**
$P, N ::= \emptyset \mid \{a\} \cup P$	**Atom Sets**

$\top \equiv \langle \langle \#f, \{\} \rangle, \mathbb{1}, \mathbb{1} \rangle$	Top Type Rep.
$\bot \equiv \langle \langle \#t, \{\} \rangle, \mathbb{0}, \mathbb{0} \rangle$	Bottom Type Rep.
$\top^{\iota} \equiv \langle \langle \#f, \{\} \rangle, \mathbb{0}, \mathbb{0} \rangle$	Top Base Type Rep.
$\top^{*} \equiv \langle \langle \#t, \{\} \rangle, \mathbb{1}, \mathbb{0} \rangle$	Top Pair Type Rep.
$\top^{\texttt{->}} \equiv \langle \langle \#t, \{\} \rangle, \mathbb{0}, \mathbb{1} \rangle$	Top Function Type Rep.

$$\boxed{\mathsf{a} : n \to a}$$
$$\mathsf{a}(\langle a, b_l, b_m, b_r \rangle) = a$$

$$\boxed{\mathsf{l} : n \to b}$$
$$\mathsf{l}(\langle a, b_l, b_m, b_r \rangle) = b_l$$

$$\boxed{\mathsf{m} : n \to b}$$
$$\mathsf{m}(\langle a, b_l, b_m, b_r \rangle) = b_m$$

$$\boxed{\mathsf{r} : n \to b}$$
$$\mathsf{r}(\langle a, b_l, b_m, b_r \rangle) = b_r$$

$$\boxed{\langle \_,\_,\_,\_ \rangle : a\ b\ b\ b \to b}$$
$$\begin{aligned} \langle a, b_l, \mathbb{1}, b_r \rangle &= \mathbb{1} \\ \langle a, b, b_m, b \rangle &= b \cup b_m \\ \langle a, b_l, b_m, b_r \rangle &= \langle a, b_l, b_m, b_r \rangle \end{aligned}$$

Figure 4: internal type representation syntax, node accessors, and smart node construction

Second, perform standard logical transformations so Γ is in CNF. Note that atoms with the same subject can be joined into a single atom either via intersection (i.e. $x \in \tau_1$ *and* $x \in \tau_2$ implies $x \in \tau_1 \& \tau_2$) or disjunction (i.e. $x \in \tau_1$ *or* $x \in \tau_2$ implies $x \in \tau_1 | \tau_2$) when appropriate. Also, $x \in$ Any and $x \in$ Empty are logically equivalent to $\mathbb{tt}$ and $\mathbb{ff}$ respectively (for deciding emptiness, see section 4.3).

Once in CNF, use the principle of disjunctive syllogism to simplify and reduce disjunctions by eliminating their impossible atoms.

4.1.3 Applying Subsumption. Subsumption should regularly be used to simplify type results and propositions in the two specific ways highlighted in section 3.4. Additionally, each time the environment is extended with a logical proposition (i.e. T-If and T-Let), the information from that proposition should be included in the overall output type result.

4.2 Set-theoretic Type Representation

In figure 1 we define the "surface level" syntax of set-theoretic types. This syntax is human readable and can express any set-theoretic type, but it is not an efficient representation conducive to the computations we wish to perform. In this section we describe a specialized type representation based on ideas described by Frisch et al. (2008) and Castagna (2015) which we have successfully used to implement the ideas in this work. The syntactic grammar for this representation is given in figure 4.

4.2.1 Type Emptiness and Disjunctive Normal Form. First we make an important observation that shapes our type representation: many type inquiries can be phrased in terms of emptiness. For example, here is how subtyping can be phrased as an emptiness inquiry:

$$\tau \texttt{ <: } \sigma \text{ iff } [\![\tau]\!] \subseteq [\![\sigma]\!] \text{ iff } [\![\tau]\!] \backslash [\![\sigma]\!] = \emptyset \text{ iff } [\![\tau]\!] \cap \overline{[\![\sigma]\!]} = \emptyset \text{ iff } \tau\texttt{\&!}\sigma \approx \textsc{Empty}$$

Because of this, it is useful to choose a representation suited for deciding type emptiness. Fortunately, like boolean formula satisfiability, type emptiness is conveniently decided with a disjunctive normal form (DNF) representation, i.e. a union of intersections of positive and negative atoms. In this normal form, each clause can be described with a pair (P, N), where P is the set of positive atoms and N is the set of negative atoms in that conjunction. Furthermore, a DNF type τ can naturally be partitioned into three disjoint "specialized" types τ^{ι}, τ^{*}, and $\tau^{\texttt{->}}$ (each containing atoms of one particular kind) such that $\tau = (\textsc{Any}^{\iota}\texttt{\&}\tau^{\iota})\texttt{|}(\textsc{Any}^{*}\texttt{\&}\tau^{*})\texttt{|}(\textsc{Any}^{\texttt{->}}\texttt{\&}\tau^{\texttt{->}})$. Individually these partitions have the following structure:

$$\tau^{\iota} = \bigcup_{(P,N)\in\tau^{\iota}} \left(\left(\bigcap_{\iota\in P} \iota \right) \texttt{\&} \left(\bigcap_{\iota\in N} \neg\iota \right) \right)$$

$$\tau^{*} = \bigcup_{(P,N)\in\tau^{*}} \left(\left(\bigcap_{\tau_1\texttt{*}\tau_2\in P} \tau_1\texttt{*}\tau_2 \right) \texttt{\&} \left(\bigcap_{\tau_1\texttt{*}\tau_2\in N} \neg(\tau_1\texttt{*}\tau_2) \right) \right)$$

$$\tau^{\texttt{->}} = \bigcup_{(P,N)\in\tau^{\texttt{->}}} \left(\left(\bigcap_{\tau_1\texttt{->}\tau_2\in P} \tau_1\texttt{->}\tau_2 \right) \texttt{\&} \left(\bigcap_{\tau_1\texttt{->}\tau_2\in N} \neg(\tau_1\texttt{->}\tau_2) \right) \right)$$

Where the large $\bigcup$ and $\bigcap$ are used as convenient syntax for describing an accumulating type union and intersection while iterating over a particular set.

*4.2.2 Internal Type Representation (*t *or* s*).* As figure 4 illustrates, this partitioning is exactly how we internally represent set-theoretic types. The internal representation for a type is simply a 3-tuple $\langle \beta, \mathrm{b}^{*}, \mathrm{b}^{\texttt{->}} \rangle$ with a specialized representation for each partition. Type operations (e.g. union, intersection, etc) on these data structures work out to be simple point-wise operations across the three subcomponents:

$$\boxed{\_\cup\_ : \mathrm{t}\ \mathrm{t} \rightarrow \mathrm{t}}$$
$$\langle \beta_1, \mathrm{b}^{*}_1, \mathrm{b}^{\texttt{->}}_1 \rangle \cup \langle \beta_2, \mathrm{b}^{*}_2, \mathrm{b}^{\texttt{->}}_2 \rangle = \langle \beta_1 \cup \beta_2, \mathrm{b}^{*}_1 \cup \mathrm{b}^{*}_2, \mathrm{b}^{\texttt{->}}_1 \cup \mathrm{b}^{\texttt{->}}_2 \rangle$$

$$\boxed{\_\cap\_ : \mathrm{t}\ \mathrm{t} \rightarrow \mathrm{t}}$$
$$\langle \beta_1, \mathrm{b}^{*}_1, \mathrm{b}^{\texttt{->}}_1 \rangle \cap \langle \beta_2, \mathrm{b}^{*}_2, \mathrm{b}^{\texttt{->}}_2 \rangle = \langle \beta_1 \cap \beta_2, \mathrm{b}^{*}_1 \cap \mathrm{b}^{*}_2, \mathrm{b}^{\texttt{->}}_1 \cap \mathrm{b}^{\texttt{->}}_2 \rangle$$

$$\boxed{\_\backslash\_ : \mathrm{t}\ \mathrm{t} \rightarrow \mathrm{t}}$$
$$\langle \beta_1, \mathrm{b}^{*}_1, \mathrm{b}^{\texttt{->}}_1 \rangle \setminus \langle \beta_2, \mathrm{b}^{*}_2, \mathrm{b}^{\texttt{->}}_2 \rangle = \langle \beta_1 \setminus \beta_2, \mathrm{b}^{*}_1 \setminus \mathrm{b}^{*}_2, \mathrm{b}^{\texttt{->}}_1 \setminus \mathrm{b}^{\texttt{->}}_2 \rangle$$

$$\boxed{\neg : \mathrm{t} \rightarrow \mathrm{t}}$$
$$\neg\mathrm{t} = \top \setminus \mathrm{t}$$

In figure 5 we define a metafunction parse which parses the surface level type syntax given in figure 1 into this internal representation, which may be helpful in understanding the correspondence between the two.

