On Declarative Rewriting for Sound and Complete Union, Intersection and Negation Types

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
david.pearce@ecs.vuw.ac.nz

Abstract

Implementing the type system of a programming language is a critical task that is often done in an ad-hoc fashion. Whilst this makes it hard to ensure the system is sound, it also makes it difficult to extend as the language evolves. We are interested in describing type systems using rewrite rules from which an implementation can be automatically generated. Whilst not all type systems are easily expressed in this manner, those involving unions, intersections and negations are well-suited for this. For example, subtyping in such systems is naturally expressed as a maximal reduction over the intersection of types involved.

In this paper, we consider a relatively complex type system involving unions, intersections and negations developed previously in the context of type checking. This system was not developed with rewriting in mind, though clear parallels are immediately apparent from the original presentation. For example, the system presented required types be first converted into a variation on Disjunctive Normal Form. However, some aspects of the original system are more challenging to express with rewrite rules. We identify that the original system can, for the most part, be reworked to enable a natural expression using rewrite rules. We present an implementation of our rewrite rules in the Whiley Rewrite Language (WyRL), and report performance results compared with a hand-coded solution. We also present an implementation of our system in the Rascal rewriting system, and find different trade offs.

Keywords: Rewrite Systems, Type Systems, Type Theory.

1. Introduction

In the pursuit of efficient, flexible and correct programming language implementations language designers can benefit from a wide range of tools to aid their craft. *Parser generators* are perhaps the most well-known example and are routinely used to generate highly efficient parsers supporting complex forms of error-recovery [38, 81, 85]. *Meta-programming frameworks* simplify the development of tools which manipulate programs in some way, such as for program transformation [106, 30, 15]. Likewise, *IDE generator frameworks* provide a "one-click" approach to the generation of an IDE plugin from a given language definition [26, 45]. Other tools simplify the development of specific parts of a compiler, such as *name resolution* [62] or *type checking* [6, 99, 59]. Finally, *language workbenches* take this one step further by bringing

an array of such tools together coherently to reduce the costs of developing domain-specific languages [104, 70, 108, 44, 68]. This term was coined by Fowler to capture a growing trend in software development [50]:

> "*These tools take an old style of development - which I call language oriented programming and use IDE tooling in a bid to make language oriented programming a viable approach*"

Following on from this Kats and Visser captured the concept behind language workbenches more succinctly [70]:

> "*With a wealth of language construction tools, a need arose for comprehensive tools that integrated these different solutions and guided the development of languages*"

Whilst the need for workbenches is clear, the need to further develop individual construction tools remains. To that end, this paper considers techniques for generating a core component of a language's type system — namely, its *subtyping operator*.

General purpose term rewrite systems have been used in the engineering of programming languages and related tools for some time [82]. Success has been achieved in a wide range of areas, such as: *program transformation* [106, 19], *program analysis* [72], *formal verification* [36, 84], and *domain-specific languages* [105]. Numerous industrial-strength rewriting tools have also been developed, with notable examples including: CafeOBJ [35, 36], ELAN [14], Maude [27, 28], Stratego [106], ASF+SDF [105, 17] and Rascal [72] (amongst others). To that end, there is little doubt that certain problems can benefit significantly from being expressed with rewrite rules. Specifically, the separation of rewrite rules from the application *strategy* allows easy experimentation with different strategies and, furthermore, conceptually divides up the problem. For example, in the development of theorem proving tools, the benefits of separating rewrite rules from their application strategies are well-known [82]. Despite this, many widely-used automated theorem proving tools (e.g. [34, 9, 33, 20]) are still implemented in an ad-hoc fashion which, although typically efficient, makes them difficult to debug and maintain.

We are interested in tools which can simplify the development of programming language *type systems*. Indeed, it is well known that designing and implementing a type system from scratch is an arduous task. At the same time, this is one of the most critical components of a language's implementation as it enforces the properties upon which other components rely (e.g. type soundness) [59]. Of course, this can be done by encoding the system using an appropriate proof assistant (e.g. Isabelle/HOL [83], Coq [101], Agda [16], etc) — but this is far from trivial [69]. As an alternative, the lightweight approach of Roberson *et al.* employs model checking to check soundness of type systems [99]. Likewise, Veritas can automatically generate soundness proofs and efficient type checkers for (relatively simple) type systems from their descriptions [59, 56]. From the perspective of this paper, we are interested in applying declarative rewrite rules for describing and implementing type systems as this helps to separate type checking rules from their implementation. However, applying rewriting here is not straightforward as many type systems are not amenable to this approach. For example, type checking Java programs is more about traversing inheritance hierarchies and name resolution than anything else [74, 42].

