Sound and Complete Flow Typing with Unions, Intersections and Negations

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
{djp}@ecs.vuw.ac.nz

August 2012

Abstract

Flow typing is becoming a popular mechanism for typing existing programs written in untyped languages (e.g. JavaScript, Racket, Groovy). Such systems require intersections for the true-branch of a type test, negations for the false-branch, and unions to capture the flow of information at meet points. Type systems involving unions, intersections and negations require a subtype operator which is non-trivial to implement. Frisch *et al.* demonstrated that this problem was decidable. However, their proof was not constructive and does not lend itself naturally to an implementation. In this paper, we present a sound and complete algorithm for subtype testing in the presence of unions, intersections and negations.

1 Introduction

Statically typed programming languages lead to programs which are more efficient and where errors are easier to detect ahead-of-time [1, 2]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. In contrast, dynamically typed languages are more flexible in nature which helps reduce overheads and increase productivity [3–6].

A common complaint against statically typed languages is the need for often unnecessarily verbose type declarations. Hindley-Milner Type inference [7, 8] is a common approach to addressing this problem, where type declarations are inferred automatically. Scala [9], C#3.0 [10] and OCaml [11] provide good examples of this in an imperative setting. However, such languages still require each program variable to have exactly one type. Flow typing offers an alternative to Hindley-Milner type inference where a variable may have different types at different program points. The technique is adopted from flow-sensitive program analysis and has been used for non-null types [12–15], information flow [16–18], purity checking [19] and more [12, 13, 20–25].

Few languages exist which incorporate flow typing directly. Typed Racket [23, 26] provides a *typed* sister language for *untyped* Racket, where flow typing is used to capture common idioms in the untyped language. Similarly, the recent 2.0 release of the popular Groovy language includes a flow typing algorithm [27]. Again, this is designed to handle common idioms in (previously) untyped Groovy programs. Finally, the Whiley language employs flow-typing to give it the look-and-feel of an untyped language [28, 29].

1.1 Flow Typing

A defining characteristic of flow typing is the ability to *retype* a variable — that is, assign it a completely unrelated type. The JVM Bytecode Verifier [30], perhaps the most widely-used example of a flow typing system, provides a good illustration:

```
public static float convert(int):
  iload 0    // load register 0 on stack
  i2f        // convert int to float
  fstore 0   // store float to register 0
  fload 0    // load register 0 on stack
  freturn    // return value on stack
```

In the above, register 0 contains the parameter value on entry and, initially, has type **`int`**. The type of register 0 is subsequently changed to `float` by the `fstore` bytecode. To ensure type safety, the JVM bytecode verifier employs a typing algorithm based upon dataflow analysis [31]. This tracks the type of a variable at each program point, allowing it easily to handle the above example.

Flow typing can also retype variables after conditionals. A *non-null type system* (e.g. [12–15]) prevents variables which may hold **`null`** from being dereferenced. The following illustrates:

```
int cmp(String s1, @NonNull String s2) {
  if(s1 != null) {
    return s1.compareTo(s2);
  } else {
    return -1;
  }
}
```

The modifier `@NonNull` indicates a variable definitely cannot hold **`null`** and, hence, that it can be safely dereferenced. To deal with the above example, a non-null type system will retype variable `s1` to `@NonNull` on the true branch — thus allowing it to type check the subsequent dereference of `s1`.

Whiley [28, 29, 32] employs a flow type system to give it the look-and-feel of a dynamically typed language. Variable retyping through conditionals is supported using the **`is`** operator (similar to `instanceof` in Java) as follows:

```
define Circle as {int x, int y, int r}
define Rect as {int x, int y, int w, int h}
define Shape as Circle | Rect

real area(Shape s):
    if s is Circle:
        return PI * s.r * s.r
    else:
        return s.w * s.h
```

A `Shape` is either a `Rect` or a `Circle` (which are both record types). The type test "`s` **`is`** `Circle`" determines whether `s` is a `Circle` or not. Unlike Java, Whiley automatically retypes `s` to have type `Circle` (resp. `Rect`) on the true (resp. false) branches of the **`if`** statement. There is no need to explicitly cast `s` to the appropriate `Shape` before accessing its fields.

1.2 Unions, Intersections and Negations

Union types (e.g. $T_1 \vee T_2$) are commonly used in flow typing systems to capture the type of variables at meet points. For example, consider this code snippet:

```
if ...:
    x = 1
else:
    x = true
...
```

After the assignment `x=1`, the type of variable `x` is `int`. Likewise, after the assignment `x=true` it is `bool`. Finally, `x` has type $\texttt{int} \vee \texttt{bool}$ immediately after the **`if`** statement (i.e. at the meet point). This indicates `x` can hold either an `int` *or* a `bool` at that point.

Retyping variables after runtime type tests is typically achieved through a type system which supports both *intersections* (e.g. $T_1 \wedge T_2$) and *negations* (e.g. $\neg T_1$). For example, consider again this code snippet:

```
real area(Shape s):
    if s is Circle:
        return PI * s.r * s.r
    else:
        return s.w * s.h
```

To determine the type of variable `s` on the true branch, we *intersect* its declared type (i.e. `Shape`) with the type test (i.e. `Circle`). Likewise, on the false branch, we compute the difference of these two types (i.e. `Shape - Circle`). Observe that the difference of two types in such a system is given by: $\mathtt{T}_1 - \mathtt{T}_2 \equiv \mathtt{T}_1 \wedge \neg\mathtt{T}_2$.

1.3 Contributions

Subtype testing (i.e. establishing whether $\mathtt{T}_1 \leq \mathtt{T}_2$ holds or not) is a challenging algorithmic problem for a type system involving unions, intersections and negations. In particular, we desire that subtyping is both *sound* and *complete* with respect to a semantic model where types are viewed as sets. The former requires $\mathtt{T}_1 \leq \mathtt{T}_2$ holds *only* when $\mathtt{T}_1$ is a subset of $\mathtt{T}_2$, whilst the latter requires that $\mathtt{T}_1 \leq \mathtt{T}_2$ holds *whenever* $\mathtt{T}_1$ is a subset of $\mathtt{T}_2$. Frisch *et al.* demonstrated that this problem was decidable [33]. However, their proof was not constructive and does not lend itself naturally to an implementation. In this paper, we present a sound and complete algorithm for subtyping in the presence of unions, intersections and negations. This contrasts with previous flow type systems (e.g. [23, 26]) which are shown sound, but not complete.

2 A Flow-Typing Calculus — FT

We now introduce our flow-typing calculus, FT, within which we frame our flow typing problem. The calculus is specifically kept to a minimum to allow us to succinctly capture the important issues. In this section, we introduce the syntax, semantics and subtyping rules for FT. We tacitly assume at this point that an appropriate subtyping operator exists. Subsequently, in §3 and §4, we will detail the algorithms which implement this operator (and which are the core contribution of this paper).

2.1 Types

The following gives a *syntactic* definition of types in FT:

$$\mathtt{T} ::= \mathtt{any} \mid \mathtt{int} \mid (\mathtt{T}_1, \ldots, \mathtt{T}_n) \mid \neg\mathtt{T} \mid \mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n \mid \mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n$$

Here, **`any`** represents $\top$, **`int`** the set of all integers and $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ represents tuples with one or more elements. The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. Union types are generally useful in flow typing systems, as they can characterise types generated at meet points in the control-flow graph. The intersection $\mathtt{T}_1 \wedge \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *and* $\mathtt{T}_2$. Intersections are needed in our flow type system to capture the type of a variable (e.g. `x`) after a type test (e.g. `x is T`). The type $\neg\mathtt{T}$ is the *negation* type containing those values *not* in `T`. Thus, $\neg\mathtt{any}$ represents $\bot$ (i.e. the empty set) and we will often write `void` as a short-hand for this. Negations are also useful for capturing the type of a variable on the false branch of a type test. Finally, we make some simplifying assumptions regarding unions and intersections: *namely, that elements are unordered and duplicates are removed.* Thus, $\mathtt{T}_1 \vee \mathtt{T}_2$ is indistinguishable from $\mathtt{T}_2 \vee \mathtt{T}_1$. Likewise, $\mathtt{T}_1 \vee \mathtt{T}_1$ is not distinguishable from $\mathtt{T}_1$. Whilst these simplifications are not strictly necessary, they simplify our presentation. Furthermore, they can be implemented easily enough by sorting elements according to a fixed total ordering of types.