4.2.3 Base Type Unions (β). For the base type portion of a type, because base types are by their nature disjoint, any specialized type τ^ι is actually equivalent to either a union of base types $\iota_0 \mathbf{|} \iota_1 \mathbf{|} \ldots$ or a negated union of base types $\mathbf{!}(\iota_0 \mathbf{|} \iota_1 \mathbf{|} \ldots)$. With this insight, we represent the base portion of a type as a tuple $\langle \mathbb{B}, \mathrm{B} \rangle$ with a boolean flag indicating if this represents a standard or negated union and a set containing the base types in the union.

Binary operations on the base portion of a type—defined in figure 5—corresponds to determining the correct binary set operation to combine the respective sets.

4.2.4 Lazy BDDs for Pairs and Functions (b^{*} and $\mathrm{b}^{\texttt{->}}$). To represent the product and function portion of a DNF type, we again borrow a technique from boolean formula representation: binary decision diagrams. More specifically, we use "lazy" binary decision diagrams (henceforth BDD), one BDD to represent the product portion of the type (b^{*}), and another BDD to represent the function portion of the type ($\mathrm{b}^{\texttt{->}}$). Note that the superscripts are used only to distinguish which kind of atoms a BDD contains; a b with no superscript is used to describe an atom-agnostic operation.

These BDDs are actually ternary trees in which each path from the root to a non-absurd leaf node (i.e. $\mathbb{1}$) represents a particular intersection of the atoms along that path, and the overall meaning of the BDD is the union of each such path. In particular, a non-leaf node $\langle \mathrm{a}, \mathrm{b}_l, \mathrm{b}_m, \mathrm{b}_r \rangle$ contains an atom and three child BDDs and is interpreted to mean the type $(\mathrm{a}\mathbf{\&}\mathrm{b}_l) \mathbf{|} \mathrm{b}_m \mathbf{|} (\mathbf{!}\mathrm{a}\mathbf{\&}\mathrm{b}_r)$, where $\mathbb{1}$ is interpreted as Any, $\mathbb{0}$ is interpreted as Empty, and each child BDD is recursively interpreted in the same fashion. This approach and its usage of a lazy union as the middle child helps avoid unnecessary explosion in type size and is reasonable since our primary concern is type *inhabitation* and computing the union of two types is a strictly additive operation (i.e. the union of two types always produces a type at least as large as the originals).

Operations on BDDs—which are described in figure 6—have several straightforward cases (i.e. those specified in the top row) and are only somewhat complex when both BDDs are unique non-leaf nodes. The non-leaf node cases—all of which are specified in the bottom half of figure 6—utilize an ordering on atoms and are defined piecewise based on this ordering, i.e. the first case for when the atom of n_1 comes before the atom of n_2, the second case for when the atom of n_2 comes before the atom of n_1, and the third case for when their atoms are equal. This ordering simply ensures a particular BDD has a unique tree representation. When we construct a non-leaf node, we use the "smart constructor" defined in figure 4 which checks for a few obvious simplifications before actually constructing the node. Finally, the "lazy" nature of these BDDs can be seen in that the middle child (i.e. the lazy union portion) is only perturbed by type intersection and difference computations; while computing unions, if the atoms are not equal, we simply store one of the BDDs in the middle subtree of the other to lazily record the union.

4.3 Subtyping and Emptiness

As we discussed in section 4.2.1, subtyping (and many other inquieries) can be determined simply by deciding emptiness when working semantically with set-theoretic types. Because of this, our internal definition of subtyping—defined in figure 7— immediately delegates to the empty metafunction after computing the difference between the types.

To determine if a DNF type is empty, we check if each clause in the overall disjunction is empty. With our internal type representation, this means we need to be able to decide if the base portion, product portion, and function portion of a type are each empty.

$$\boxed{\mathsf{parse} : \tau \to \mathrm{t}}$$

$$\begin{aligned}
\mathsf{parse}(\iota) &= \langle\langle\#\mathrm{t}, \{\iota\}\rangle, \mathbb{0}, \mathbb{0}\rangle \\
\mathsf{parse}(\tau\texttt{*}\sigma) &= \langle\langle\#\mathrm{t}, \{\}\rangle, \langle\mathrm{t}\texttt{*}\mathrm{s}, \mathbb{1}, \mathbb{0}, \mathbb{0}\rangle, \mathbb{0}\rangle \\
&\quad \text{where } \mathrm{t} = \mathsf{parse}(\tau), \mathrm{s} = \mathsf{parse}(\sigma) \\
\mathsf{parse}(\tau\texttt{->}\sigma) &= \langle\langle\#\mathrm{t}, \{\}\rangle, \mathbb{0}, \langle\mathrm{t}\texttt{->}\mathrm{s}, \mathbb{1}, \mathbb{0}, \mathbb{0}\rangle\rangle \\
&\quad \text{where } \mathrm{t} = \mathsf{parse}(\tau), \mathrm{s} = \mathsf{parse}(\sigma) \\
\mathsf{parse}(\tau\,|\,\sigma) &= \mathsf{parse}(\tau) \cup \mathsf{parse}(\sigma) \\
\mathsf{parse}(\tau\texttt{\&}\sigma) &= \mathsf{parse}(\tau) \cap \mathsf{parse}(\sigma) \\
\mathsf{parse}(\texttt{!}\tau) &= \neg\mathsf{parse}(\tau) \\
\mathsf{parse}(\textsc{Any}) &= \langle\langle\#\mathrm{f}, \{\}\rangle, \mathbb{1}, \mathbb{1}\rangle \\
\mathsf{parse}(\textsc{Empty}) &= \langle\langle\#\mathrm{t}, \{\}\rangle, \mathbb{0}, \mathbb{0}\rangle
\end{aligned}$$

$$\boxed{\_\cup\_ : \beta\ \beta \to \beta}$$

$$\begin{aligned}
\langle\#\mathrm{t}, \mathrm{B}_1\rangle \cup \langle\#\mathrm{t}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_1 \cup \mathrm{B}_2\rangle \\
\langle\#\mathrm{f}, \mathrm{B}_1\rangle \cup \langle\#\mathrm{f}, \mathrm{B}_2\rangle &= \langle\#\mathrm{f}, \mathrm{B}_1 \cap \mathrm{B}_2\rangle \\
\langle\#\mathrm{t}, \mathrm{B}_1\rangle \cup \langle\#\mathrm{f}, \mathrm{B}_2\rangle &= \langle\#\mathrm{f}, \mathrm{B}_2 \setminus \mathrm{B}_1\rangle \\
\langle\#\mathrm{f}, \mathrm{B}_1\rangle \cup \langle\#\mathrm{t}, \mathrm{B}_2\rangle &= \langle\#\mathrm{f}, \mathrm{B}_1 \setminus \mathrm{B}_2\rangle
\end{aligned}$$

$$\boxed{\_\cap\_ : \beta\ \beta \to \beta}$$

$$\begin{aligned}
\langle\#\mathrm{t}, \mathrm{B}_1\rangle \cap \langle\#\mathrm{t}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_1 \cap \mathrm{B}_2\rangle \\
\langle\#\mathrm{f}, \mathrm{B}_1\rangle \cap \langle\#\mathrm{f}, \mathrm{B}_2\rangle &= \langle\#\mathrm{f}, \mathrm{B}_1 \cup \mathrm{B}_2\rangle \\
\langle\#\mathrm{t}, \mathrm{B}_1\rangle \cap \langle\#\mathrm{f}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_1 \setminus \mathrm{B}_2\rangle \\
\langle\#\mathrm{f}, \mathrm{B}_1\rangle \cap \langle\#\mathrm{t}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_2 \setminus \mathrm{B}_1\rangle
\end{aligned}$$

$$\boxed{\_\setminus\_ : \beta\ \beta \to \beta}$$

$$\begin{aligned}
\langle\#\mathrm{t}, \mathrm{B}_1\rangle \setminus \langle\#\mathrm{t}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_1 \setminus \mathrm{B}_2\rangle \\
\langle\#\mathrm{f}, \mathrm{B}_1\rangle \setminus \langle\#\mathrm{f}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_2 \setminus \mathrm{B}_1\rangle \\
\langle\#\mathrm{t}, \mathrm{B}_1\rangle \setminus \langle\#\mathrm{f}, \mathrm{B}_2\rangle &= \langle\#\mathrm{t}, \mathrm{B}_1 \cap \mathrm{B}_2\rangle \\
\langle\#\mathrm{f}, \mathrm{B}_1\rangle \setminus \langle\#\mathrm{t}, \mathrm{B}_2\rangle &= \langle\#\mathrm{f}, \mathrm{B}_1 \cup \mathrm{B}_2\rangle
\end{aligned}$$

Figure 5: Type parsing and base operations

Finally, since some of these computations will necessarily have an exponential worst case time complexity, our algorithms aim to at least consume only linear space and short-circuit when possible. In practice, memoizing subtyping/emptiness calculations and using a shared memory representation for types will be worthwhile.