The Whiley programming language has been developed to enable compile-time verification of programs and, furthermore, to make this accessible to everyday programmers [110, 92, 88, 93]. Of relevance here is the use of *flow typing* in Whiley (see

below) which necessitated *union*, *intersection* and *negation* types for type checking within the compiler [87, 69]. In short, we had stumbled upon a type system whose implementation is well-suited to declarative rewriting. The class of structural type systems involving *union*, *intersection* and *negation* types are of interest because of the challenge in developing a *sound* and *complete* subtype operator [2, 31, 22, 52, 37]. Here, soundness means the implementation cannot incorrectly say two things are subtypes when, in fact, they are not. Conversely, completeness means that the implementation cannot fail to report that two types are subtypes when, indeed, they are. Whilst it is easy enough to develop a system which is one or the other, achieving both is challenging. We should note also that having both does not automatically imply a system is "good" in some sense. From our perspective, we simply want the subtype operator to work in all situations where it could be reasonably expected to do so, and knowing it is complete gives us confidence. However, the trade-off is that the implementation is more complex.[1]

In previous work, we developed a sound and complete system and established its correctness [87]. On reflection, what was most striking about our approach was the obvious parallel with rewriting. In particular, it is critical in such a system that types are properly simplified in order to obtain the completeness property. Furthermore, subtyping in such systems corresponds to determining whether a specially-crafted intersection type can be reduced to `void` or not (more later).

1.1. Flow Typing

This paper explores an alternative implementation for a type system involving *union*, *intersection* and *negation* types. Such types are not common in mainstream programming languages. The question then maybe, *why we are interested in them?* The answer lies in our seemingly innocuous choice to use *flow typing* in Whiley. Flow-sensitive typing — or, flow typing — is a relatively new type checking discipline arising out of work on flow-sensitive static analysis. Hunt & Sands describe it thusly [66]:

> "*... we investigate flow-sensitive typings for a simple While language. We present a family of semantically sound security type systems ... which allow the type of a variable to* float*, assigning different security types at different points in the program.*"

Flow typing has recently been popularised by its inclusion in a number of up-and-coming programming languages, including Ceylon (Redhat) [24], Kotlin (JetBrains) [75], Flow (Facebook) [47], TypeScript (Microsoft) [11], Groovy [60], Racket [102] and, of course, Whiley [87]. Whiley employs a flow type system for flexibility and to reduce programmer burden [110, 87, 92, 93]. A defining characteristic of flow typing is the ability to *retype* variables after runtime type tests. This is achieved through the trifecta of union, intersection and negation types. Union types (e.g. $\mathtt{T}_1 \vee \mathtt{T}_2$) provide something similar to *algebraic data types* (ADTs) or *sum types* [8, 31, 67, 13]. Consider the following snippet in Whiley:

```
function indexOf(int[] xs,int x)->int|null:
   ...
```

[1] Care is also needed when handling *uninhabited types* which (for a complete subtype relation) are subtypes of everything. More specifically, we must ensure programmers cannot accidentally write such types. In Whiley, this is done by flagging an error for any type at the source level which is equivalent to **`void`** (i.e. is uninhabitable).

Here, `indexOf()` returns the first index of a given integer in the array, or **`null`** if there is none. The type **`int|null`** is a union type, meaning it is either an **`int`** *or* **`null`**. The system ensures the type **`int|null`** cannot be treated as an **`int`**, thus providing a neat solution to the problem of preventing `NullPointerException`s [41, 80, 25, 46].

Union types provide a platform upon which we can retype variables. However, *intersection types* (e.g. $\mathtt{T}_1 \wedge \mathtt{T}_2$) and *negation types* (e.g. $\neg\mathtt{T}_1$) provide the mechanism by which this happens. The following use of `indexOf()` illustrates:

```
int|null idx = indexOf(...)
if idx is int:
    ...             // idx has type int
else:
    ...             // idx has type null
```

Here, we must first ensure `idx` is an **`int`** before using it on the true branch. This is done using the type test operator **`is`** (similar to `instanceof` in Java). To determine the type of variable `idx` on the true branch, Whiley *intersects* its declared type (i.e. **`int|null`**) with the type test (i.e. **`int`**). Likewise, on the false branch, it computes the difference of these two types (i.e. **`int|null`** - **`int`**).[2] As an aside we note that, if the type of a variable on either branch of a type test becomes **`void`** (i.e. the empty type), then an error is reported.