To better understand the meaning of types in FT, it is helpful to give a *semantic interpretation* (following e.g. [33–36]). The aim is to give a set-theoretic model where subtype corresponds to subset. The *domain* $\mathbb{D}$ of values in our model consists of the integers and all records constructible from values in $\mathbb{D}$:

$$\mathbb{D} = \mathbb{Z} \cup \Big\{ (\mathtt{v}_1, \ldots, \mathtt{v}_n) \mid \ \mathtt{v}_1 \in \mathbb{D}, \ldots, \mathtt{v}_n \in \mathbb{D} \Big\}$$

Syntax:

$\mathtt{t} ::=$	*terms:*
$\mathtt{x}$	*variable*
$(\mathtt{t}_1, \ldots, \mathtt{t}_n)$	*tuple*
$\mathtt{f}\ \mathtt{t}_1$	*application*
$\mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$	*declaration*
$\mathtt{if}(\mathtt{x}\ \mathtt{is}\ \mathtt{T})\ \mathtt{t}_1\ \mathtt{else}\ \mathtt{t}_2$	*type test*
$\mathtt{v} ::=$	*values:*
$\mathtt{i}$	*integer*
$(\mathtt{v}_1, \ldots, \mathtt{v}_n)$	*tuple*

Operational Semantics:

$$\frac{\Delta \vdash \mathtt{t}_k \longrightarrow \mathtt{t}'_k}{\Delta \vdash (\ldots, \mathtt{t}_k, \ldots) \longrightarrow (\ldots, \mathtt{t}'_k, \ldots)} \quad \text{(E-TUP)}$$

$$\frac{\Delta \vdash \mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\Delta \vdash \mathtt{f}\ \mathtt{t}_1 \longrightarrow \mathtt{f}\ \mathtt{t}'_1} \quad \text{(E-APP1)}$$

$$\frac{\Delta(\mathtt{f}) = \langle \mathtt{T}, \mathtt{x}, \mathtt{t}_2 \rangle \quad \mathtt{v}_1 \in [\![\mathtt{T}]\!]}{\Delta \vdash \mathtt{f}\ \mathtt{v}_1 \longrightarrow \mathtt{t}_2[\mathtt{x} \mapsto \mathtt{v}_1]} \quad \text{(E-APP2)}$$

$$\frac{\Delta[\mathtt{f} \mapsto \langle \mathtt{T}, \mathtt{x}, \mathtt{t}_1 \rangle] \vdash \mathtt{t}_2 \longrightarrow \mathtt{t}'_2}{\Delta \vdash \mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow \mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}'_2} \quad \text{(E-DEC1)}$$

$$\frac{}{\Delta \vdash \mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{v}_2 \longrightarrow \mathtt{v}_2} \quad \text{(E-DEC2)}$$

$$\frac{\mathtt{v}_1 \in [\![\mathtt{T}]\!]}{\Delta \vdash \mathtt{if}(\mathtt{v}_1\ \mathtt{is}\ \mathtt{T})\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_2} \quad \text{(E-IF1)}$$

$$\frac{\mathtt{v}_1 \notin [\![\mathtt{T}]\!]}{\Delta \vdash \mathtt{if}(\mathtt{v}_1\ \mathtt{is}\ \mathtt{T})\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_3} \quad \text{(E-IF2)}$$

Figure 1: Syntax and (small-step) operational semantics for FT. Here, $\mathtt{n}, \mathtt{m}$ and $\mathtt{i}$ represent variable identifiers, whilst $\mathcal{I}$ represents the set of integers.

Definition 1 (Type Semantics) *Every type* T *is characterized by the set of values it accepts, given by* $[\![\mathtt{T}]\!]$ *and defined as follows:*

$$\begin{aligned}
[\![\mathtt{any}]\!] &= \mathbb{D} \\
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!] &= \{(\mathtt{v}_1, \ldots, \mathtt{v}_n) \mid \mathtt{v}_1 \in [\![\mathtt{T}_1]\!], \ldots, \mathtt{v}_n \in [\![\mathtt{T}_n]\!]\} \\
[\![\neg\mathtt{T}]\!] &= \mathbb{D} - [\![\mathtt{T}]\!] \\
[\![\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cap \ldots \cap [\![\mathtt{T}_n]\!] \\
[\![\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cup \ldots \cup [\![\mathtt{T}_n]\!]
\end{aligned}$$

It is important to distinguish the *syntactic* representation from the *semantic* model of types. The former corresponds (roughly speaking) to a physical machine representation, whist the latter is a mathematical ideal. As such, the syntactic representation diverges from the semantic model and, to compensate, we must establish a correlation between them. For example int and $\neg\neg$int have distinct syntactic representations, but are semantically indistinguishable. Similarly for $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ and $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$.

Ultimately, we want to construct a subtyping algorithm that is both *sound* and *complete* (i.e. that $\mathtt{T}_1 \leq \mathtt{T}_2 \iff [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$). The distinction between syntactic and semantic forms presents a significant challenge in doing this.

2.2 Syntax & Semantics

Figure 1 gives the syntax of FT along with a small-step operational semantics, where $\Delta[\mathtt{f} \mapsto \langle \mathtt{T}, \mathtt{x}, \mathtt{t} \rangle]$ returns Δ with f now mapped to a triple $\langle \mathtt{T}, \mathtt{x}, \mathtt{t} \rangle$ representing its declaration. Similarly, $\mathtt{t}[\mathtt{x} \mapsto \mathtt{v}]$ returns the term t with all occurrences of x now substituted with v. To avoid issues of variable capture, we assume parameter names are unique and may only occur within their function body (i.e. that for $\mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ parameter x can only occur in $\mathtt{t}_1$).

From the figure, we see that a semantic notion of type is explicitly required for the operational semantics (as e.g. E-APP2 uses $[\![\mathtt{T}]\!]$). Thus, the semantic notion of execution is separated from the algorithmic notion of typing — and our goal in developing a complete subtyping algorithm is to ensure as many correct programs as possible are typeable. Finally, The reader may be surprised to see that FT does not include a first-class notion of function value (i.e. a term of the form $\lambda\mathtt{x}.\mathtt{t}$). This avoids a well-known problem of circularity in the definitions (i.e. where the semantic definition of types depends on the operational semantics and vice-versa) [33, 36, 37]. Instead, functions are declared explicitly and a runtime environment, Δ, is used to maintain the mapping from declared functions to their bodies.

An example FT program and its evaluation is given below:

$$\begin{array}{l}
\mathtt{f(any\ x) = if(x\ is\ int)\ 1\ else\ 0} \\
\qquad \mathtt{in\ (f\ 1, f\ (1,2))} \\[1em]
\hookrightarrow \mathtt{f(any\ x) = if(x\ is\ int)\ 1\ else\ 0} \\
\qquad \mathtt{in\ (if(1\ is\ int)\ 1\ else\ 0, f\ (1,2))} \\[1em]
\hookrightarrow \mathtt{f(any\ x) = if(x\ is\ int)\ 1\ else\ 0} \\
\qquad \mathtt{in\ (1, f\ (1,2))} \\[1em]
\hookrightarrow \mathtt{f(any\ x) = if(x\ is\ int)\ 1\ else\ 0} \\
\qquad \mathtt{in\ (1, if((1,2)\ is\ int)\ 1\ else\ 0)} \\[1em]
\hookrightarrow \mathtt{f(any\ x) = if(x\ is\ int)\ 1\ else\ 0} \\
\qquad \mathtt{in\ (1,0)} \\[1em]
\hookrightarrow \mathtt{(1,0)}
\end{array}$$

This example illustrates a few interesting aspects of Figure 1. Firstly, for simplicity, the order of evaluation for tuples is undefined under E-TUP. This could easily be specified, but is not important here. Secondly, the term $\mathtt{if(x\ is\ T)\ t_1\ else\ t_2}$ implements a runtime type test (similar to e.g. Java's `instanceof` operator). The left-hand side of this operator is restricted to a variable, rather than a general term. This succinctly captures the problem of retyping a variable within the true (resp. false) branches of the conditional.

2.3 Flow-Typing Rules

The flow-typing rules are given in Figure 2. These are presented as judgements of the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$, which can be read as saying: *term* t *can be shown to have type* T *under environment* Γ. The environment maps variable names to their current type, and also function names to a pair $\mathtt{T_1} \rightarrow \mathtt{T_2}$ capturing the declared parameter and inferred return type. For simplicity, we assume that function names and parameter names do not intersect.

Rules T-INT, T-VAR and T-TUP are straightforward and do not warrant further discussion. The remaining rules are more interesting, and we now consider them in more detail:

- Rule T-APP. For a function application, the type of the argument is determined recursively, whilst the function's declared parameter and inferred return types are obtained from the environment. The rule checks the argument type (i.e. $\mathtt{T_1}$) is a subtype of the declared parameter type (i.e. $\mathtt{T_2}$) using the subtype operator (i.e. $\mathtt{T_1} \leq \mathtt{T_2}$). The subtype operator will be discussed in more detail below.

- Rule T-DEC. For a function declaration, the return type is inferred by typing the body (i.e. $\mathtt{t_2}$) using the current environment updated to map the parameter (i.e. x) to its declared type (i.e. $\mathtt{T_1}$). Using this, the type of the outer term (i.e. $\mathtt{t_3}$) is then determined. Observe that, under this rule, recursive function calls cannot be typed as f is not included when typing $\mathtt{t_2}$ — however, this is of little relevance to the problem being addressed.