4.3.1 Base Type Emptiness. With our representation of the base portion of a DNF type, deciding emptiness is trivial: if the base portion is literally $\langle\#\mathrm{t}, \{\}\rangle$ (i.e. an empty positive union), then the base portion is empty, otherwise it is non-empty.

4.3.2 Product Type Emptiness. For the product portion of a DNF type to be empty, each clause in the disjunct must be empty. As we discussed in section 4.2.1, clauses in the product portion of a DNF can be represented by a tuple containing the positive and negative pair types in that conjunction (P, N). Because pairs types are covariant, the information contained in the pairs in P can be combined into a single pair type $\sigma_1\texttt{*}\sigma_2$ by intersecting all of the first and all of the second fields respectively. Now consider that for each negative pair type $\tau_1\texttt{*}\tau_2 \in N$, by De Morgan's law, we can distribute the negation across the product and see that $\texttt{!}(\tau_1\texttt{*}\tau_2)$ is equivalent logically to either the first field being a $\texttt{!}\tau_1$ *or* the second field being a $\texttt{!}\tau_2$. Therefore, for this product type to be empty, it must be empty in *both* cases. In order to check this for each of the negations in N, we ensure that for each $N' \subseteq N$ the following boolean statement is true:

$$\left(\sigma_1 <: \bigcup_{\tau_1\texttt{*}\tau_2 \in N'} \tau_1\right) \text{ or } \left(\sigma_2 <: \bigcup_{\tau_1\texttt{*}\tau_2 \in N \setminus N'} \tau_2\right)$$

This is in effect exploring each possible combination of negations, ensuring that the negative information would render one of the fields (and thus the whole product type) empty for each possibility.

$$\begin{array}{l|l|l|l}
\boxed{\_\cup\_ : \mathrm{b}\ \mathrm{b} \to \mathrm{b}} & \boxed{\_\cap\_ : \mathrm{b}\ \mathrm{b} \to \mathrm{b}} & \boxed{\_\setminus\_ : \mathrm{b}\ \mathrm{b} \to \mathrm{b}} & \boxed{\neg : \mathrm{b} \to \mathrm{b}} \\
\mathrm{b} \cup \mathrm{b} = \mathrm{b} & \mathrm{b} \cap \mathrm{b} = \mathrm{b} & \mathrm{b} \setminus \mathrm{b} = \mathbb{0} & \neg\mathbb{1} = \mathbb{0} \\
\mathrm{b} \cup \mathbb{1} = \mathbb{1} & \mathrm{b} \cap \mathbb{1} = \mathrm{b} & \mathrm{b} \setminus \mathbb{1} = \mathbb{0} & \neg\mathbb{0} = \mathbb{1} \\
\mathbb{1} \cup \mathrm{b} = \mathbb{1} & \mathbb{1} \cap \mathrm{b} = \mathrm{b} & \mathbb{1} \setminus \mathrm{b} = \neg\mathrm{b} & \neg\langle \mathrm{a}, \mathrm{b}_1, \mathrm{b}_2, \mathbb{0}\rangle = \langle \mathrm{a}, \mathbb{0}, \neg\mathrm{b}_2 \cup \mathrm{b}_1, \neg\mathrm{b}_2\rangle \\
\mathrm{b} \cup \mathbb{0} = \mathrm{b} & \mathrm{b} \cap \mathbb{0} = \mathbb{0} & \mathrm{b} \setminus \mathbb{0} = \mathrm{b} & \neg\langle \mathrm{a}, \mathbb{0}, \mathrm{b}_2, \mathrm{b}_3\rangle = \langle \mathrm{a}, \neg\mathrm{b}_2, \neg\mathrm{b}_2 \cup \mathrm{b}_3, \mathbb{0}\rangle \\
\mathbb{0} \cup \mathrm{b} = \mathrm{b} & \mathbb{0} \cap \mathrm{b} = \mathbb{0} & \mathbb{0} \setminus \mathrm{b} = \mathbb{0} & \neg\langle \mathrm{a}, \mathrm{b}_1, \mathbb{0}, \mathrm{b}_3\rangle = \langle \mathrm{a}, \neg\mathrm{b}_1, \neg\mathrm{b}_1 \cup \mathrm{b}_3, \neg\mathrm{b}_3\rangle \\
 & & & \neg\langle \mathrm{a}_1, \mathrm{b}_1, \mathrm{b}_2, \mathrm{b}_3\rangle = \langle \mathrm{a}_1, \neg\mathrm{b}_1 \cup \mathrm{b}_2, \mathbb{0}, \neg\mathrm{b}_3 \cup \mathrm{b}_2\rangle
\end{array}$$

$$\boxed{\_\cup\_ : \mathrm{n}\ \mathrm{n} \to \mathrm{b}}$$

$$\begin{array}{ll}
\mathrm{n}_1 \cup \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_1), \mathsf{l}(\mathrm{n}_1), \mathsf{m}(\mathrm{n}_1) \cup \mathrm{n}_2, \mathsf{r}(\mathrm{n}_1)\rangle & \text{where } \mathsf{a}(\mathrm{n}_1) < \mathsf{a}(\mathrm{n}_2) \\
\mathrm{n}_1 \cup \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_2), \mathsf{l}(\mathrm{n}_2), \mathsf{m}(\mathrm{n}_2) \cup \mathrm{n}_1, \mathsf{r}(\mathrm{n}_2)\rangle & \text{where } \mathsf{a}(\mathrm{n}_1) > \mathsf{a}(\mathrm{n}_2) \\
\mathrm{n}_1 \cup \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_1), \mathsf{l}(\mathrm{n}_1) \cup \mathsf{l}(\mathrm{n}_2), \mathsf{m}(\mathrm{n}_1) \cup \mathsf{m}(\mathrm{n}_2), \mathsf{r}(\mathrm{n}_1) \cup \mathsf{r}(\mathrm{n}_2)\rangle &
\end{array}$$

$$\boxed{\_\cap\_ : \mathrm{n}\ \mathrm{n} \to \mathrm{b}}$$

$$\begin{array}{ll}
\mathrm{n}_1 \cap \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_1), \mathsf{l}(\mathrm{n}_1) \cap \mathrm{n}_2, \mathsf{m}(\mathrm{n}_1) \cap \mathrm{n}_2, \mathsf{r}(\mathrm{n}_1) \cap \mathrm{n}_2\rangle & \text{where } \mathsf{a}(\mathrm{n}_1) < \mathsf{a}(\mathrm{n}_2) \\
\mathrm{n}_1 \cap \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_2), \mathrm{n}_1 \cap \mathsf{l}(\mathrm{n}_2), \mathrm{n}_1 \cap \mathsf{m}(\mathrm{n}_2), \mathrm{n}_1 \cap \mathsf{r}(\mathrm{n}_2)\rangle & \text{where } \mathsf{a}(\mathrm{n}_1) > \mathsf{a}(\mathrm{n}_2) \\
\mathrm{n}_1 \cap \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_1), \mathrm{b}_\mathrm{l}, \mathbb{0}, \mathrm{b}_\mathrm{r}\rangle &
\end{array}$$

$$\text{where } \mathrm{b}_\mathrm{l} = (\mathsf{l}(\mathrm{n}_1) \cup \mathsf{m}(\mathrm{n}_1)) \cap (\mathsf{l}(\mathrm{n}_2) \cup \mathsf{m}(\mathrm{n}_2)),\ \mathrm{b}_\mathrm{r} = (\mathsf{r}(\mathrm{n}_1) \cup \mathsf{m}(\mathrm{n}_1)) \cap (\mathsf{r}(\mathrm{n}_2) \cup \mathsf{m}(\mathrm{n}_2))$$

$$\boxed{\_\setminus\_ : \mathrm{n}\ \mathrm{n} \to \mathrm{b}}$$

$$\begin{array}{ll}
\mathrm{n}_1 \setminus \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_1), (\mathsf{l}(\mathrm{n}_1) \cup \mathsf{m}(\mathrm{n}_1)) \setminus \mathrm{n}_2, \mathbb{0}, (\mathsf{r}(\mathrm{n}_1) \cup \mathsf{m}(\mathrm{n}_1)) \setminus \mathrm{n}_2\rangle & \text{where } \mathsf{a}(\mathrm{n}_1) < \mathsf{a}(\mathrm{n}_2) \\
\mathrm{n}_1 \setminus \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_2), \mathrm{n}_1 \setminus (\mathsf{l}(\mathrm{n}_2) \cup \mathsf{m}(\mathrm{n}_2)), \mathbb{0}, \mathrm{n}_1 \setminus (\mathsf{r}(\mathrm{n}_2) \cup \mathsf{m}(\mathrm{n}_2))\rangle & \text{where } \mathsf{a}(\mathrm{n}_1) > \mathsf{a}(\mathrm{n}_2) \\
\mathrm{n}_1 \setminus \mathrm{n}_2 = \langle \mathsf{a}(\mathrm{n}_1), \mathsf{l}(\mathrm{n}_1) \setminus \mathrm{n}_2, \mathsf{m}(\mathrm{n}_1) \setminus \mathrm{n}_2, \mathsf{r}(\mathrm{n}_1) \setminus \mathrm{n}_2\rangle &
\end{array}$$

Figure 6: Type BDD operations

In figure 7 we define a metafunction empty* which performs this computation by traversing the product BDD accumulating the positive and negative type information in parameters s_1, and s_2, and N respectively. Once it is has reached non-absurd leaf $\mathbb{1}$ (and thus accumulated all the positive and negative information for a particular clause in the DNF), it then calls the helper metafunction θ^* to explore the space of possible combinations of negated fields, only returning #t if for each possibility, one of the product type's fields is necessarily uninhabited. Note that for θ^*, we avoid requiring accumulators for the negative type information by subtracting it from the positive type accumulator parameters for the appropriate field in each recursive calls. Also, if a field is found to be empty we need not explore the remaining negative type information.