1.2. Recursion

Although not the subject of this paper, Whiley additionally supports *recursive types*. These present some challenges for sound and complete *subtyping* and require, for example, checking *contractiveness* and *inhabitability* [69]. Indeed, they are beyond the techniques considered here (i.e. traditional rewriting). As such, we discuss them at various points to clarify the limits of our approach and motivate future work. The following illustrates the syntax for recursive types:

```
type Node is { any data, List next }
type List is null | Node
```

The type `{any data, List next}` indicates a record with two fields, `data` and `next`. Thus, a `List` is either **`null`** or a record with the given structure. A simple function operating over `List`s is given as follows:

```
function length(List list) -> int:
    if list is null:
        return 0
    else:
        return 1 + length(list.next)
```

This counts the number of nodes in a list. Here, we see flow typing in action as `list`, for example, is automatically retyped to `{any data, List next}` on the false branch [87, 86].

1.3. Contributions

In this paper, we develop a type checker using declarative rewrite rules for a type system developed previously [87]. This system provided a partial formalisation of the

[2] Observe type difference in such a system is given by: $\mathtt{T}_1 - \mathtt{T}_2 \equiv \mathtt{T}_1 \wedge \neg\mathtt{T}_2$

flow typing system used in Whiley but omits more complex features, such as *recursive* and *function* types. The key challenge is that this system was not developed with declarative rewrite rules in mind and requires adaptation for this purpose (see §2.3 for more). But, at the same time, it is an excellent candidate for the use of declarative rewrite rules because of the need to simplify union, intersection and negation types. Our system succinctly captures the original though, unfortunately, supports pairs not arbitrary tuples (see §6.1 for more). We evaluate the performance of our system compared with the existing ad-hoc implementation found in the Whiley compiler. Of interest is an efficient mechanism for producing a fixed set of types generated uniformly at random. We also note that although flow typing is our particular interest, the techniques developed in this paper can be applied to other related areas (e.g. typing XML documents [10, 65]).

Finally, an earlier version of this paper published at the conference on "Generative Programming: Concepts & Experience" [90]. The present paper contains: a more detailed exposition; an extended discussion on tuples and recursive types; an additional case study examining the use of the Rascal meta-programming system [15] as an alternative implementation; and, an extended related work section. In addition, it should be noted that the present paper does not establish a soundness or completeness result for the type system considered as this was established previously [87].

2. Background

We now set out the context of our problem. In particular, we examine in detail how a type system involving unions, intersections and negations could be implemented using declarative rewrite rules. The key is that the expressive nature of types in this system, along with the desire for certain properties (namely, *soundness* and *completeness*), make subtyping a challenging theoretical and practical problem.

Since we have already worked through the theory in detail elsewhere [87], our concern here lies with developing a simple implementation using appropriate language engineering tools (i.e. a declarative rewrite system, in this case).

2.1. Overview

We now examine a (slightly simplified) version of our type system involving union, intersection and negation types developed previously [87]. This is about the simplest possible type system involving these combinators which is practically useful. The system was presented in the context of a small flow-typed programming language called `FT`. For our purposes here, we are not concerned with other aspects of `FT`, such as its complete syntax, operational semantics, typing judgements, etc. Our interest lies in the language of types presented and, in particular, the subtyping operator defined over them. The following recaps the definition of types in `FT`:

$$
\begin{array}{rcll}
\mathtt{T} & ::= & \mathtt{any} \mid \mathtt{void} \mid \mathtt{int} & \textit{primitives} \\
& \mid & \mathtt{T}_1 \times \mathtt{T}_2 & \textit{pairs} \\
& \mid & \neg \mathtt{T} & \textit{negations} \\
& \mid & \mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n & \textit{intersections} \\
& \mid & \mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n & \textit{unions}
\end{array}
$$

Here, `any` represents the set of all values, `void` the empty set, `int` the set of all integers

and $\mathtt{T}_1 \times \mathtt{T}_2$ the set of all pairs.[3] The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. Amongst other things, union types are useful for characterising 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* $\mathtt{T}_2$, and is typically used in flow type systems to capture the type of a variable after a runtime type test. The type $\neg\mathtt{T}$ is the *negation* type containing those values *not* in $\mathtt{T}$. Negations are also useful for capturing the type of a variable on the false branch of a type test.