- Rule T-IF. For a type test, the true and false branches are typed using updated environments. For the true branch, the variable being tested (i.e. x) is mapped to the intersection of its

Flow-Typing:

$$\frac{\mathtt{v} \in \mathbb{Z}}{\Gamma \vdash \mathtt{v} \ :\ \mathtt{int}} \qquad \text{(T-INT)}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} \ :\ \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{t}_n : \mathtt{T}_n}{\Gamma \vdash (\mathtt{t}_1, \ldots, \mathtt{t}_b) \ :\ (\mathtt{T}_1, \ldots, \mathtt{T}_n)} \qquad \text{(T-TUP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \\ \Gamma(\mathtt{f}) = \mathtt{T}_2 \rightarrow \mathtt{T}_3 \quad \mathtt{T}_1 \leq \mathtt{T}_2\end{array}}{\Gamma \vdash \mathtt{f}\ \mathtt{t}_1 \ :\ \mathtt{T}_3} \qquad \text{(T-APP)}$$

$$\frac{\begin{array}{c}\Gamma[\mathtt{x} \mapsto \mathtt{T}_1] \vdash \mathtt{t}_2 : \mathtt{T}_2 \\ \Gamma[\mathtt{f} \mapsto \mathtt{T}_1 \rightarrow \mathtt{T}_2] \vdash \mathtt{t}_3 : \mathtt{T}_3\end{array}}{\Gamma \vdash \mathtt{f}(\mathtt{T}_1\ \mathtt{x}) = \mathtt{t}_2\ \mathtt{in}\ \mathtt{t}_3 \ :\ \mathtt{T}_3} \qquad \text{(T-DEC)}$$

$$\frac{\begin{array}{c}\Gamma[\mathtt{x} \mapsto \Gamma(\mathtt{x}) \wedge \mathtt{T}_1] \vdash \mathtt{t}_2 : \mathtt{T}_2 \\ \Gamma[\mathtt{x} \mapsto \Gamma(\mathtt{x}) \wedge \neg \mathtt{T}_1] \vdash \mathtt{t}_3 : \mathtt{T}_3\end{array}}{\Gamma \vdash \mathtt{if}(\mathtt{x}\ \mathtt{is}\ \mathtt{T}_1)\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}_2 \vee \mathtt{T}_3} \qquad \text{(T-IF)}$$

Figure 2: Flow-typing rules for FT.

current type and that of the type test (i.e. to $\Gamma(\mathtt{x}) \wedge \mathtt{T}_1$) — this captures the fact that its values are known to be in both $\Gamma(\mathtt{x})$ and $\mathtt{T}_1$. Similarly, for the false branch, the variable being tested is mapped to the intersection of its current type and that of the negated type test (i.e. to $\Gamma(\mathtt{x}) \wedge \neg\mathtt{T}_1$) — this captures the fact that its values are known to be in $\Gamma(\mathtt{x})$ but not in $\mathtt{T}_1$. The resulting type of the type test is then the most precise type which includes the types determined for each branch (i.e. $\mathtt{T}_2 \vee \mathtt{T}_3$).

Observe that, in rule T-IF, the type of the tested variable may be determined as `void` for either branch — which, in such case, indicates that branch is unreachable. A modern compiler would most likely report such a situation as a syntax error (but this is an orthogonal issue).

Discussion. Having considered the flow-typing rules, we can now consider why certain constructs are included in our calculus. Firstly, function application is included since T-App requires a subtype test. Without this construct, there is no need for a subtyping algorithm such as presented in this paper. Secondly, tuple types are included because they make the subtyping problem harder (in fact, without tuples the subtyping problem for this system is fairly trivial).

2.4 Subtype Algorithm

Figure 2 employs an operation on types whose implementation is not immediately obvious — namely, determining whether one type subtypes another (i.e. $\mathtt{T}_1 \leq \mathtt{T}_2$). Indeed, there are many possible implementations of this operator and it is useful to consider two desirable properties:

Definition 2 (Subtype Soundness) *A subtype operator,* $\leq$, *is* sound *if, for any types* $\mathtt{T}_1$ *and* $\mathtt{T}_2$*, it holds that* $\mathtt{T}_1 \leq \mathtt{T}_2 \Longrightarrow [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$.

Definition 3 (Subtype Completeness) *A subtype operator,* $\leq$, *is* complete *if, for any types* $\mathtt{T}_1$ *and* $\mathtt{T}_2$*, it holds that* $[\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!] \Longrightarrow \mathtt{T}_1 \leq \mathtt{T}_2$.

A subtype operator which exhibits both of these properties is said to be *sound* and *complete*. We know of no previous flow typing system which has this property. The most notable existing system is

Subtyping (incomplete):

$$\frac{}{\mathtt{T} \leq \mathtt{any}}\ \text{[S-ANY1]} \qquad \frac{}{\mathtt{void} \leq \mathtt{T}}\ \text{[S-ANY2]}$$

$$\frac{}{\mathtt{int} \leq \neg(\mathtt{T_1}, \ldots, \mathtt{T_n})}\ \text{[S-INT1]} \qquad \frac{}{(\mathtt{T_1}, \ldots, \mathtt{T_n}) \leq \neg\mathtt{int}}\ \text{[S-INT2]}$$

$$\frac{\forall \mathtt{i}.\mathtt{T_i} \leq \mathtt{S_i}}{(\mathtt{T_1}, \ldots, \mathtt{T_n}) \leq (\mathtt{S_1}, \ldots, \mathtt{S_n})}\ \text{[S-TUP1]} \qquad \frac{\mathtt{n} \neq \mathtt{m} \vee \exists \mathtt{i}.\mathtt{T_i} \leq \neg\mathtt{S_i}}{(\mathtt{T_1}, \ldots, \mathtt{T_n}) \leq \neg(\mathtt{S_1}, \ldots, \mathtt{S_m})}\ \text{[S-TUP2]}$$

$$\frac{\forall \mathtt{i}.\mathtt{T_i} \geq \mathtt{S_i}}{\neg(\mathtt{T_1}, \ldots, \mathtt{T_n}) \leq \neg(\mathtt{S_1}, \ldots, \mathtt{S_n})}\ \text{[S-TUP3]}$$

$$\frac{\forall \mathtt{i}.\mathtt{T_i} \leq \mathtt{S}}{\mathtt{T_1} \vee \ldots \vee \mathtt{T_n} \leq \mathtt{S}}\ \text{[S-UNION1]} \qquad \frac{\exists \mathtt{i}.\mathtt{T} \leq \mathtt{S_i}}{\mathtt{T} \leq \mathtt{S_1} \vee \ldots \vee \mathtt{S_n}}\ \text{[S-UNION2]}$$

$$\frac{\exists \mathtt{i}.\mathtt{T_i} \leq \mathtt{S}}{\mathtt{T_1} \wedge \ldots \wedge \mathtt{T_n} \leq \mathtt{S}}\ \text{[S-INTERSECT1]} \qquad \frac{\forall \mathtt{i}.\mathtt{T} \leq \mathtt{S_i}}{\mathtt{T} \leq \mathtt{S_1} \wedge \ldots \wedge \mathtt{S_n}}\ \text{[S-INTERSECT2]}$$

Figure 3: A sound but *incomplete* subtyping algorithm for the language of types defined in §2.1.

that of Tobin-Hochstadt and Felleisen, who developed a flow type system for Racket (formerly PLT Scheme) [23, 26]. Like the system presented here, this supports subtyping in the presence of unions and negations and was shown to be sound. However, subtyping in their system is not complete, meaning that many potentially typeable programs cannot be typed.

Example. Figure 3 provides a typical set of rules defining a subtype algorithm over the language of types from §2.1, and is comparable to those of [23, 26]. Rules S-ANY1, S-ANY2, S-INT1, S-INT2 and S-TUP1 are straightforward. We illustrate the remainder by example. Under S-TUP2, for example, we have $(\mathtt{int}, \mathtt{int}) \leq \neg(\mathtt{int}, \mathtt{int}, \mathtt{int})$ and $((\mathtt{int}, \mathtt{int}), \mathtt{int}) \leq \neg(\mathtt{int}, \mathtt{int})$ whilst, similarly, we have $\neg(\mathtt{any}, \mathtt{any}) \leq \neg(\mathtt{int}, \mathtt{int})$ under S-TUP3. Under S-UNION1 we have e.g. $(\mathtt{int}, \mathtt{any}) \vee (\mathtt{any}, \mathtt{int}) \leq (\mathtt{any}, \mathtt{any})$, and e.g. $(\mathtt{int}, \mathtt{int}) \leq \mathtt{int} \vee (\mathtt{int}, \mathtt{any})$ under S-UNION2. Finally, under S-INTERSECT1 we have e.g. $\mathtt{int} \wedge (\mathtt{int}, \mathtt{int}) \leq \mathtt{int}$ and under S-INTERSECT2 e.g. $(\mathtt{int}, \mathtt{int}) \leq (\mathtt{any}, \mathtt{int}) \wedge (\mathtt{int}, \mathtt{any})$.

The rules of Figure 3 can be shown as sound with respect to our semantic interpretation of types (i.e. Definition 1). However, they are evidently not complete. For example, neither of the following can be shown under Figure 3 (but are implied by Definition 1):

$$\mathtt{any} \leq \mathtt{int} \vee \neg\mathtt{int}$$

$$(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{int}) \leq (\mathtt{int}, \mathtt{int}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{int})$$

The rules of Figure 3 can be further extended to handle specific cases (such as those above). For example, we could add the following rules:

$$\frac{}{\mathtt{any} \leq \mathtt{T} \vee \neg\mathtt{T}}\ \text{[S-ANY3]}$$

$$\frac{\mathtt{T} = (\mathtt{T_1}, \ldots, \mathtt{T_{k-1}}, \bullet, \mathtt{T_{k+1}}, \mathtt{T_n})}{\mathtt{T}[\bullet \mapsto \mathtt{S_1} \vee \ldots \vee \mathtt{S_n}] \leq \mathtt{T}[\bullet \mapsto \mathtt{S_1}] \vee \ldots \vee \mathtt{T}[\bullet \mapsto \mathtt{S_n}]}\ \text{[S-TUP4]}$$

S-ANY3 allows $\mathtt{any} \leq \mathtt{int} \vee \neg\mathtt{int}$ to be shown. Likewise S-TUP4 captures distributivity across tuples, allowing $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{int}) \leq (\mathtt{int}, \mathtt{int}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{int})$. However, adding additional rules seems (in our experience) somewhat futile and just forces ever-more esoteric counter-examples. For example, using the above rules we still cannot show that the following (which is implied by Definition 1) holds: $\texttt{any} \leq (\texttt{int}, \texttt{int}) \vee (\texttt{int}, \texttt{int}, \texttt{int}) \vee \big(\neg(\texttt{int}, \texttt{int}) \wedge \neg(\texttt{int}, \texttt{int}, \texttt{int})\big)$. In essence, the issue is that the number and variety of possible equivalences between types make it very difficult to construct a set of complete rules. To address this, our approach first normalises types to eliminate many such equivalences, and to make them more manageable.