4.3.3 Function Type Emptiness. Like products, a function DNF is empty when each of its clauses (P, N) is empty. For an individual clause (i.e. an intersection of positive and negative arrows) to be empty, there must exist some τ_1->$\tau_2 \in N$ which *contradicts* the positive information in P. How is the positive information contradicted? Well, if we know it is not a function of type τ_1->τ_2, but we can show that τ_1 <: $(\bigcup_{\sigma_1\text{->}\sigma_2} \sigma_1)$ (i.e. τ_1 is in the domain of the function) *and* that the arrows in P necessarily map τ_1 to a subtype of

$\boxed{\_\approx\_ : \mathrm{t}\ \mathrm{t} \to \mathbb{B}}$

$$\mathrm{s} \approx \mathrm{t} \;=\; \mathrm{s} <: \mathrm{t} \;\text{ and }\; \mathrm{t} <: \mathrm{s}$$

$\boxed{\_<:\_ : \mathrm{t}\ \mathrm{t} \to \mathbb{B}}$

$$\mathrm{s} <: \mathrm{t} \;=\; \mathsf{empty}(\mathrm{s} \setminus \mathrm{t})$$

$\boxed{\mathsf{empty}^{*} : \mathrm{b}^{*}\ \mathrm{s}\ \mathrm{s}\ \mathrm{N} \to \mathbb{B}}$

$$\begin{aligned}
&\mathsf{empty}^{*}(\mathbb{0}, \mathrm{s}_1, \mathrm{s}_2, \mathrm{N}) \;=\; \#\mathrm{t}\\
&\mathsf{empty}^{*}(\mathbb{1}, \mathrm{s}_1, \mathrm{s}_2, \mathrm{N})\\
&\quad = \mathsf{empty}(\mathrm{s}_1)\\
&\qquad \text{or } \mathsf{empty}(\mathrm{s}_2)\\
&\qquad \text{or } \theta^{*}(\mathrm{s}_1, \mathrm{s}_2, \mathrm{N})\\
&\mathsf{empty}^{*}(\langle \mathrm{t}_1 {*} \mathrm{t}_2, \mathrm{b}^{*}{}_{\mathrm{l}}, \mathrm{b}^{*}{}_{\mathrm{m}}, \mathrm{b}^{*}{}_{\mathrm{r}}\rangle, \mathrm{s}_1, \mathrm{s}_2, \mathrm{N})\\
&\quad = \mathsf{empty}^{*}(\mathrm{b}^{*}{}_{\mathrm{l}}, \mathrm{s}_1 \cap \mathrm{t}_1, \mathrm{s}_2 \cap \mathrm{t}_2, \mathrm{N})\\
&\qquad \text{and } \mathsf{empty}^{*}(\mathrm{b}^{*}{}_{\mathrm{m}}, \mathrm{s}_1, \mathrm{s}_2, \mathrm{N})\\
&\qquad \text{and } \mathsf{empty}^{*}(\mathrm{b}^{*}{}_{\mathrm{r}}, \mathrm{s}_1, \mathrm{s}_2, \{\mathrm{t}_1 {*} \mathrm{t}_2\} \cup \mathrm{N})
\end{aligned}$$

$\boxed{\theta^{*} : \mathrm{s}\ \mathrm{s}\ \mathrm{N} \to \mathbb{B}}$

$$\begin{aligned}
&\theta^{*}(\mathrm{s}_1, \mathrm{s}_2, \emptyset) \;=\; \#\mathrm{f}\\
&\theta^{*}(\mathrm{s}_1, \mathrm{s}_2, \{\mathrm{t}_1 {*} \mathrm{t}_2\} \cup \mathrm{N})\\
&\quad = (\mathrm{s}_1 <: \mathrm{t}_1 \text{ or } \theta^{*}(\mathrm{s}_1 \setminus \mathrm{t}_1, \mathrm{s}_2, \mathrm{N}))\\
&\qquad \text{and } (\mathrm{s}_2 <: \mathrm{t}_2 \text{ or } \theta^{*}(\mathrm{s}_1, \mathrm{s}_2 \setminus \mathrm{t}_2, \mathrm{N}))
\end{aligned}$$

$\boxed{\mathsf{empty} : \mathrm{t} \to \mathbb{B}}$

$$\begin{aligned}
&\mathsf{empty}(\langle\langle \#\mathrm{t}, \{\}\rangle, \mathrm{b}^{*}, \mathrm{b}^{\text{->}}\rangle)\\
&\quad = \mathsf{empty}^{*}(\mathrm{b}^{*}, \top, \top, \emptyset)\\
&\qquad \text{and } \mathsf{empty}^{\text{->}}(\mathrm{b}^{\text{->}}, \bot, \emptyset, \emptyset)\\
&\mathsf{empty}(\_) \;=\; \#\mathrm{f}
\end{aligned}$$

$\boxed{\mathsf{empty}^{\text{->}} : \mathrm{b}^{\text{->}}\ \mathrm{s}\ \mathrm{P}\ \mathrm{N} \to \mathbb{B}}$

$$\begin{aligned}
&\mathsf{empty}^{\text{->}}(\mathbb{0}, \mathrm{s}, \mathrm{P}, \mathrm{N}) \;=\; \#\mathrm{t}\\
&\mathsf{empty}^{\text{->}}(\mathbb{1}, \mathrm{s}, \mathrm{P}, \emptyset) \;=\; \#\mathrm{f}\\
&\mathsf{empty}^{\text{->}}(\mathbb{1}, \mathrm{s}, \mathrm{P}, \{\mathrm{t}_1\text{->}\mathrm{t}_2\} \cup \mathrm{N})\\
&\quad = (\mathrm{t}_1 <: \mathrm{s} \text{ and } \theta^{\text{->}}(\mathrm{t}_1, \neg\mathrm{t}_2, \mathrm{P}))\\
&\qquad \text{or } \mathsf{empty}^{\text{->}}(\mathbb{1}, \mathrm{s}, \mathrm{P}, \mathrm{N})\\
&\mathsf{empty}^{\text{->}}(\langle \mathrm{s}_{\mathrm{d}}\text{->}\mathrm{s}_{\mathrm{r}}, \mathrm{b}^{\text{->}}{}_{\mathrm{l}}, \mathrm{b}^{\text{->}}{}_{\mathrm{m}}, \mathrm{b}^{\text{->}}{}_{\mathrm{r}}\rangle, \mathrm{s}, \mathrm{P}, \mathrm{N})\\
&\quad = \mathsf{empty}^{\text{->}}(\mathrm{b}^{\text{->}}{}_{\mathrm{l}}, \mathrm{s} \cup \mathrm{s}_{\mathrm{d}}, \{\mathrm{s}_{\mathrm{d}}\text{->}\mathrm{s}_{\mathrm{r}}\} \cup \mathrm{P}, \mathrm{N})\\
&\qquad \text{and } \mathsf{empty}^{\text{->}}(\mathrm{b}^{\text{->}}{}_{\mathrm{m}}, \mathrm{s}, \mathrm{P}, \mathrm{N})\\
&\qquad \text{and } \mathsf{empty}^{\text{->}}(\mathrm{b}^{\text{->}}{}_{\mathrm{r}}, \mathrm{s}, \mathrm{P}, \{\mathrm{s}_{\mathrm{d}}\text{->}\mathrm{s}_{\mathrm{r}}\} \cup \mathrm{N})
\end{aligned}$$