Finally, the original system made some common 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 indistinguishable from $\mathtt{T}_1$. These assumptions were not strictly necessary, but simplified the formal presentation. Furthermore, as we will see, they are naturally handled by our declarative rewrite system.

Semantics. To better understand the meaning of types, it is helpful to give them a *semantic interpretation* [2, 31, 22, 52]. This is a set-theoretic model where *subtype* corresponds to *subset*. Castagna and Frisch described it thusly [22]:

> "*... one starts with a model of the language and an interpretation of types as subsets of the model, then defines the subtyping relation as the inclusion of denoted sets, and, finally, when the relation is decidable, derives a subtyping algorithm from the semantic definition.*"

For the purposes of this paper, the *domain* $\mathbb{D}$ is the set of all values constructible from integers or pairs, inductively defined as follows:

$$\mathbb{D} = \mathbb{Z} \cup \left\{ (\mathtt{v}_1, \mathtt{v}_2) \mid \ \mathtt{v}_1 \in \mathbb{D}, \mathtt{v}_2 \in \mathbb{D} \right\}$$

Using the domain $\mathbb{D}$ we can define the interpretation for a given type $\mathtt{T}$, denoted by $[\![\mathtt{T}]\!]$, as follows:

Definition 1 (Semantic Interpretation). *Every type* $\mathtt{T}$ *is characterized by the set of values it contains, given by* $[\![\mathtt{T}]\!]$*:*

$$\begin{aligned}
[\![\mathtt{any}]\!] &= \mathbb{D} \\
[\![\mathtt{void}]\!] &= \emptyset \\
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![\mathtt{T}_1 \times \mathtt{T}_2]\!] &= \{(\mathtt{v}_1, \mathtt{v}_2) \mid \mathtt{v}_1 \in [\![\mathtt{T}_1]\!], \mathtt{v}_2 \in [\![\mathtt{T}_2]\!]\} \\
[\![\neg\mathtt{T}]\!] &= \mathbb{D} - [\![\mathtt{T}]\!] \\
[\![\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cap \ldots \cap [\![\mathtt{T}_n]\!] \\
[\![\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cup \ldots \cup [\![\mathtt{T}_n]\!]
\end{aligned}$$

Essentially, $[\![\mathtt{T}]\!]$ determines the set of values to which the type $\mathtt{T}$ corresponds. It is important to distinguish the *syntactic* representation $\mathtt{T}$ from its *semantic* interpretation $[\![\mathtt{T}]\!]$. The former corresponds (roughly speaking) to a physical machine representation, such as found in the source code of a programming language. In contrast, the latter corresponds to a mathematical ideal which, although unlikely to exist in any

[3] This is the only simplification over our original formulation which supported arbitrary tuples of the form $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ and is necessary due to technical limitations of the underlying rewrite system (more on this later).

practical implementation, provides a simple model we are aspiring to. As such, the syntactic representation diverges from the semantic model and, to compensate, we must establish a correlation between them. For example $\texttt{int}$ and $\neg\neg\texttt{int}$ have distinct syntactic representations, but are semantically indistinguishable. Similarly for $\big(\texttt{int} \vee (\texttt{int} \times \texttt{int})\big) \times \texttt{any}$ and $(\texttt{int} \times \texttt{any}) \vee \big((\texttt{int} \times \texttt{int}) \times \texttt{any}\big)$.

For completeness, we note the semantic interpretation of types given above can be defined in either an *inductive* or *coinductive* fashion [13]. To better understand this, consider solutions to the following equation:

$$[\![\texttt{T}]\!] = [\![\texttt{int} \times \texttt{T}]\!]$$

Under an inductive interpretation of types we are interested in the *least solution* to the above equation (which is $[\![\texttt{T}]\!] = \emptyset$ in this case). As such, the above type is said to be *uninhabited* under an inductive interpretation. In contrast, a coinductive interpretation corresponds to the *greatest solution* to the above equation (which, in this case, corresponds to the set of all lists of infinite length). For the purposes of this paper, we are interested in the inductive interpretation as this is the natural choice for describing Whiley.