2.5 Problem Statement

We can now succinctly express the problem addressed in this paper, namely: *to develop a sound and complete subtype algorithm for the language of types defined in* §*2.1.* We know of no previous algorithm with this property for a comparable language of types. In §4, we present such an algorithm which, in the worst case, requires an exponential number of steps to answer a subtyping query (in the size of the types involved). This complements the work of Frisch *et al.* who provided an existence proof but did not present a practical algorithm [33]. Furthermore, determining whether a polynomial time subtyping algorithm exists for this system remains, to the best of our knowledge, an open problem.

Positive Subtyping:

$$\frac{}{\mathtt{T}^{+} \leq \mathtt{T}^{+}} \qquad \text{(S-REFLEX)}$$

$$\frac{}{\mathtt{T}^{+} \leq \mathtt{any}} \qquad \text{(S-ANY)}$$

$$\frac{\forall \mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{n}\}. \mathtt{T}_{\mathtt{i}}^{+} \leq \mathtt{S}_{\mathtt{i}}^{+}}{(\mathtt{T}_{\mathtt{1}}^{+} \ldots, \mathtt{T}_{\mathtt{n}}^{+}) \leq (\mathtt{S}_{\mathtt{1}}^{+}, \ldots, \mathtt{S}_{\mathtt{n}}^{+})} \qquad \text{(S-TUP)}$$

Figure 4: Subtyping rules for positive atoms in FW.

3 Preliminaries

Before we present our algorithm for sound and complete subtyping over the language of types defined in §2.1, we first introduce the key concepts which underpin it. These then form the building blocks for our algorithmic developments in the following section.

3.1 Atoms

An important aspect of our algorithm is the definition of an *atom*. These are indivisible types which are split into the *positive* and *negative* atoms as follows:

Definition 4 (Type Atoms) *Let* $\mathtt{T}^{*}$ *denote a* type atom*, defined as follows:*

$$\begin{array}{l} \mathtt{T}^{*} ::= \mathtt{T}^{+} \mid \mathtt{T}^{-} \\ \mathtt{T}^{-} ::= \neg\mathtt{T}^{+} \\ \mathtt{T}^{+} ::= \mathtt{any} \mid \mathtt{int} \mid (\mathtt{T}_{\mathtt{1}}^{+}, \ldots, \mathtt{T}_{\mathtt{n}}^{+}) \end{array}$$

Here, $\mathtt{T}^{+}$ *denotes a* positive atom *whilst* $\mathtt{T}^{-}$ *denotes a* negative atom*.*

We can see from Definition 4 that a negative atom is simply a negated positive atom. Furthermore, the elements of tuple atoms are themselves positive atoms — which differs from the original definition of types, where an element could hold any possible type (including e.g. a union or intersection type). As we will see, one of the challenges we face lies in the process of converting from the general types of §2.1 into the more restricted forms used here. For example, $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ can be converted into $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$ — which is a union of positive atoms.

The first building block we require is that of subtyping between atoms. For our purposes, this operation need only be defined for positive atoms (a fact which at first surprised us), but could be extended to negative atoms as well. Figure 4 presents the subtyping relation between positive atoms. These employ judgements of the form "$\mathtt{T}_1 \leq \mathtt{T}_2$", which are read simply as: *the set of values described by* $\mathtt{T}_1$ *subtypes those of* $\mathtt{T}_2$. The rules of Figure 4 are mostly straightforward. Furthermore, we can trivially obtain soundness and completeness for the subtype relation given in Figure 4:

Lemma 1 *Let* $\mathtt{T}_1^{+}$ *and* $\mathtt{T}_2^{+}$ *be positive atoms. Then,* $\mathtt{T}_1^{+} \leq \mathtt{T}_2^{+} \iff [\![\mathtt{T}_1^{+}]\!] \subseteq [\![\mathtt{T}_2^{+}]\!]$.

Proof 1 *Straightforward by inspection of Definition 1 and Figure 4.* □

The second important building block is the observation that type atoms are *finitely indivisible*. That is, a type atom cannot be represented equivalently as a finite set of atoms which does not include itself:

Lemma 2 (Atom Indivisibility) *Let* $\mathtt{T}^{+}$ *be a positive type atom and* $\mathtt{S}_1^{+}, \ldots, \mathtt{S}_{\mathtt{n}}^{+}$ *a finite set of positive atoms where* $[\![\mathtt{T}^{+}]\!] = [\![\mathtt{S}_1^{+} \cup \ldots \cup \mathtt{S}_{\mathtt{n}}^{+}]\!]$*. Then, for some* $\mathtt{i}$*, we have* $\mathtt{T}^{+} = \mathtt{S}_{\mathtt{i}}^{+}$.

Proof 2 *Straightforward by inspection of Definitions 1 + 4.* □

The implications of Lemma 2 should not be overlooked. By construction, we have $\forall \mathtt{i}.[\![\mathtt{S}_\mathtt{i}^+]\!] \subseteq [\![\mathtt{T}^+]\!]$ and, hence, $\mathtt{T}^+$ is the unique canonical representative of the set it describes (i.e. $[\![\mathtt{T}^+]\!]$). Furthermore, given any $\mathtt{S}_1^+, \ldots, \mathtt{S}_\mathtt{n}^+$ where $[\![\mathtt{T}^+]\!] = [\![\mathtt{S}_1^+ \cup \ldots \cup \mathtt{S}_\mathtt{n}^+]\!]$, we can quickly and easily identify $\mathtt{T}^+$ using the subtype operator of Figure 4.

The third important building block we require is that of (positive) atom intersection. We let $\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+$ denote the construction of a type representing the intersection of the values in $\mathtt{T}_1^+$ with those of $\mathtt{T}_2^+$. Note that $\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+$ produces either a positive atom or `void` (in the case of no intersection):

Definition 5 (Atom Intersection) *Let $\mathtt{T}_1^+$ and $\mathtt{T}_2^+$ be positive atoms. Then, $\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+$ is a positive atom or* `void` *determined as follows:*

$$\begin{array}{lcll}
\mathtt{T}^+ \sqcap \mathtt{T}^+ & = & \mathtt{T}^+ & (1) \\
\mathtt{any} \sqcap \mathtt{T}^+ & = & \mathtt{T}^+ & (2) \\
\mathtt{T}^+ \sqcap \mathtt{any} & = & \mathtt{T}^+ & (3) \\
\mathtt{int} \sqcap (\mathtt{T}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+) & = & \mathtt{void} & (4) \\
(\mathtt{T}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+) \sqcap \mathtt{int} & = & \mathtt{void} & (5) \\
(\mathtt{T}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+) \sqcap (\mathtt{S}_1^+, \ldots, \mathtt{S}_\mathtt{m}^+) & = & \mathtt{void}, \textbf{if } \mathtt{n} \neq \mathtt{m} \textbf{ or } \exists \mathtt{i}.\mathtt{T}_\mathtt{i}^+ \sqcap \mathtt{S}_\mathtt{i}^+ = \mathtt{void} & (6) \\
 & = & (\mathtt{T}_1^+ \sqcap \mathtt{S}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+ \sqcap \mathtt{S}_\mathtt{n}^+), \textbf{otherwise} & (7)
\end{array}$$

Observe that (2) + (3) and (4) + (5) are symmetric.

Definition 5 is mostly straightforward. For example, $(\mathtt{int}) \sqcap (\mathtt{int}, \mathtt{int}) = \mathtt{void}$ as the number of fields differs (which follows Definition 1 where $[\![(\mathtt{int})]\!] \cap [\![(\mathtt{int}, \mathtt{int})]\!] = \emptyset$). Likewise, $(\mathtt{any}, \mathtt{any}) \sqcap (\mathtt{int}, \mathtt{int}) = (\mathtt{int}, \mathtt{int})$ as expected. Finally, we can trivially obtain soundness and completeness for this operation:

Lemma 3 *Let $\mathtt{T}_1^+$ and $\mathtt{T}_2^*$ be atoms. Then, $[\![\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+]\!] = [\![\mathtt{T}_1^+]\!] \cap [\![\mathtt{T}_2^+]\!]$.*

Proof 3 *Straightforward by inspection of Definition 1 and Definition 5.* □

3.2 Disjunctive Normal Form (DNF)

We now consider the procedure for converting a general type into a more classical Disjunctive Normal Form (DNF):