$\boxed{\theta^{\to} : \mathrm{t}_1\ \mathrm{t}_2\ \mathrm{P} \to \mathbb{B}}$

$$\begin{aligned}
&\theta^{\text{->}}(\mathrm{t}_1, \mathrm{t}_2, \emptyset) \;=\; \mathsf{empty}(\mathrm{t}_1) \text{ or } \mathsf{empty}(\mathrm{t}_2)\\
&\theta^{\text{->}}(\mathrm{t}_1, \mathrm{t}_2, \{\mathrm{s}_1\text{->}\mathrm{s}_2\} \cup \mathrm{P})\\
&\quad = (\mathrm{t}_1 <: \mathrm{s}_1 \text{ or } \theta^{\text{->}}(\mathrm{t}_1 \setminus \mathrm{s}_1, \mathrm{t}_2, \mathrm{P}))\\
&\qquad \text{and } (\mathrm{t}_2 <: \neg\mathrm{s}_2 \text{ or } \theta^{\text{->}}(\mathrm{t}_1, \mathrm{t}_2 \cap \mathrm{s}_2, \mathrm{P}))
\end{aligned}$$

Figure 7: internal type equivalence, subtyping, and emptiness metafunctions

τ_2, then we have a contradiction. To compute this, after performing the domain check, we check if for each non-empty set of arrows $P' \subseteq P$ the following holds:

$$\left(\tau_1 <: \bigcup_{\sigma_1\text{->}\sigma_2 \in P \setminus P'} \sigma_1\right) \text{ or } \left(\bigcap_{\sigma_1\text{->}\sigma_2 \in P'} \sigma_2 <: \tau_2\right)$$

The left-hand side of the above disjunction is checking if the current set of arrows P' can be ignored because the domain of the arrows *not* in P' already cover inputs of type τ_1. If that is not the case, we ensure that the range of the intersection of arrows in P' would have a range that is a subtype of τ_2. If this formula holds for each non-empty set of arrows P' then this clause in the function DNF is uninhabited since it is contradictory.

This formula is implemented by the metafunction $\mathsf{empty}^{\text{->}}$ defined in figure 7. It traverses the function BDD accumulating the positive and negative sets of arrows and the domain in parameters P, N, and s respectively. At the non-absurd leafs (i.e. the base case where the first parameter is $\mathbb{1}$) it begins examining each negated arrow $\mathrm{t}_1\text{->}\mathrm{t}_2$ checking if t_1 is contained in the current domain of P and whether the aforementioned contradiction holds for each non-empty $P' \subseteq P$ this clause.

The contradiction check for each $P' \subseteq P$ is performed with metafunction $\theta^{\texttt{->}}$. One important note for the helper function is that t_2 is negated before it is passed to $\theta^{\texttt{->}}$. This is done because instead of checking $\bigcap_{\sigma_1\texttt{->}\sigma_2 \in P'} \sigma_2 <: \tau_2$ as the above equation suggests, it is more convenient to be checking for the logically equivalent contrapositive statement $\texttt{!}\tau_2 <: \bigcup_{\sigma_1\texttt{->}\sigma_2 \in P'} \texttt{!}\sigma_2$. Also note that instead of explicitly constructing each P', for each arrow in P, $\theta^{\texttt{->}}$ performs two recursive checks—one where the arrow is in the current implicit set P' and one where it is not—which either short-circuit immediately if we can *already* confirm the condition will hold, or we remember that type by appropriately adding to the domain or range accumulator (i.e. t_1 or t_2) and we continue checking with the remaining arrows in P.

4.4 Predicate Types

The first of the three semantic metafunctions we will describe is the one which is a novel contribution of our work: $\mathsf{pred}^?$. Recall that when type checking a function application, the T-App rule uses a metafunction $\mathsf{pred}^?$ to determine what can be learned about the argument's type whether the function returns a non-`false` value or whether it returns `false`. Because the function can have an arbitrarily complex set-theoretic function type, we need a general way to compute this information for any possible function type τ_f.

For this purpose, we present a general algorithm which, given a function type τ_f and output type σ_{out}, determines the most specific type σ_{in} that an argument must have had given that it caused the function to produce a value of type σ_{out}:

$$\sigma_{in} = \mathsf{dom}(\tau_f)\texttt{\&}\left(\bigcup_{(P,N)\in\tau} \sigma_+ \backslash \sigma_-\right)$$

In this equation, for each intersection of function types (i.e. each set of arrows P in the DNF), we calculate σ_+—the type of inputs which can *possibly* be mapped to a value of type σ_{out}—and subtract σ_-—the type of inputs which *cannot possibly* be mapped to a value of type σ_{out}—and we union these results (since we cannot be certain which clause in the DNF would be responsible for producing the value). Finally, we intersect the domain of the overall function with this result since the domain of a union of functions is the intersection of the respective domains.

The types σ_+ and σ_- for each set of arrows P in τ_f are calculated as follows:

$$\sigma_+ = \bigcup_{\emptyset\subset P'\subseteq P} \begin{cases} \bigcap_{\tau_1\texttt{->}\tau_2\in P'} \tau_1 & \text{if } \left(\bigcap_{\tau_1\texttt{->}\tau_2\in P'} \tau_2\right)\texttt{\&}\sigma_{out} \not\approx \textsc{Empty} \\ \textsc{Empty} & \text{otherwise} \end{cases}$$

$$\sigma_- = \bigcup_{\emptyset\subset P'\subseteq P} \begin{cases} \bigcap_{\tau_1\texttt{->}\tau_2\in P'} \tau_1 & \text{if } \left(\bigcap_{\tau_1\texttt{->}\tau_2\in P'} \tau_2\right)\texttt{\&}\sigma_{out} \approx \textsc{Empty} \\ \textsc{Empty} & \text{otherwise} \end{cases}$$

For σ_+, we consider each possible set of arrows P', and if that set of arrows together has a range that overlaps with σ_{out}, we remember the input type which would have caused that to occur (i.e. the intersection of their domains). By doing this for all non-empty sets

$\boxed{\mathsf{input}^{?} : \mathrm{t}\ \mathrm{t} \to \mathrm{t}\ \text{or}\ \#\mathrm{f}}$

$\mathsf{input}^{?}(\mathrm{t_f}, \mathrm{s}) \;=\; \mathsf{input}(\mathrm{b}^{\text{->}}, \mathrm{t_d}, \mathrm{s}, \emptyset)$
 where $\mathrm{t_d} = \mathsf{dom}^{?}(\mathrm{t_f})$, $\langle \_, \_, \mathrm{b}^{\text{->}}\rangle = \mathrm{t_f}$

$\mathsf{input}^{?}(\mathrm{t_f}, \mathrm{s}) \;=\; \#\mathrm{f}$

$\boxed{\gamma : \mathrm{t}\ \mathrm{t}\ \mathrm{P} \to \langle \mathrm{t}, \mathrm{t}\rangle}$

$\gamma(\mathrm{s}, \mathrm{t_d}, \emptyset) \;=\; \langle \bot, \mathrm{t_d}\rangle$
 where $\mathsf{empty}(\mathrm{s})$

$\gamma(\mathrm{s}, \mathrm{t_d}, \emptyset) \;=\; \langle \mathrm{t_d}, \bot\rangle$

$\gamma(\mathrm{s}, \mathrm{t_d}, \{\mathrm{t_1}\text{->}\mathrm{t_2}\}\cup\mathrm{P}) \;=\; \langle \mathrm{s_{1+}} \cup \mathrm{s_{2+}}, \mathrm{s_{1-}} \cup \mathrm{s_{2-}}\rangle$
 where $\langle \mathrm{s_{1+}}, \mathrm{s_{1-}}\rangle = \gamma(\mathrm{s} \cap \mathrm{t_2}, \mathrm{t_d} \cap \mathrm{t_1}, \mathrm{P})$,
 $\langle \mathrm{s_{2+}}, \mathrm{s_{2-}}\rangle = \gamma(\mathrm{s}, \mathrm{t_d}, \mathrm{P})$

$\boxed{\mathsf{proj}^{?} : \mathrm{i}\ \mathrm{t} \to \mathrm{t}\ \text{or}\ \#\mathrm{f}}$

$\mathsf{proj}^{?}(\mathrm{i}, \mathrm{t}) \;=\; \mathsf{proj}(\mathrm{i}, \mathrm{b}^{*}, \top, \top, \emptyset)$
 where $\mathrm{t} <: \top^{*}$, $\langle \_, \mathrm{b}^{*}, \_\rangle = \mathrm{t}$

$\mathsf{proj}^{?}(\mathrm{i}, \mathrm{t}) \;=\; \#\mathrm{f}$

$\boxed{\phi : \mathrm{i}\ \mathrm{s}\ \mathrm{s}\ \mathrm{N} \to \mathrm{t}}$

$\phi(\mathrm{i}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N}) \;=\; \bot$
 where $\mathsf{empty}(\mathrm{s_1})$ or $\mathsf{empty}(\mathrm{s_2})$