Subtyping. Determining whether one type is a subtype of another (i.e. $\texttt{T}_1 \leq \texttt{T}_2$) is an operation whose implementation is not immediately obvious. There are two well-known properties which an implementation should have:

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

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

A subtype operator which exhibits both of these properties is said to be *sound* and *complete* [51, 22, 52]. For illustration, Figure 1 provides a typical set of subtyping rules used elsewhere [103, 102]. We illustrate them by example. Under rule S-TUP2, we have $\big((\texttt{int} \times \texttt{int}) \times \texttt{int}\big) \leq \neg(\texttt{int} \times \texttt{int})$ whilst $\neg(\texttt{any} \times \texttt{any}) \leq \neg(\texttt{int} \times \texttt{int})$ under S-TUP3. Under S-UNION1 we have $(\texttt{int} \times \texttt{any}) \vee (\texttt{any} \times \texttt{int}) \leq (\texttt{any} \times \texttt{any})$, and e.g. $(\texttt{int} \times \texttt{int}) \leq \texttt{int} \vee (\texttt{int} \times \texttt{any})$ under S-UNION2. Finally, under rule S-INTERSECT1, we have $\texttt{int} \wedge (\texttt{int} \times \texttt{int}) \leq \texttt{int}$ and under S-INTERSECT2 we have $(\texttt{int} \times \texttt{int}) \leq (\texttt{any} \times \texttt{int}) \wedge (\texttt{int} \times \texttt{any})$.

The rules of Figure 1 are *sound* with respect to Definition 1, but not *complete*. For example, consider this simple subtype test:

$$\texttt{any} \leq \texttt{int} \vee \neg\texttt{int} \tag{1}$$

This is clearly implied by Definition 1 as $\texttt{int} \vee \neg\texttt{int}$ contains the set of things which *are* integers, as well as those which *are not* integers. As such, it is equivalent to $\texttt{any}$ and the above should follow by reflexivity. And yet, unfortunately, it does not hold under the rules of Figure 1. The problem stems from S-UNION2 which breaks the subtype test into two smaller tests, of which neither holds:

$$\begin{array}{c} \texttt{any} \leq \texttt{int} \\ \bigvee \\ \texttt{any} \leq \neg\texttt{int} \end{array} \tag{2}$$

As a second example, consider the following subtype test:

$$\frac{}{\mathtt{T} \leq \mathtt{any}} \quad \frac{}{\mathtt{void} \leq \mathtt{T}} \qquad \text{(S-ANY, S-VOID)}$$

$$\frac{}{\mathtt{int} \leq \neg(\mathtt{T}_1 \times \mathtt{T}_2)} \qquad \text{(S-INT1)}$$

$$\frac{}{\mathtt{T}_1 \times \mathtt{T}_2 \leq \neg\mathtt{int}} \qquad \text{(S-INT2)}$$

$$\frac{\mathtt{T}_1 \leq \mathtt{S}_1 \quad \mathtt{T}_2 \leq \mathtt{S}_2}{\mathtt{T}_1 \times \mathtt{T}_2 \leq \mathtt{S}_1 \times \mathtt{S}_2} \qquad \text{(S-TUP1)}$$

$$\frac{\exists \mathtt{i}.(\mathtt{T}_\mathtt{i} \leq \neg\mathtt{S}_\mathtt{i})}{\mathtt{T}_1 \times \mathtt{T}_2 \leq \neg(\mathtt{S}_1 \times \mathtt{S}_2)} \qquad \text{(S-TUP2)}$$

$$\frac{\mathtt{S}_1 \leq \mathtt{T}_1 \quad \mathtt{S}_2 \leq \mathtt{T}_2}{\neg(\mathtt{T}_1 \times \mathtt{T}_2) \leq \neg(\mathtt{S}_1 \times \mathtt{S}_2)} \qquad \text{(S-TUP3)}$$

$$\frac{\forall \mathtt{i}.(\mathtt{T}_\mathtt{i} \leq \mathtt{S})}{\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_\mathtt{n} \leq \mathtt{S}} \qquad \text{(S-UNION1)}$$

$$\frac{\exists \mathtt{i}.(\mathtt{T} \leq \mathtt{S}_\mathtt{i})}{\mathtt{T} \leq \mathtt{S}_1 \vee \ldots \vee \mathtt{S}_\mathtt{n}} \qquad \text{(S-UNION2)}$$

$$\frac{\exists \mathtt{i}.(\mathtt{T}_\mathtt{i} \leq \mathtt{S})}{\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_\mathtt{n} \leq \mathtt{S}} \qquad \text{(S-INTERSECT1)}$$

$$\frac{\forall \mathtt{i}.(\mathtt{T} \leq \mathtt{S}_\mathtt{i})}{\mathtt{T} \leq \mathtt{S}_1 \wedge \ldots \wedge \mathtt{S}_\mathtt{n}} \qquad \text{(S-INTERSECT2)}$$

Figure 1: A sound but *incomplete* subtyping algorithm for the language of types defined in §2.1.