Definition 6 (DNF) *Let $\mathtt{T} \Longrightarrow^* \mathtt{T}'$ denote the application of zero or more rewrite rules (defined below) to type $\mathtt{T}$, producing a potentially updated type $\mathtt{T}'$.*

$$\begin{array}{lcll}
\neg\neg\mathtt{T} & \Longrightarrow & \mathtt{T} & (1) \\
\neg \bigvee_\mathtt{i} \mathtt{T}_\mathtt{i} & \Longrightarrow & \bigwedge_\mathtt{i} \neg\mathtt{T}_\mathtt{i} & (2) \\
\neg \bigwedge_\mathtt{i} \mathtt{T}_\mathtt{i} & \Longrightarrow & \bigvee_\mathtt{i} \neg\mathtt{T}_\mathtt{i} & (3) \\
(\bigvee_\mathtt{i} \mathtt{S}_\mathtt{i}) \wedge \bigwedge_\mathtt{j} \mathtt{T}_\mathtt{j} & \Longrightarrow & \bigvee_\mathtt{i} (\mathtt{S}_\mathtt{i} \wedge \bigwedge_\mathtt{j} \mathtt{T}_\mathtt{j}) & (4) \\
(\ldots, \bigvee_\mathtt{i} \mathtt{T}_\mathtt{i}, \ldots) & \Longrightarrow & \bigvee_\mathtt{i} (\ldots, \mathtt{T}_\mathtt{i}, \ldots) & (5) \\
(\ldots, \bigwedge_\mathtt{i} \mathtt{T}_\mathtt{i}, \ldots) & \Longrightarrow & \bigwedge_\mathtt{i} (\ldots, \mathtt{T}_\mathtt{i}, \ldots) & (6) \\
(\ldots, \neg\mathtt{T}, \ldots) & \Longrightarrow & (\ldots, \mathtt{any}, \ldots) \wedge \neg(\ldots, \mathtt{T}, \ldots) & (7)
\end{array}$$

$\mathtt{DNF}(\mathtt{T}) = \mathtt{T}'$ *denotes the computation $\mathtt{T} \Longrightarrow^* \mathtt{T}'$, such that no more rewrite rules apply.*

These DNF rewrite rules convert a type into something similar to classical disjunctive normal form. Rules 2 + 3 push negations inwards such that, for example, $\neg(\mathtt{T}_1 \wedge \mathtt{T}_2)$ rewrites to $\neg\mathtt{T}_1 \vee \neg\mathtt{T}_2$. Rule 4 factors unions out of intersections, such that e.g. $\mathtt{T}_1 \wedge (\mathtt{T}_2 \vee \mathtt{T}_3)$ rewrites to $(\mathtt{T}_1 \wedge \mathtt{T}_2) \vee (\mathtt{T}_1 \wedge \mathtt{T}_3)$. Recall from §2.1 that $\mathtt{T}_1 \wedge \mathtt{T}_2$ is indistinguishable from $\mathtt{T}_2 \wedge \mathtt{T}_1$ and, hence, rule 4 is not restricted to rewriting only a leftmost union (as the presentation might suggest). Rules 5, 6 + 7 are responsible for factoring union and intersection types out of tuples. For example, $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ rewrites by rule 4 to $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$. Similarly, $(\mathtt{any} \wedge \neg\mathtt{int}, \mathtt{any})$ rewrites by rule 6 and then by rule 7 to give $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any})$. Finally, we note that $\mathtt{DNF}(\mathtt{T})$ may produce an exponential number of terms in the worst-case [38–40].

Lemma 4 (DNF Construction) *Let* $\mathtt{T}$ *be a type where* $\mathtt{DNF(T)} = \mathtt{T'}$. *Then,* $\mathtt{T'}$ *has the form* $\bigvee_{\mathtt{i}} \bigwedge_{\mathtt{j}} \mathtt{T}^{*}_{\mathtt{i,j}}$.

Proof 4 *Straightforward by case analysis on the different ways the form* $\bigvee_{\mathtt{i}} \bigwedge_{\mathtt{j}} \mathtt{T}^{*}_{\mathtt{i,j}}$ *can be broken.* □

In considering Lemma 4, recall from Definition 4 that a type $\mathtt{T}^{*}$ represents a positive or negative atom. Thus, $\mathtt{T}^{*}$ is either a positive atom or a negated positive atom and may only recursively contain positive atoms. We now establish that the rewrite rules are guaranteed to produce equivalent types.

Lemma 5 (DNF Preservation) *Let* $\mathtt{T}$ *be a type where* $\mathtt{T} \Longrightarrow \mathtt{T'}$ *by a rewrite rule from Definition 6. Then,* $[\![\mathtt{T}]\!] = [\![\mathtt{T'}]\!]$.

Proof 5 *Straightforward by case analysis on the different rewrite rules.* □

Lemma 6 (DNF Termination) *Let* $\mathtt{T}$ *be a type. Then, there exists a type* $\mathtt{T'}$ *for which no further rewrite rules from Definition 6 apply, such that* $\mathtt{T} \Longrightarrow^{*} \mathtt{T'}$.

Proof 6 *Straightforward by analysing the movement of intersections, unions and negations. That is, all three are consistently moved out of tuples. Negations are consistently moved into unions and negations. Finally, unions are consistently moved out of intersections. Thus, each step lowers the overall entropy of the system.* □

4 Subtyping Algorithm

We now present our algorithm for sound and complete subtyping over the language of types defined in §2.1. We begin with an overview of the problem and our solution, and then proceed to progressively introduce the main pieces of the algorithm.

4.1 Overview

Let us reconsider the example subtyping algorithm presented in Figure 3. Recall that, whilst this algorithm can be shown sound, it is not complete. In particular, the following two rules are problematic:

$$\frac{\forall \mathtt{i}.\mathtt{T_i} \leq \mathtt{S}}{\mathtt{T_1} \vee \ldots \vee \mathtt{T_n} \leq \mathtt{S}} \text{ [S-UNION1]} \qquad \frac{\exists \mathtt{i}.\mathtt{T} \leq \mathtt{S_i}}{\mathtt{T} \leq \mathtt{S_1} \vee \ldots \vee \mathtt{S_n}} \text{ [S-UNION2]}$$

The problem is that examples of the form $\mathtt{T_1} \vee \mathtt{T_2} \leq \mathtt{T_3} \vee \mathtt{T_4}$ where $[\![\mathtt{T_1} \vee \mathtt{T_2}]\!] \subseteq [\![\mathtt{T_3} \vee \mathtt{T_4}]\!]$ exist, but where neither S-UNION1 nor S-UNION2 can apply (and, hence, such examples cannot be shown under Figure 3). The following illustrates two such examples:

$$\mathtt{int} \vee \neg\mathtt{int} \leq (\mathtt{int}, \mathtt{int}) \vee \neg(\mathtt{int}, \mathtt{int}) \qquad (1)$$

$$(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{int}) \leq (\mathtt{int}, \mathtt{int}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{int}) \qquad (2)$$

Another example is $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{int}) \wedge (\mathtt{int}, \mathtt{int} \vee (\mathtt{int}, \mathtt{int})) \leq (\mathtt{int}, \mathtt{int})$ which exploits a similar problem with the S-INTERSECT1 rule.

The problem common to all these examples seems to be the number and variety of equivalences between types. To tackle these problems, we build our algorithm around the intuition that $[\![\mathtt{T_1}]\!] \subseteq [\![\mathtt{T_2}]\!]$ iff $[\![\mathtt{T_1}]\!] - [\![\mathtt{T_2}]\!] = \emptyset$. This requires an algorithm for computing the difference of two types, such that $\mathtt{T_1} - \mathtt{T_2} = \mathtt{void}$ iff $[\![\mathtt{T_1}]\!] - [\![\mathtt{T_2}]\!] = \emptyset$. When types are represented as disjuncts of canonical conjuncts (referred to as *Canonicalised Disjunctive Normal Form* or DNF$^+$ for short), then computing their difference in a way that obtains the desired property is relatively easy. We proceed by first defining the notion of a *canonical conjunct* (§4.2) and then how one is constructed (§4.3). Finally, we show how a general type can be converted into DNF$^+$ (§4.4), and put the whole thing together illustrated with an example (§4.5).

4.2 Canonical Conjuncts

The first step in the canonicalisation process is to canonicalise intersections of the form $\mathtt{T_1} \wedge \ldots \wedge \mathtt{T_n}$. For example, $\mathtt{int} \wedge \mathtt{any}$ can be safely simplified to $\mathtt{int}$. Likewise, $(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{any})$ can be simplified to $\mathtt{void}$, while $(\mathtt{int}, \mathtt{int}) \wedge \neg\mathtt{int}$ can be simplified to $(\mathtt{int}, \mathtt{int})$ and, finally, $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any})$ can be simplified to $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$. As we will see, any intersection $\bigwedge_\mathtt{i} \mathtt{T_i^*}$ between atoms can be represented as a positive atom conjuncted with zero or more negative atoms, i.e. as $\mathtt{T_1^+} \wedge \neg\mathtt{T_2^+} \wedge, \ldots, \wedge\neg\mathtt{T_n^+}$.