$\phi(\mathrm{i}, \mathrm{s_1}, \mathrm{s_2}, \emptyset) \;=\; \mathsf{select}(\mathrm{i}, \mathrm{s_1}, \mathrm{s_2})$

$\phi(\mathrm{i}, \mathrm{s_1}, \mathrm{s_2}, \{\mathrm{t_1}*\mathrm{t_2}\}\cup\mathrm{N}) \;=\; \mathrm{t_l} \cup \mathrm{t_r}$
 where $\mathrm{t_l} = \phi(\mathrm{i}, \mathrm{s_1} \setminus \mathrm{t_1}, \mathrm{s_2}, \mathrm{N})$,
 $\mathrm{t_r} = \phi(\mathrm{i}, \mathrm{s_1}, \mathrm{s_2} \setminus \mathrm{t_2}, \mathrm{N})$

$\boxed{\mathsf{input} : \mathrm{b}^{\text{->}}\ \mathrm{t}\ \mathrm{t}\ \mathrm{P} \to \mathrm{t}}$

$\mathsf{input}(\mathbb{0}, \mathrm{t_d}, \mathrm{s}, \mathrm{P}) \;=\; \bot$

$\mathsf{input}(\mathbb{1}, \mathrm{t_d}, \mathrm{s}, \mathrm{P}) \;=\; \mathrm{s_+} \setminus \mathrm{s_-}$
 where $\langle \mathrm{s_+}, \mathrm{s_-}\rangle = \gamma(\mathrm{s}, \mathrm{t_d}, \mathrm{P})$

$\mathsf{input}(\langle \mathrm{t_1}\text{->}\mathrm{t_2}, \mathrm{b}^{\text{->}}_{\mathrm{l}}, \mathrm{b}^{\text{->}}_{\mathrm{m}}, \mathrm{b}^{\text{->}}_{\mathrm{r}}\rangle, \mathrm{t_d}, \mathrm{s}, \mathrm{P})$
 $=\; \mathrm{s_l} \cup \mathrm{s_m} \cup \mathrm{s_r}$
 where $\mathrm{s_l} = \mathsf{input}(\mathrm{b}^{\text{->}}_{\mathrm{l}}, \mathrm{t_d}, \mathrm{s}, \{\mathrm{t_1}\text{->}\mathrm{t_2}\}\cup\mathrm{P})$,
 $\mathrm{s_m} = \mathsf{input}(\mathrm{b}^{\text{->}}_{\mathrm{m}}, \mathrm{t_d}, \mathrm{s}, \mathrm{P})$,
 $\mathrm{s_r} = \mathsf{input}(\mathrm{b}^{\text{->}}_{\mathrm{r}}, \mathrm{t_d}, \mathrm{s}, \mathrm{P})$

$\boxed{\mathsf{select} : \mathrm{i}\ \mathrm{t}\ \mathrm{t} \to \mathrm{t}}$

$\mathsf{select}(1, \mathrm{t_1}, \mathrm{t_2}) \;=\; \mathrm{t_1}$

$\mathsf{select}(2, \mathrm{t_1}, \mathrm{t_2}) \;=\; \mathrm{t_2}$

$\boxed{\mathsf{proj} : \mathrm{i}\ \mathrm{b}^{*}\ \mathrm{s}\ \mathrm{s}\ \mathrm{N} \to \mathrm{t}}$

$\mathsf{proj}(\mathrm{i}, \mathbb{0}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N}) \;=\; \bot$

$\mathsf{proj}(\mathrm{i}, \mathrm{b}^{*}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N}) \;=\; \bot$
 where $\mathsf{empty}(\mathrm{s_1})$ or $\mathsf{empty}(\mathrm{s_2})$

$\mathsf{proj}(\mathrm{i}, \mathbb{1}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N}) \;=\; \phi(\mathrm{i}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N})$

$\mathsf{proj}(\mathrm{i}, \langle \mathrm{t_1}*\mathrm{t_2}, \mathrm{b}^{*}_{\mathrm{l}}, \mathrm{b}^{*}_{\mathrm{m}}, \mathrm{b}^{*}_{\mathrm{r}}\rangle, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N})$
 $=\; \mathrm{t_l} \cup \mathrm{t_m} \cup \mathrm{t_r}$
 where $\mathrm{t_l} = \mathsf{proj}(\mathrm{i}, \mathrm{b}^{*}_{\mathrm{l}}, \mathrm{s_1} \cap \mathrm{t_1}, \mathrm{s_2} \cap \mathrm{t_2}, \mathrm{N})$,
 $\mathrm{t_m} = \mathsf{proj}(\mathrm{i}, \mathrm{b}^{*}_{\mathrm{m}}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{N})$,
 $\mathrm{t_r} = \mathsf{proj}(\mathrm{i}, \mathrm{b}^{*}_{\mathrm{r}}, \mathrm{s_1}, \mathrm{s_2}, \{\mathrm{t_1}*\mathrm{t_2}\}\cup\mathrm{N})$

Figure 8: input type calculation

P' we cover all possibilities for that function type. σ_{-} is calculated in the same way, except we check which sets of arrows have ranges that do not overlap with σ_{out}.

In figure 8 we define a metafunction $\mathsf{input}^{?}$ which performs these calculations on our internal type representation. Given an arbitrary type t_f and type s, $\mathsf{input}^{?}$ first checks that t_f is a function by attempting to calculate its domain t_d (if this fails, #f is returned), then it proceeds to calculate what type of arguments would necessarily be mapped to values of type s by calling the input metafunction. input traverses the function BDD accumulating each set of arrows (i.e. the arrows in each conjunction in the DNF) in parameter P and at non-absurd leaves it calls metafunction γ to calculate the pair of types $\langle\sigma_{+}, \sigma_{-}\rangle$ we described previously, i.e. the union of the domains for the combinations of arrows that can produce values of type σ and which cannot respectively. input then computes the difference between the two returned types and unions this result with the results from the other collections of arrows as our original formula does.

4.5 Product Projection

Recall from our description of calculating product emptiness in section 4.3.2 that because a negated product type is logically equivalent to a disjunction of statements negating the respective field types, we had to reason about each possible combination of negations. Product projection is similar: the result of the projection from a product type DNF is the union of the results of projecting from each clause (P, N) in the DNF. For each clause (P, N), we consider the possible combinations of negated fields and, for each non-absurd case, we include the intersection of the types for that field in the result. Thus for each clause (P, N), if i is the field we are projecting, j is the other index, and σ_i and σ_j are the consolidated positive types for the respective fields, then the result of the projection is as follows:

$$\bigcup_{N' \subseteq N} \left(\begin{cases} \texttt{Empty} & \text{if } \sigma_j \texttt{\&} \left(\bigcap_{\tau_1 \texttt{*} \tau_2 \in N \setminus N'} \texttt{!}\tau_j \right) \approx \texttt{Empty} \\ \sigma_i \texttt{\&} \left(\bigcap_{\tau_1 \texttt{*} \tau_2 \in N'} \texttt{!}\tau_i \right) & \text{otherwise} \end{cases} \right)$$

The $\mathsf{proj}^?$ metafunction defined in figure 8 implements this calculation. It first checks the type is indeed a product, and then passes the product BDD to proj which, like empty^*, traverses the BDD accumulating the positive information in parameters s_1 and s_2 and the negative types in the set N. At the non-trivial leaves, it calls helper metafunction ϕ which explores the field types under each possible combination of field negations, unioning the results.

4.6 Function Application

In order to semantically determine the most specific return type for the application of a function of type τ_f to an argument of type τ_a, we must reason semantically about τ_f.

First, we verify τ_a is in the domain of τ_f. τ_f's domain is the intersection of the union of the domains in each clause in τ_f's DNF. In other words, an intersection of arrows can accept any input that is covered by the arrows in aggregate (i.e. the union of the domains), and with a union of such intersections, we can only accept arguments which would be accepted by *each* intersection (since we cannot be certain which it is).

Next—as we did when checking function type emptiness—for each clause (P, N) in the DNF we check the non-empty combinations of possible arrows $P' \subseteq P$ and determine which sets of arrows would need to handle some portion of the possible input values (i.e. when the arrows not in P' do not cover the input type), unioning each applicable range:

$$\bigcup_{\emptyset \subset P' \subseteq P} \begin{cases} \texttt{Empty} & \text{if } \tau_a \texttt{ <: } \bigcup_{\sigma_1 \texttt{->} \sigma_2 \in P \setminus P'} \sigma_1 \\ \bigcap_{\sigma_1 \texttt{->} \sigma_2 \in P'} \sigma_2 & \text{otherwise} \end{cases}$$

For each clause (P, N) in the DNF of τ_f, we union the above calculation (since any of the intersections of arrows could apply) to determine the overall result type.