$$\big(\mathtt{int} \vee (\mathtt{int} \times \mathtt{int})\big) \times \mathtt{int} \leq (\mathtt{int} \times \mathtt{int}) \vee \big((\mathtt{int} \times \mathtt{int}) \times \mathtt{int}\big) \tag{3}$$

Again, this is clearly implied by Definition 1 as $\big(\mathtt{int} \vee (\mathtt{int} \times \mathtt{int})\big) \times \mathtt{int}$ includes all pairs of the form $\mathtt{int} \times \mathtt{int}$ as well as those of the form $(\mathtt{int} \times \mathtt{int}) \times \mathtt{int}$. But, again, the above does not hold under the rules of Figure 1. We start by applying S-UNION2 (as this is the only applicable rule) to break it down into two further tests:

$$\begin{array}{c} \big(\mathtt{int} \vee (\mathtt{int} \times \mathtt{int})\big) \times \mathtt{int} \leq (\mathtt{int} \times \mathtt{int}) \\ \bigvee \\ \big(\mathtt{int} \vee (\mathtt{int} \times \mathtt{int})\big) \times \mathtt{int} \leq \big((\mathtt{int} \times \mathtt{int}) \times \mathtt{int}\big) \end{array} \tag{4}$$

Immediately, we can see the problem as, again, clearly neither of the above hold. The conundrum then, is how to obtain a sound and complete system. Certainly, one can try to extend Figure 1 with new rules to catch new cases (such as for above). However, our experience suggests that this is ultimately futile and that an entirely different approach is needed. Indeed, this problem of finding a sound and complete subtype operator has been studied extensively [3, 2, 65, 22, 52, 87].

2.2. Subtyping as Rewriting

In prior work, we obtained a sound and complete subtyping operator by reducing the problem to determining whether a given type is equivalent to `void` [87]. Of relevance here is the mixture of hand-written rewrite rules and other computation used. However, we did not attempt to encode these rules using an existing rewrite engine. We now clarify how subtyping can be reduced to rewriting. The key observation is this well-known equivalence [52, 87]:

$$\mathtt{T}_1 \leq \mathtt{T}_2 \iff \mathtt{T}_1 \wedge \neg\mathtt{T}_2 = \mathtt{void} \tag{5}$$

To understand how we arrive at this equivalence it is helpful to break it down as follows:

$$\begin{array}{rcl} \mathtt{T}_1 \leq \mathtt{T}_2 & \iff & [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!] \\ & \iff & [\![\mathtt{T}_1]\!] - [\![\mathtt{T}_2]\!] = \emptyset \\ & \iff & \mathtt{T}_1 \wedge \neg\mathtt{T}_2 = \mathtt{void} \end{array}$$

This equivalence is critical as it means we can check subtyping by constructing the above intersection and attempting to rewrite it to `void`. Soundness and completeness of subtyping thus relies on soundness and completeness of rewriting. That is, every rewrite must preserve the semantics of types involved (soundness) and, for types equivalent to `void`, it must succeed in rewriting them to `void` (completeness).

Canonical Form. The original approach taken in our previous work relies on a number of steps. In particular, all types are rewritten into a variant of disjunctive normal form referred to as *Canonicalised Disjunctive Normal Form* which, essentially, is just a disjunction of *canonical conjuncts*. We now summarise the salient aspects of this representation, and the interested reader is referred to our earlier work for a more detailed exposition [87].

An important aspect of our approach was 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*:*

$$\begin{array}{l} \mathtt{T}^* ::= \mathtt{T}^+ \mid \mathtt{T}^- \\ \mathtt{T}^- ::= \neg\mathtt{T}^+ \\ \mathtt{T}^+ ::= \mathtt{any} \mid \mathtt{int} \mid \mathtt{T}_1^+ \times \mathtt{T}_2^+ \end{array}$$

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

From Definition 4 we see that a negative atom is simply a negated positive atom. Furthermore, the elements of pair atoms are 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).