Given the tools developed in §3.1 (i.e. Figure 4 and Definition 5), we can now formalise the notion of a *canonical conjunct* as follows:

Definition 7 (Canonical Conjunct) *Let* $\mathtt{T}^\wedge$ *denote a* canonical conjunct*. Then:*

1. $\mathtt{T}^\wedge$ *is a type of the form* $\mathtt{T_1^+} \wedge \neg\mathtt{T_2^+} \wedge, \ldots, \wedge\neg\mathtt{T_n^+}$*, and*
2. *for every negation* $\neg\mathtt{T_k^+}$*, we have* $\mathtt{T_1^+} \neq \mathtt{T_k^+}$ *and* $\mathtt{T_1^+} \geq \mathtt{T_k^+}$*, and*
3. *for any two distinct negations* $\neg\mathtt{T_k^+}$ *and* $\neg\mathtt{T_m^+}$*, we have* $\mathtt{T_k^+} \not\geq \mathtt{T_m^+}$*.*

Rule 1 from Definition 7 makes sense if we recall that $\mathtt{T_1} \wedge \neg\mathtt{T_2}$ can be thought of as $\mathtt{T_1} - \mathtt{T_2}$; thus, in rule 1 we require that the amount "subtracted" from the positive atom by any given negative atom is strictly less than the total. For example, $(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{any})$ is not permitted since this corresponds to the $\mathtt{void}$ (i.e. empty) type. Likewise, $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any})$ is not permitted either since this is more precisely represented as $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$. Rule 2 prohibits negative atoms from subsuming each other. For example, $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{int})$ is not permitted. However, we need not worry about atoms that overlap but where neither subsumes the other (i.e. where $[\![\mathtt{T}_1^+]\!] \cap [\![\mathtt{T}_2^+]\!] \neq \emptyset$ but $[\![\mathtt{T}_1^+]\!] \not\subseteq [\![\mathtt{T}_2^+]\!]$ and $[\![\mathtt{T}_1^+]\!] \not\supseteq [\![\mathtt{T}_2^+]\!]$). This follows from Lemma 2 (indivisibility) as such types canonically represent distinct sets (hence must be retained in the conjunct).

We can make the following strong statement about canonical conjuncts based on Definition 7 — namely, that canonical conjuncts are indeed canonical:

Lemma 7 (Canonical Conjuncts) *Let* $\mathtt{T}^\wedge = \mathtt{T}_1^+ \wedge \bigwedge_{\mathtt{i}} \neg\mathtt{T}_{\mathtt{i}}^+$ *and* $\mathtt{S}^\wedge = \mathtt{S}_1^+ \wedge \bigwedge_{\mathtt{j}} \neg\mathtt{S}_{\mathtt{j}}^+$ *be canonical conjuncts. Then, it follows that* $[\![\mathtt{T}^\wedge]\!] = [\![\mathtt{S}^\wedge]\!] \iff \mathtt{T}^\wedge = \mathtt{S}^\wedge$.

Proof 7 *Proof by contradiction. Assume* $\mathtt{T}^\wedge \neq \mathtt{S}^\wedge$. *There are three cases to consider:*

- $\mathtt{T}^+ = \mathtt{S}^+$. *Then, by Lemma 2, it follows that every* $\neg\mathtt{T}_{\mathtt{i}}^+$ *in* $\mathtt{T}^\wedge$ *must appear in* $\mathtt{S}^\wedge$ *and vice-versa (otherwise,* $[\![\mathtt{T}^\wedge]\!] = [\![\mathtt{S}^\wedge]\!]$ *cannot hold). Hence, a contradiction.*
- $\mathtt{T}^+ \neq \mathtt{S}^+$ *and* $\mathtt{T}^+ = \mathtt{any}$. *Conceptually, we must show no amounts could be subtracted from* $\mathtt{T}^+$ *and* $\mathtt{S}^+$ *to yield equivalent types. By construction* $\mathtt{any}$ *contains an infinite number of tuples (e.g.* $(\mathtt{T}_1^+, \mathtt{T}_2^+), (\mathtt{T}_1^+, \mathtt{T}_2^+, \mathtt{T}_3^+), \ldots$*), whilst* $\mathtt{T}^\wedge$ *and* $\mathtt{S}^\wedge$ *contain finitely many negations,* $\neg\mathtt{S}_{\mathtt{j}}^+$. *Therefore, contradiction as* $[\![\mathtt{T}^\wedge]\!] \neq [\![\mathtt{S}^\wedge]\!]$.
- $\mathtt{T}^+ \neq \mathtt{S}^+$ *and* $\mathtt{T}^+ \neq \mathtt{any}$ *and* $\mathtt{S}^+ \neq \mathtt{any}$. *We now generalise the argument from the previous case. By Definition 7, we have* $[\![\mathtt{T}^\wedge]\!] \neq \emptyset$ *and* $[\![\mathtt{S}^\wedge]\!] \neq \emptyset$ *and, hence, it follows that* $\mathtt{T}^+ \neq \mathtt{int}$ *and* $\mathtt{S}^+ \neq \mathtt{int}$ *(since* $[\![\mathtt{int}]\!] \cap [\![(\mathtt{U}_1^+, \ldots, \mathtt{U}_{\mathtt{n}}^+)]\!] = \emptyset$*). Thus, both* $\mathtt{T}^+ = (\mathtt{U}_1^+, \ldots, \mathtt{U}_{\mathtt{n}}^+)$ *and* $\mathtt{S}^+ = (\mathtt{V}_1^+, \ldots, \mathtt{V}_{\mathtt{m}}^+)$ *and (by a similar argument)* $\mathtt{n} = \mathtt{m}$. *By construction, some* $\mathtt{i}$ *exists where* $\mathtt{U}_{\mathtt{i}}^+ \neq \mathtt{V}_{\mathtt{i}}^+$. *Assume either* $\mathtt{T}_{\mathtt{i}}^+ = \mathtt{any}$ *or* $\mathtt{S}_{\mathtt{i}}^+ = \mathtt{any}$. *Then, a contradiction as, again, one of* $\mathtt{T}^\wedge$ *and* $\mathtt{S}^\wedge$ *describes infinitely many types that cannot be described by the other. Finally, if* $\mathtt{T}_{\mathtt{i}}^+ \neq \mathtt{any}$ *and* $\mathtt{S}_{\mathtt{i}}^+ \neq \mathtt{any}$, *then they have tuple type and we apply an inductive argument.*

□

4.3 Conjunct Construction

We now develop the mechanism for constructing a canonical conjunct from an arbitrary conjunct of atoms:

Definition 8 (Conjunct Canonicalisation) *Let* $\bigwedge_{\mathtt{i}} \mathtt{T}_{\mathtt{i}}^* \Longrightarrow^* \bigwedge_{\mathtt{j}} \mathtt{S}_{\mathtt{j}}^*$ *denote the application of zero or more rewrite rules (defined below) to* $\bigwedge_{\mathtt{i}} \mathtt{T}_{\mathtt{i}}^*$*, producing a potentially updated version* $\bigwedge_{\mathtt{j}} \mathtt{S}_{\mathtt{j}}^*$.

$$
\begin{array}{llllr}
\mathtt{void} \wedge \ldots & \Longrightarrow & \mathtt{void} & & (1) \\
\mathtt{T}_{\mathtt{i}}^+ \wedge \mathtt{T}_{\mathtt{j}}^+ \wedge \ldots & \Longrightarrow & (\mathtt{T}_{\mathtt{i}}^+ \sqcap \mathtt{T}_{\mathtt{j}}^+) \wedge \ldots & & (2) \\
\mathtt{T}_{\mathtt{x}}^+ \wedge \neg\mathtt{T}_{\mathtt{y}}^+ \wedge \ldots & \Longrightarrow & \mathtt{void} & \textbf{if } \mathtt{T}_{\mathtt{x}}^+ \leq \mathtt{T}_{\mathtt{y}}^+ & (3) \\
 & \Longrightarrow & \mathtt{T}_{\mathtt{x}}^+ \wedge \ldots & \textbf{if } \mathtt{T}_{\mathtt{x}}^+ \sqcap \mathtt{T}_{\mathtt{y}}^+ = \mathtt{void} & (4) \\
 & \Longrightarrow & \mathtt{T}_{\mathtt{x}}^+ \wedge \neg(\mathtt{T}_{\mathtt{x}}^+ \sqcap \mathtt{T}_{\mathtt{y}}^+) \wedge \ldots & \textbf{if } \mathtt{T}_{\mathtt{x}}^+ \not\geq \mathtt{T}_{\mathtt{y}}^+ & (5) \\
\neg\mathtt{T}_{\mathtt{x}}^+ \wedge \neg\mathtt{T}_{\mathtt{y}}^+ \wedge \ldots & \Longrightarrow & \neg\mathtt{T}_{\mathtt{x}}^+ \wedge \ldots & \textbf{if } \mathtt{T}_{\mathtt{x}}^+ \geq \mathtt{T}_{\mathtt{y}}^+ & (6)
\end{array}
$$

Let $\mathtt{CAN}(\bigwedge_{\mathtt{i}} \mathtt{T}_{\mathtt{i}}^*) = \bigwedge_{\mathtt{j}} \mathtt{S}_{\mathtt{j}}^*$ *denote the computation* $\bigwedge_{\mathtt{i}} \mathtt{T}_{\mathtt{i}}^* \Longrightarrow^* \bigwedge_{\mathtt{j}} \mathtt{S}_{\mathtt{j}}^*$*, such that no further rewrite rules apply.*

In considering the rules from Definition 8, we must recall that $\mathtt{T}_1 \wedge \mathtt{T}_2$ is not distinguishable from $\mathtt{T}_2 \wedge \mathtt{T}_1$. Therefore e.g. rule (2) picks two arbitrary positive atoms from $\bigwedge_{\mathtt{i}} \mathtt{T}_{\mathtt{i}}^*$, not just the leftmost two (as the presentation might suggest). Rule 1 reduces a conjunct containing void to void. Rule 2 simply combines all the positive atoms together using the intersection operator for positive atoms (Definition 5). After repeated applications of rule 2, there will be at most one positive atom remaining. Rule 3 catches the case when the negative contribution exceeds the positive contribution (e.g. $\mathtt{int} \wedge \neg\mathtt{any} \Longrightarrow \mathtt{void}$). Rule 4 catches negative components which lie outside the domain (e.g. $\mathtt{int} \wedge \neg(\mathtt{int}, \mathtt{int}) \Longrightarrow \mathtt{int}$). Rule 5 covers negative components which lie partially outside the domain and, hence, should be trimmed down (e.g. $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any}) \Longrightarrow (\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$). Finally, rule 6 catches the case where one negative component is completely consumed by another (e.g. $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \Longrightarrow (\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any})$).