In figure 9 we define the metafunction $\mathsf{apply}^?$ which performs these calculations for function type t_f and argument type t_a, returning either the specific result type or #f if t_f is not a function that can be applied to an argument of type t_a.

After checking the domain covers the input, $\mathsf{apply}^?$ calls helper function apply whose parameters are the argument type, an accumulator for calculating the range of the various

$$\boxed{\mathsf{apply}^{?} : \mathrm{t}\ \mathrm{t} \rightarrow \mathrm{t}\ \text{or}\ \#\mathrm{f}}$$

$$\mathsf{apply}^{?}(\mathrm{t_f}, \mathrm{t_a}) \;=\; \mathsf{apply}(\mathrm{t_a}, \top, \mathrm{b}^{\text{->}})$$
$$\text{where } \mathrm{t_d} = \mathsf{dom}^{?}(\mathrm{t_f}),\ \mathrm{t_a} <: \mathrm{t_d},$$
$$\langle \_, \_, \mathrm{b}^{\text{->}} \rangle = \mathrm{t_f}$$
$$\mathsf{apply}^{?}(\mathrm{t_f}, \mathrm{t_a}) \;=\; \#\mathrm{f}$$

$$\boxed{\mathsf{apply} : \mathrm{t}\ \mathrm{t}\ \mathrm{b}^{\text{->}} \rightarrow \mathrm{t}}$$

$$\mathsf{apply}(\mathrm{t_a}, \mathrm{t}, \mathbb{0}) \;=\; \bot$$
$$\mathsf{apply}(\mathrm{t_a}, \mathrm{t}, \mathrm{b}^{\text{->}}) \;=\; \bot$$
$$\text{where } \mathsf{empty}(\mathrm{t_a})$$
$$\mathsf{apply}(\mathrm{t_a}, \mathrm{t}, \mathbb{1}) \;=\; \mathrm{t}$$
$$\mathsf{apply}(\mathrm{t_a}, \mathrm{t}, \langle \mathrm{s_1}\text{->}\mathrm{s_2}, \mathrm{b}^{\text{->}}_{\mathrm{l}}, \mathrm{b}^{\text{->}}_{\mathrm{m}}, \mathrm{b}^{\text{->}}_{\mathrm{r}} \rangle)$$
$$=\; \mathrm{t_{l1}} \cup \mathrm{t_{l2}} \cup \mathrm{t_m} \cup \mathrm{t_r}$$
$$\text{where } \mathrm{t_{l1}} = \mathsf{apply}(\mathrm{t_a}, \mathrm{t} \cap \mathrm{s_2}, \mathrm{b}^{\text{->}}_{\mathrm{l}}),$$
$$\mathrm{t_{l2}} = \mathsf{apply}(\mathrm{t_a} \setminus \mathrm{s_1}, \mathrm{t}, \mathrm{b}^{\text{->}}_{\mathrm{l}}),$$
$$\mathrm{t_m} = \mathsf{apply}(\mathrm{t_a}, \mathrm{t}, \mathrm{b}^{\text{->}}_{\mathrm{m}}),$$
$$\mathrm{t_r} = \mathsf{apply}(\mathrm{t_a}, \mathrm{t}, \mathrm{b}^{\text{->}}_{\mathrm{r}})$$

$$\boxed{\mathsf{dom}^{?} : \mathrm{t} \rightarrow \mathrm{t}\ \text{or}\ \#\mathrm{f}}$$

$$\mathsf{dom}^{?}(\mathrm{t}) \;=\; \mathsf{dom}(\bot, \mathrm{b}^{\text{->}})$$
$$\text{where } \mathrm{t} <: \top^{\text{->}},\ \langle \_, \_, \mathrm{b}^{\text{->}} \rangle = \mathrm{t}$$
$$\mathsf{dom}^{?}(\mathrm{t}) \;=\; \#\mathrm{f}$$

$$\boxed{\mathsf{dom} : \mathrm{t}\ \mathrm{b}^{\text{->}} \rightarrow \mathrm{t}}$$

$$\mathsf{dom}(\mathrm{t}, \mathbb{1}) \;=\; \mathrm{t}$$
$$\mathsf{dom}(\mathrm{t}, \mathbb{0}) \;=\; \top$$
$$\mathsf{dom}(\mathrm{t}, \langle \mathrm{s_1}\text{->}\mathrm{s_2}, \mathrm{b}^{\text{->}}_{\mathrm{l}}, \mathrm{b}^{\text{->}}_{\mathrm{m}}, \mathrm{b}^{\text{->}}_{\mathrm{r}} \rangle) \;=\; \mathrm{t_l} \cap \mathrm{t_m} \cap \mathrm{t_r}$$
$$\text{where } \mathrm{t_l} = \mathsf{dom}(\mathrm{t} \cup \mathrm{s_1}, \mathrm{b}^{\text{->}}_{\mathrm{l}}),$$
$$\mathrm{t_m} = \mathsf{dom}(\mathrm{t}, \mathrm{b}^{\text{->}}_{\mathrm{m}}),\ \mathrm{t_r} = \mathsf{dom}(\mathrm{t}, \mathrm{b}^{\text{->}}_{\mathrm{r}})$$

Figure 9: metafunctions for function application calculations

sets of arrows as we traverse the function BDD, and the function BDD itself. apply then walks the BDD in the standard fashion in order to explore each possible set of arrows P' for each clause's set of arrows P. Note that apply does not explicitly construct each set P', rather it uses the first two parameters to accumulatively track whether the arrows not in the current set cover the input (i.e. by subtracting from the first parameter appropriately at recursive calls) and to accumulate the range for the current set of arrows so they can be included in the non-absurd base cases of the traversal when appropriate. The traversal, which is exploring each $\emptyset \subset P' \subseteq P$ for each P, unions the results for each clause and each P' to construct the appropriate result.

5 SOUNDNESS

In this section we introduce the operation semantics and an outline the proof of soundness.

5.1 Operational Semantics

The operational semantics for λ_{so} are described in figure 10 and figure 11 with a standard big-step relation. This relation, written $\rho \vdash e \Downarrow v$, states that in runtime environment ρ expression e evaluates to value v. We omit the definition for δ^2, which performs the obvious respective mathematical operations on integers.

$$\boxed{\rho \vdash e \Downarrow v}$$

$$\textsc{B-Const}\ \frac{}{\rho \vdash c \Downarrow c} \qquad \textsc{B-Abs}\ \frac{}{\rho \vdash (\lambda\{\tau, ...\}(x)\ e) \Downarrow [\rho, (\lambda\{\tau, ...\}(x)\ e)]}$$

$$\textsc{B-Var}\ \frac{v = \rho(x)}{\rho \vdash x \Downarrow v} \qquad \textsc{B-AppUOp}\ \frac{\rho \vdash e_1 \Downarrow uop \qquad \rho \vdash e_2 \Downarrow v_2 \qquad v = \delta^1(uop, v_2)}{\rho \vdash (e_1\ e_2) \Downarrow v} \qquad \textsc{B-AppClosure}\ \frac{\rho \vdash e_1 \Downarrow [\rho_c, (\lambda\{\tau, ...\}\ (x)\ e_c)] \qquad \rho \vdash e_2 \Downarrow v_2 \qquad \rho_c[x \mapsto v_2] \vdash e_c \Downarrow v}{\rho \vdash (e_1\ e_2) \Downarrow v}$$

$$\textsc{B-BOp}\ \frac{\rho \vdash e_1 \Downarrow \mathit{int}_1 \qquad \rho \vdash e_2 \Downarrow \mathit{int}_2}{\rho \vdash (bop\ e_1\ e_2) \Downarrow \delta^2(bop, \mathit{int}_1, \mathit{int}_2)} \qquad \textsc{B-Proj}\ \frac{\rho \vdash e \Downarrow (\texttt{pair}\ v_1\ v_2)}{\rho \vdash (\texttt{proj}\ i\ e) \Downarrow v_i} \qquad \textsc{B-Pair}\ \frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash (\texttt{pair}\ e_1\ e_2) \Downarrow (\texttt{pair}\ v_1\ v_2)}$$

$$\textsc{B-IfNonFalse}\ \frac{\rho \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \texttt{false} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow v} \qquad \textsc{B-IfFalse}\ \frac{\rho \vdash e_1 \Downarrow \texttt{false} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow v} \qquad \textsc{B-Let}\ \frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho[x \mapsto v_1] \vdash e_2 \Downarrow v}{\rho \vdash (\texttt{let}\ (x\ e_1)\ e_2) \Downarrow v}$$