One of the challenges lies in converting from the general types of §2.1 into a more restricted form made up from disjuncts and conjuncts of atoms. The first step is to convert a general type `T` into an equivalent of the following form:

$$\bigvee_{\mathtt{i}} \bigwedge_{\mathtt{j}} \mathtt{T}^*_{\mathtt{i},\mathtt{j}} \tag{6}$$

This was achieved in our original presentation using a procedure, `DNF(T)` defined as follows:

Definition 5 (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* $\mathtt{T}'$*.*

$$
\begin{array}{lll}
\neg\neg\mathtt{T} & \Longrightarrow \mathtt{T} & (1) \\
\neg\bigvee_{\mathtt{i}}\mathtt{T_i} & \Longrightarrow \bigwedge_{\mathtt{i}}\neg\mathtt{T_i} & (2) \\
\neg\bigwedge_{\mathtt{i}}\mathtt{T_i} & \Longrightarrow \bigvee_{\mathtt{i}}\neg\mathtt{T_i} & (3) \\
\big(\bigvee_{\mathtt{i}}\mathtt{S_i}\big)\wedge\bigwedge_{\mathtt{j}}\mathtt{T_j} & \Longrightarrow \bigvee_{\mathtt{i}}\big(\mathtt{S_i}\wedge\bigwedge_{\mathtt{j}}\mathtt{T_j}\big) & (4) \\
(\bigvee_{\mathtt{i}}\mathtt{T_i})\times\mathtt{T} & \Longrightarrow \bigvee_{\mathtt{i}}(\mathtt{T_i}\times\mathtt{T}) & (5a) \\
\mathtt{T}\times(\bigvee_{\mathtt{i}}\mathtt{T_i}) & \Longrightarrow \bigvee_{\mathtt{i}}(\mathtt{T}\times\mathtt{T_i}) & (5b) \\
(\bigwedge_{\mathtt{i}}\mathtt{T_i})\times\mathtt{T} & \Longrightarrow \bigwedge_{\mathtt{i}}(\mathtt{T_i}\times\mathtt{T}) & (6a) \\
\mathtt{T}\times(\bigwedge_{\mathtt{i}}\mathtt{T_i}) & \Longrightarrow \bigwedge_{\mathtt{i}}(\mathtt{T}\times\mathtt{T_i}) & (6b) \\
\neg\mathtt{T_1}\times\mathtt{T_2} & \Longrightarrow (\mathtt{any}\times\mathtt{T_2})\wedge\neg(\mathtt{T_1}\times\mathtt{T_2}) & (7a) \\
\mathtt{T_1}\times\neg\mathtt{T_2} & \Longrightarrow (\mathtt{T_1}\times\mathtt{any})\wedge\neg(\mathtt{T_1}\times\mathtt{T_2}) & (7b)
\end{array}
$$

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

Rules $(1-3)$ above are fairly straightforward and simply move a term into *Negation Normal Form (NNF)* where negations are eliminated or pushed *inwards* as much as possible. However, rules $(5a-6b)$ are somewhat non-standard and pull unions, intersections and negations *outside* of pairs. For example, type $\big(\mathtt{int}\vee(\mathtt{int}\times\mathtt{int})\big)\times\mathtt{any}$ is rewritten into a union of positive atoms $(\mathtt{int}\times\mathtt{any})\vee\big((\mathtt{int}\times\mathtt{int})\times\mathtt{any}\big)$.

The next step in the process is to simplify conjuncts into *canonical conjuncts*, which were defined as follows:

Definition 6 (Canonical Conjunct). *A* canonical conjunct*, denoted* $\mathtt{T}^{\wedge}$*, has the form* $\mathtt{T}_1^+\wedge\neg\mathtt{T}_2^+\wedge,\ldots,\wedge\neg\mathtt{T}_n^+$ *where:*

1. *For each negation* $\neg\mathtt{T}_k^+$*, we have* $\mathtt{T}_1^+\neq\mathtt{T}_k^+$ *and* $\mathtt{T}_1^+\geq\mathtt{T}_k^+$*.*
2. *For 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 6 makes sense if we consider $\mathtt{T_1}\wedge\neg\mathtt{T_2}$ as $\mathtt{T_1}-\mathtt{T_2}$; thus, the amount "subtracted" from the positive atom by any given negative atom is strictly less than the total. For example, $(\mathtt{int}\times\mathtt{int})\wedge\neg(\mathtt{any}\times\mathtt{any})$ is not permitted since this corresponds to `void`. Likewise, $(\mathtt{any}\times\mathtt{int})\wedge\neg(\mathtt{int}\times\mathtt{any})$ is not permitted as it is more precisely represented as $(\mathtt{any}\times\mathtt{int})\wedge\neg(\mathtt{int}\times\mathtt{int})$. Rule 2 prohibits negative atoms from subsuming each other, such as in $(\mathtt{any}\times\mathtt{any})\wedge\neg(\mathtt{int}\times\mathtt{int})\wedge\neg(\mathtt{any}\times\mathtt{int})$.