Lemma 8 *Let $\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$ be an arbitrary conjunct of atoms containing at least one positive atom. Then, $\mathtt{CAN}(\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}})$ is either a canonical conjunct or* `void`*.*

Proof 8 *We proceed by a case analysis on the ways in which an arbitrary conjunct of atoms, $\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$, does not meet the requirements of Definition 7. Each corresponds to a case from Definition 7:*

1. *$\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$ does not contain exactly one positive atom (and, hence, by construction must contain more one than positive atom). Then, rule (2) from Definition 8 applies.*

2. *$\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$ contains a positive atom $\mathtt{T}^+_{\mathtt{i}}$ and a negative atom $\neg\mathtt{T}^+_{\mathtt{j}}$ where either $\mathtt{T}^+_{\mathtt{i}} = \mathtt{T}^+_{\mathtt{j}}$ or $\mathtt{T}^+_{\mathtt{i}} \not\geq \mathtt{T}^+_{\mathtt{j}}$. In such case, either rule (3), (4) or (5) from Definition 8 will apply.*

3. *$\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$ contains two negative atoms, $\neg\mathtt{T}^+_{\mathtt{i}}$ and $\neg\mathtt{T}^+_{\mathtt{j}}$, where $\mathtt{T}^+_{\mathtt{i}} \geq \mathtt{T}^+_{\mathtt{j}}$. In such case, rule (6) from Definition 8 applies.*

Finally, it remains to show that the rules of Definition 8 always reach a fixed point (i.e. that $\mathtt{CAN}(\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}})$ terminates). This is straight forward as, in all but one case, the rules strictly reduce the number of atoms in the conjunct. Rule (5) is the exception as it does not reduce the number of atoms. However, by inspection of Figure 4 and Definition 5, it is clear that $\mathtt{T}^+_{\mathtt{x}} \sqcap \mathtt{T}^+_{\mathtt{y}}$ yields a type strictly "smaller" than $\mathtt{T}^+_{\mathtt{y}}$ when $\mathtt{T}^+_{\mathtt{x}} \not\geq \mathtt{T}^+_{\mathtt{y}}$. □

The requirement in Lemma 8 for at least one positive atom arises because the rules of Definition 8 do not introduce positive atoms, but canonical conjuncts require them. In fact, we can easily ensure an arbitrary conjunct of atoms, $\bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$, has at least one positive atom — we simply add `any` to give $\mathtt{any} \wedge \bigwedge_{\mathtt{i}} \mathtt{T}^*_{\mathtt{i}}$. Finally, observe that we are not showing the rules are "correct" (i.e. semantically preserving) in Lemma 8 — only that the rules of Definition 8 produce a canonical conjunct. We will return shortly to show they are correct in this sense.

Definition 9 (Conjunct Intersection) *Let $\mathtt{T}^\wedge_1, \ldots, \mathtt{T}^\wedge_{\mathtt{n}}$ be canonical conjuncts. Then, $\mathtt{T}^\wedge_1 \sqcap \ldots \sqcap \mathtt{T}^\wedge_{\mathtt{n}}$ denotes their intersection, and is defined as $\mathtt{CAN}(\mathtt{T}^\wedge_1 \wedge \ldots \wedge \mathtt{T}^\wedge_{\mathtt{n}})$.*

Observe that, by construction, $\bigsqcap_{\mathtt{i}} \mathtt{T}^\wedge_{\mathtt{i}}$ yields either a canonical conjunct or `void`. Since a canonical conjunct cannot represent `void`, we have the required property that $[\![\bigwedge_{\mathtt{i}} \mathtt{T}^\wedge_{\mathtt{i}}]\!] = \emptyset \iff \bigsqcap_{\mathtt{i}} \mathtt{T}^\wedge_{\mathtt{i}} = \mathtt{void}$. To see why a canonical conjunct cannot represent `void`, recall that `void` is short-hand for $\neg\mathtt{any}$. Thus, we might consider $\mathtt{any} \wedge \neg\mathtt{any}$ to be a canonical conjunct representing `void` — but, this is invalid as, for a type $\mathtt{T}_1 \wedge \neg\mathtt{T}_2$ to be a canonical conjunct, Definition 7 requires $\mathtt{T}_1 > \mathtt{T}_2$. In fact, by construction, no canonical conjunct $\mathtt{T}^\wedge$ exists where $[\![\mathtt{T}^\wedge]\!] = [\![\mathtt{void}]\!]$. Finally, the following ensures the overall canonicalisation process is sound:

Lemma 9 *Let $\mathtt{T}^\wedge_1, \ldots, \mathtt{T}^\wedge_{\mathtt{n}}$ be canonical conjuncts. Then, $[\![\bigwedge_{\mathtt{i}} \mathtt{T}^\wedge_{\mathtt{i}}]\!] = [\![\, \bigsqcap_{\mathtt{i}} \mathtt{T}^\wedge_{\mathtt{i}} \,]\!]$.*

Proof 9 *We proceed by case analysis on the rules of Definition 8 and show that each rule preserves the described semantic set before and after the rewrite:*

1. *Straightforward since, for any* `T`*, we have $[\![\mathtt{void}]\!] \cap [\![\mathtt{T}]\!] = \emptyset$.*

2. *Follows immediately from Lemma 3.*

3. *Follows immediately from Definition 1 and Lemma 1.*

4. *Follows immediately from Definition 1 and Lemma 3.*

5. *Follows immediately from Definition 1 and Lemma 3.*

6. *Follows immediately from Definition 1 and Lemma 1.*

(Note, each case matches the corresponding rule from Definition 8) □

4.4 Canonicalised Disjunctive Normal Form (DNF$^+$)

Finally, we can now formally define the process for converting an arbitrary type (as defined in §2.1) into the variant of disjunctive normal form we refer to as *Canonicalised Disjunctive Normal Form* (DNF$^+$):

Definition 10 (DNF$^+$) *Let* $\texttt{T}^\vee$ *denote a type in* Canonicalised Disjunctive Normal Form (DNF$^+$). *Then, either* $\texttt{T}^\vee$ *has the form* $\bigvee_{\texttt{i}} \texttt{T}_{\texttt{i}}^\wedge$ *or is* $\texttt{void}$.

In our definition of DNF$^+$, we must include a special case for when $\texttt{T} = \texttt{void}$ since (as discussed earlier) $\texttt{void}$ is not a canonical conjunct. We can now easily construct types in DNF$^+$ as follows:

Definition 11 (DNF$^+$ Construction) *Let* $\texttt{T}$ *be a type where* $\texttt{T}' = \texttt{DNF}(\texttt{T})$ *and, hence, by Lemma 4 we have* $\texttt{T}' = \bigvee_{\texttt{i}} \bigwedge_{\texttt{j}} \texttt{T}^*_{\texttt{i,j}}$. *Then,* $\texttt{DNF}^+(\texttt{T}) = \bigvee_{\texttt{i}} \bigsqcap_{\texttt{j}} \texttt{T}^*_{\texttt{i,j}}$.

In considering Definition 11, we must recall our assumption from §2.1 that $\texttt{T}_1 \wedge \texttt{T}_1$ is indistinguishable from $\texttt{T}_1$. This is important as it ensures that, if all the intersected conjuncts give $\texttt{void}$, then the overall result is $\texttt{void}$ (i.e. since $\texttt{void} \vee \texttt{void} = \texttt{void}$, etc). This reflects our overall goal of ensuring $[\![\texttt{T}]\!] = \emptyset \iff \texttt{DNF}^+(\texttt{T}) = \texttt{void}$. We now present the overall theorem of this paper:

Theorem 1 *Let* $\texttt{T}$ *be a type (as defined in §2.1). Then,* $[\![\texttt{T}]\!] = [\![\texttt{DNF}^+(\texttt{T})]\!]$.

Proof 10 *Follows immediately from Lemma 9 and Definition 11.* □

4.5 Putting It All Together

We can now give a single subtyping rule which is sound and complete for the language of types defined in §2.1, and which entirely replaces Figure 3:

$$\frac{\texttt{DNF}^+(\texttt{T}_1 \wedge \neg \texttt{T}_2) = \texttt{void}}{\texttt{T}_1 \leq \texttt{T}_2}$$

The proof that this rule is sound and complete follows immediately from Theorem 1. Whilst it may seem odd to replace an entire system of rules (i.e. Figure 3) with a single rule, we must emphasise that this rule requires an exponential amount of time in the worst case. This is because the first step of the process which converts $\texttt{T}$ into disjunctive normal form (i.e. Definition 6) can produce an exponential explosion in the number of terms [38–40]. As such, determining whether a polynomial time subtyping algorithm exists remains an open problem.