Figure 10: big step reduction semantics

$$\boxed{\rho \vDash p}$$

$$\textsc{M-True}\ \frac{}{\rho \vDash \mathbb{tt}} \qquad \textsc{M-Alias}\ \frac{v = \rho(x) \qquad v = \rho(\pi)}{\rho \vDash x \rightsquigarrow \pi} \qquad \textsc{M-Type}\ \frac{v = \rho(\pi) \qquad \emptyset \vdash v : \tau}{\rho \vDash \pi \in \tau}$$

$$\textsc{M-And}\ \frac{\rho \vDash p \qquad \rho \vDash q}{\rho \vDash p \wedge q} \qquad \textsc{M-Or1}\ \frac{\rho \vDash p}{\rho \vDash p \vee q} \qquad \textsc{M-Or2}\ \frac{\rho \vDash q}{\rho \vDash p \vee q}$$

$$\textsc{T-Closure}\ \frac{\rho \vDash \Gamma_c \qquad \Gamma_c, x \in \tau_i \vdash e : \sigma_i \qquad ...}{\Gamma \vdash [\rho, (\lambda\{\tau_i, ...\}(x)\ e)] : \langle(\tau_i\text{->}\sigma_i)\&..., \mathbb{tt}, \mathbb{ff}, \top^o\rangle}$$

$$\boxed{\delta^1 : uop\ v \to v \text{ or } \#f}$$

$$\begin{aligned}
&\delta^1(\texttt{zero?}, 0) &&= \texttt{true}\\
&\delta^1(\texttt{zero?}, \mathit{int}) &&= \texttt{false}\\
&\delta^1(\texttt{strlen}, \mathit{str}) &&= |\mathit{str}|\\
&\delta^1(\texttt{not}, \texttt{false}) &&= \texttt{true}\\
&\delta^1(\texttt{not}, \_) &&= \texttt{false}\\
&\delta^1(\texttt{int?}, \mathit{int}) &&= \texttt{true}\\
&\delta^1(\texttt{int?}, \_) &&= \texttt{false}\\
&\delta^1(\texttt{str?}, \mathit{str}) &&= \texttt{true}\\
&\delta^1(\texttt{str?}, \_) &&= \texttt{false}\\
&\delta^1(\texttt{pair?}, (\texttt{pair}\ \_\ \_)) &&= \texttt{true}\\
&\delta^1(\texttt{pair?}, \_) &&= \texttt{false}\\
&\delta^1(\texttt{fun?}, [\rho, (\lambda\{\tau, ...\}(x)\ e)]) &&= \texttt{true}\\
&\delta^1(\texttt{fun?}, uop) &&= \texttt{true}\\
&\delta^1(\texttt{fun?}, \_) &&= \texttt{false}\\
&\delta^1(\_, \_) &&= \#f
\end{aligned}$$

Figure 11: satisfaction, closure typing, and unary op. semantics

5.2 Type Soundness

For our proof of soundness we use the same model theoretic technique introduced by Tobin-Hochstadt and Felleisen (2010). We first give a typing rule for closures and a notion of satisfiability which indicates whether a proposition is necessarily made true by a given runtime environment ρ (both appear in figure 11). The satisfaction relation, written $\rho \vDash p$ ("ρ satisfies p"), resembles standard phrasings of satisfiability from propositional logic.

From the primary lemma—which uses the runtime environment and satisfaction to precisely describe the meaning of the typing judgment—we can prove soundness as a corollary:

Lemma 5.1. *If* $\Gamma \vdash e : \langle \tau, p, q, o \rangle$, $\rho \vDash \Gamma$, *and* $\rho \vdash e \Downarrow v$ *then all of the following hold:*
(1) either $o = \top^o$ *or* $\exists \pi$ *such that* $o = \pi$ *and* $\rho(\pi) = v$,
(2) either $v \neq \texttt{false}$ *and* $\rho \vDash p$, *or* $v = \texttt{false}$ *and* $\rho \vDash q$, *and*
(3) $\vdash v : \tau$

Proof. By induction on the derivation $\rho \vdash e \Downarrow v$. □

Theorem 5.2 (Type Soundness). *If* $\vdash e : \tau$ *and* $\vdash e \Downarrow v$ *then* $\vdash v : \tau$.

Proof. Corollary of lemma 5.1. □

6 RELATED WORK

Many previous projects have dealt with occurrence typing and semantic set-theoretic types; we discuss these (not mutually exclusive) approaches here and how our work relates.

6.1 Syntactically Recognizing Predicates

A common approach to support occurrence typing involves simply recognizing syntactic type-tests that appear in conditional test-expressions and propagating this information into the respective branches (Ceylon Project 2017; Guha et al. 2011; JetBrains 2017; Komondoor et al. 2005; Microsoft 2017; Pearce 2013; Groovy 2017). A similar but more sophisticated approach involves syntactically recognize the types implied by ML-like pattern matching statements and updating the type environment appropriately based on whether or not they matched (Castagna et al. 2016). These techniques are mostly straightforward to reason about and implement, however without *additional features* they are not compositional and thus cannot be abstracted over in any meaningful way. Our approach—which instead of looking for special syntactic forms reasons semantically about the types of functions—allows arbitrary abstractions to be effectively treated as predicates because of the rich type specifications made possible by the full range of set-theoretic types.

6.2 Expressing Predicates with Dependent Types

Another common approach for supporting occurrence typing involves using some form of dependent types (often refinement types or an equivalent mechanism) to allow a function's return type to witness how returned values logically relate to other program terms (Bonnaire-Sergeant and Tobin-Hochstadt 2016; Chaudhuri et al. 2017; Chugh et al. 2012a; Chugh et al. 2012b; Lerner et al. 2013; Microsoft 2017; Tobin-Hochstadt and Felleisen 2010; Vekris et al. 2015). These approaches are generally more expressive than ours w.r.t. the program invariants the allow, but often heavily rely on SMT solvers and complex program transformations. Our approach instead seeks to be a foundation for occurrence typing based solely on standard semantic set-theoretic types and is more likely suitable for a language already biased towards such a foundation for types.

6.3 Semantic Set-theoretic Types and Type-based Dispatch

Much of the work which pioneered recent advances in semantic set-theoretic types has featured type-based dispatch (Castagna and Lanvin 2017; Castagna et al. 2015; Castagna et al. 2014; Frisch et al. 2008). These languages either dispatch by type checking values during program execution, or use a syntactic construct (e.g. a lambda) to introduce a *new* identifier (or set of identifiers) which an overloaded type describes the possible use-cases for. These features can certainly be used to write programs *similar* to those featuring occurrence typing, but they are not the same: requiring a type checker to inspect values at run time is simply not feasible for many languages which use occurrence typing today and requiring the introduction of an overloaded lambda (or similar) for type-based dispatch does not allow any of the examples we have shown to be type checked as-is. We do, however, believe that our approach to occurrence typing is entirely compatible with this body of work.

BIBLIOGRAPHY

Ambrose Bonnaire-Sergeant and Sam Tobin-Hochstadt. Practical Optional Types for Clojure. In *Proc. European Sym. on Programming*, pp. 68–94, 2016.

Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 1993.

Robert Cartwright. User-defined data types as an aid to verifying LISP programs. In *Proc. Intl. Colloquium on Automata, Languages, and Programming*, pp. 228–256, 1976.

Giuseppe Castagna. Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). 2015. Unpublished manuscript

Giuseppe Castagna and Victor Lanvin. Gradual Typing with Union and Intersection Types. *Proc. ACM Programming Lang.* 1(ICFP), pp. 41:1–41:28, 2017.

Giuseppe Castagna, K. Nguyễn, Z. Xu, and P. Abate. Polymorphic Functions with Set-Theoretic Types. Part 2: Local Type Inference and Type Reconstruction. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 289–302, 2015.

Giuseppe Castagna, K. Nguyễn, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 5–17, 2014.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. Set-Theoretic Types for Polymorphic Variants. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 378–391, 2016.

Ceylon Project. The Ceylon Language. 2017. https://ceylon-lang.org/documentation/1.3/spec/

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proc. ACM Programming Lang.* 1(OOPSLA), pp. 48:1–48:30, 2017.

Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 587–606, 2012a.

Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 231–244, 2012b.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55(4), pp. 19:1–19:64, 2008.

Groovy. 2017. http://groovy-lang.org

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *Proc. European Conf. Object-Oriented Programming*, pp. 126–150, 2010.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *Proc. European Sym. on Programming*, pp. 256–275, 2011.

JetBrains. Kotlin. 2017. http://kotlinlang.org/docs/reference/

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proc. ACM Sym. Principles of Programming Languages*, 2012.

Raghavan Komondoor, G. Ramalingam, Satish Chandra, and John Field. Dependent Types for Program Understanding. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 157–173, 2005.

Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proc. Dynamic Languages Symposium*, pp. 1–16, 2013.

Microsoft. TypeScript. 2017. http://www.typescriptlang.org/

David J. Pearce. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Proc. International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 335–354, 2013.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 395–406, 2008.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 117–128, 2010.

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In *Proc. European Conf. Object-Oriented Programming*, pp. 52–75, 2015.