We now return to consider a simple example from §2.4, and illustrate how it is resolved:

$$\begin{array}{lcl}
\texttt{any} \leq \texttt{int} \vee \neg\texttt{int} & = & \texttt{DNF}^+(\texttt{any} \wedge \neg(\texttt{int} \vee \neg\texttt{int})) \\
& = & \texttt{CAN}(\texttt{DNF}(\texttt{any} \wedge \neg(\texttt{int} \vee \neg\texttt{int}))) \\
& = & \texttt{CAN}(\texttt{any} \wedge \neg\texttt{int} \wedge \texttt{int}) \\
& = & \texttt{void}
\end{array}$$

Therefore, the algorithm correctly concludes that $\texttt{any} \leq \texttt{int} \vee \neg\texttt{int}$ holds.

5 Related Work

Tobin-Hochstadt and Felleisen consider the problem of typing previously untyped Racket (aka Scheme) programs using a flow-typing algorithm [23, 26]. Their system retypes variables within expressions dominated by type tests. However, they employ only union types and do not consider intersections or negations, making their system significantly more conservative than presented here. The very recent work of Dardha *et al.* employs a more expressive type system which includes negations, intersections and (implicitly) unions [41]. However, like the work of Frisch *et al.* [33], this does not detail an actual subtyping algorithm and, instead, elides the numerous technical details which are the subject of this paper. The work of Guha *et al.* focuses on flow-sensitive type checking for JavaScript [24]. This assumes programmer annotations are given for parameters, and operates in two phases: first, a flow analysis inserts special runtime checks; second, a standard (i.e. flow-insensitive) type checker operates on the modified AST. The system retypes variables as a result of runtime type tests, although only simple forms are permitted.

Since locals and stack locations are untyped in Java Bytecode, the Java Bytecode Verifier employs flow typing to ensure type safety [30]. The verifier retypes variables after assignments, but does not retype them after `instanceof` tests. And, instead of supporting explicit unions, it computes the least upper bound of the types for each variable at a meet point. A well-known problem, however, is that Java's subtype relation does not form a complete lattice [31]. This arises because two classes can share the same super-class and implement the same interfaces; thus, they may not have a unique least upper bound. The solution adopted by the bytecode verifier ignores interfaces entirely and, instead, maps them to `java.lang.Object`. This approach is conservative and means some programs will fail to verify that we might otherwise expect to pass. Several works on formalising the bytecode verifier have proposed the use of intersection types as an alternative solution [42, 43].

Type qualifiers constrain the possible values a variable may hold. CQual is a flow-sensitive qualifier inference supporting numerous type qualifiers, including those for synchronisation and file I/O [12]. CQual does not account for the effects of conditionals and, hence, retyping is impossible. Fähndrich and Leino discuss a system for checking non-null qualifiers in the context of C# [15]. Here, variables are annotated with `NonNull` to indicate they cannot hold **`null`**. Non-null qualifiers are interesting because they require variables be retyped after conditionals (i.e. retyping `v` from `Nullable` to `NonNull` after `v!=`**`null`**). Fähndrich and Leino hint at the use of retyping, but focus primarily on issues related to object constructors. Ekman *et al.* implemented this system within the JustAdd compiler, although few details are given regarding variable retyping [13]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `v!=`**`null`** checks [21]. The JACK tool for verifying `@NonNull` type annotations extends the bytecode verifier with an extra level of indirection called *type aliasing* [14]. This enables the system to retype a variable `x` as `@NonNull` in the body of an **`if`**`(x!=`**`null`**`)` conditional. The algorithm is formalised using a flow-sensitive type system operating on Java bytecode. JavaCOP provides an expressive language for writing type system extensions, including non-null types [22]. This system is flow-insensitive and cannot account for the effects of conditionals; as a work around, the tool allows assignment from a nullable variable `v` to a non-null variable if this is the first statement after a `v!=null` conditional.

Information Flow Analysis is the problem of tracking the flow of information, usually to restrict certain flows for security reasons. The work of Hunt and Sands is relevant here, since they adopt a flow-sensitive approach [17]. Their system is presented in the context of a simple While language not dissimilar to ours, although they do not account for the effect of conditionals. Russo *et al.* use an extended version of this system to compare dynamic and static approaches [18]. They demonstrate that a purely dynamic system will reject programs that are considered type-safe under the Hunt and Sands system. JFlow extends Java with statically checked flow annotations which are flow-insensitive [16].

Typestate Analysis focuses on flow-sensitive reasoning about the state of objects, normally to enforce temporal safety properties. Typestates are finite-state automatons which can encode usage rules for common APIs (e.g. a file is never read before being opened), and were pioneered by Strom and Yellin [44, 45]. Fink *et al.* present an interprocedural, flow-sensitive typestate verification system which is staged to reduce overhead [46]. Bodden *et al.* develop an interprocedural typestate analysis which is flow-sensitive at the intra-procedural level [47]. This is a hybrid system which attempts to eliminate all failure points statically, but uses dynamic checks when necessary. This was later extended to include a backward propagation step that improves precision [48].

Finally, we are aware of few works which attempt to extend the Java language with intersection types. The most relevant is that of Büchi and Weck who introduce *compound types* in to Java to overcome limitations caused by a lack of multiple inheritance [49]. Another interesting work is that of Igarashi Nagira, who introduce *union types* into Java [50]. These represent values which may be one *or* other of the possibilities; this differs from intersection types which represent values that are instances of *all* the possibilities.

6 Conclusion

Flow-typing systems often require complex type systems involving unions, intersections and/or negations. For example, unions are often used to describe the types of variables at meet points. Likewise, intersections and negations can describe the effect of runtime type tests. However, subtype testing is a challenging algorithmic problem for a type system containing these features. In particular, to ensure the greatest number of programs as possible can be typed, we desire that subtyping is both *sound* and *complete*. Frisch *et al.* demonstrated that this problem was decidable [33]. However, their proof was not constructive and did not lend itself naturally to an implementation. In this paper, we presented a sound and complete algorithm for subtyping in the presence of unions, intersections and negations. This contrasts with previous flow type systems (e.g. [23, 26]) which are shown sound, but not complete.

We framed our algorithm in the context of a flow typing system, which is a natural fit for this work and has many well-known practical applications. Furthermore, our motivation for developing this algorithm stems from our work on the Whiley programming language [28, 29, 32], which incorporates an ambitious flow type system. However, there are other potential applications for our algorithm, such as e.g. typing XML Schema [51, 52].

Acknowledgements. This work is supported by the Marsden Fund, administered by the Royal Society of New Zealand.

References

[1] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 278–292. ACM Press, 1991.

[2] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the Dynamic Languages Symposium (DLS)*, pages 53–64. ACM Press, 2007.

[3] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[4] Diomidis Spinellis. Java makes scripting languages irrelevant? *IEEE Software*, 22(3):70–71, 2005.

[5] Ronald Prescott Loui. In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41(7):22–26, 2008.

[6] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 117–136, 2009.

[7] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transations of the AMS*, 146:29–60, 1969.

[8] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[9] scala. The Scala programming language. URL `http://lamp.epfl.ch/scala/`. http://lamp.epfl.ch/scala/.

[10] G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 479–498, 2007.

[11] D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *TOPS*, 4(1):27–50, 1998.

[12] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2002.

[13] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.

[14] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proceedings of the confererence on Compiler Construction (CC)*, pages 229–244, 2008.

[15] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[16] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 228–241, 1999.

[17] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 79–90. ACM Press, 2006.

[18] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. CSF*, pages 186–199, 2010.

[19] David J. Pearce. JPure: a modular purity system for Java. In *Proceedings of the confererence on Compiler Construction (CC)*, volume 6601 of *LNCS*, pages 104–123, 2011.

[20] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 192–203. ACM Press, 1999.

[21] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the confererence on Compiler Construction (CC)*, 2001.

[22] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[23] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 117–128, 2010.

[24] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 256–275, 2011.

[25] Johnni Winther. Guarded type promotion: eliminating redundant casts in Java. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs*, pages 6:1–6:8. ACM Press, 2011.

[26] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 395–406, 2008.

[27] groovy. What's new in Groovy 2.0? URL `http://www.infoq.com/articles/new-groovy-20`. http://www.infoq.com/articles/new-groovy-20.

[28] The Whiley programming language, http://whiley.org.

[29] D. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. In *Proc. BYTECODE*, 2011.

[30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[31] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[32] D. J. Pearce. Whiley: a language with flow-typing and updateable value semantics. Technical Report ECSTR12-09, Victoria University of Wellington, 2012.

[33] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):19:1–19:64, 2008.

[34] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proc. FPCA*, pages 31–41. ACM Press, 1993.

[35] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. volume 789 of *LNCS*, pages 687–706. 1994.

[36] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the ICALP*, pages 198–199, 2005.

[37] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proceedings of the LICS*, pages 137–146. IEEE Computer Society Press, 2002.

[38] M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[39] Christopher Umans. The minimum equivalent DNF problem and shortest implicants. *JCSS: Journal of Computer and System Sciences*, 63, 2001.

[40] David Buchfuhrer and Christopher Umans. The complexity of boolean formula minimization. *J. Comput. Syst. Sci*, 77(1):142–153, 2011.

[41] Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. Technical report, Laboratoire PPS, Universit Paris Diderot, 2012.

[42] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. CCS*, pages 49–58, 1998.

[43] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 89–103, 1999.

[44] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[45] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE TSE*, 19(5):478–485, 1993.

[46] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2):1–9, 2008.

[47] Eric Bodden, Patrick Lam, and Laurie J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 36–47. ACM Press, 2008.

[48] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 5–14, 2010.

[49] Martin Büchi and Wolfgang Weck. Compound types for java. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 362–373, 1998.

[50] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1435–1441, 2006.

[51] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[52] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 51–63. ACM Press, 2003.