Canonical Construction. Our original presentation included a mechanism for constructing canonical conjuncts from an arbitrary conjunct of atoms. This mechanism required the conjunct to be repeatedly reduced using a set of rewrites until no further reductions were possible.

The rewrite rules from the original presentation are shown in Figure 2. We now consider the effect each of these rules has when reducing a given type. Rule 1 reduces a conjunct containing `void` to `void`. Rule 2 simply combines all the positive atoms together using a special *intersection operator* for positive atoms (more later, but for now view as intersection). Observe that, after repeated applications of Rule 2, *there is at most one positive atom remaining*. Rule 3 catches the case when the negative contribution exceeds the positive contribution for a given conjunct (e.g. $\mathtt{int}\wedge\neg\mathtt{any}\Longrightarrow\mathtt{void}$). Rule 4 catches negative components which lie entirely outside the domain of the positive atom (e.g. $\mathtt{int}\wedge\neg(\mathtt{int}\times\mathtt{int})\Longrightarrow\mathtt{int}$) as such components can be safely dropped. Rule 5 covers negative components needing to be trimmed (e.g. $(\mathtt{any}\times\mathtt{int})\wedge\neg(\mathtt{int}\times\mathtt{any})\Longrightarrow(\mathtt{any}\times\mathtt{int})\wedge\neg(\mathtt{int}\times\mathtt{int})$). Finally, Rule 6 catches negative components which are made redundant by others. For example, $\ldots\wedge\neg(\mathtt{int}\times\mathtt{any})\wedge\neg(\mathtt{int}\times\mathtt{int})\Longrightarrow\ldots\wedge\neg(\mathtt{int}\times\mathtt{any})$).

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

Figure 2: Mechanism for constructing canonical conjuncts from arbitrary conjuncts of the form $\bigwedge_{\mathtt{i}} \mathtt{T}_{\mathtt{i}}^{*}$. Recall that $\mathtt{T}_1 \wedge \mathtt{T}_2$ is indistinguishable 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).

2.3. Problem Statement

A key observation from Definition 6 is that a canonical conjunct cannot represent `void`. The guarantee we previously obtained is that, after maximally reducing a type, we have *either* a union of canonical conjuncts *or* `void`. This gives the mechanism needed for sound and complete subtyping. Our goal in this paper is to provide an encoding of the system within an existing rewrite engine. Such an encoding improves upon the ad-hoc hand-written rewrite rules given in our original presentation in several ways. Firstly, they provide a more precise description. Secondly, they allow the system to be mechanically tested to increase confidence that the original hand-written proofs were correct. Such an encoding also allows automatic generation of a type checker, thereby closing the gap between the type system design (i.e. our original presentation [87]) and its implementation (i.e. in the Whiley compiler). Indeed, the current implementation in the compiler bears little resemblance to our presentation.

From our perspective, the *subtype operator* and *intersection operator* from Figure 2 present a challenge. The latter represents a computation which calculates the intersection of two positive atoms as a function (i.e. rather than with rewrites). Both operators are defined for positive atoms and help (respectively) ensure that: canonical conjuncts are indeed canonical and not equivalent to `void`; and, that rewriting terminates. The latter is important since, without the subtype constraint, Rule (5) from Figure 2 would not necessarily converge. That is, it could be applied repeatedly to the same term without actually reducing it. In contrast, all of the other rules are *trivially convergent* — that is, repeated applications of those rules must eventually terminate.

The subtype and intersection operators are problematic because they do not immediately correspond to declarative rewrite rules and, as such, cannot be expressed as is in our chosen rewrite language (more on this later). However, there are several key observations which we can exploit here:

- *Intersection Operator*. The intersection operator for positive atoms is a relatively straightforward procedure. As such, it can indeed be implemented using rewrite rules. The purpose of having it in Figure 2 is purely to indicate that this calculation has begun as otherwise, for example, the left- and right-hand sides of Rule (2) would be the same. For further discussion, see §4.3.

- *Subtype Operator.* Consider an arbitrary type $\mathtt{T}_1^{+} \wedge \neg\mathtt{T}_2^{+}$ which would be processed by rules (3-5) from Figure 2. The challenge lies in determining which rule to apply. Suppose we immediately transform it into $\mathtt{T}_1^{+} \wedge \neg(\mathtt{T}_1^{+} \sqcap \mathtt{T}_2^{+})$ which produced $\mathtt{T}_1^{+} \wedge \neg\mathtt{T}_3^{+}$ where $\mathtt{T}_1^{+} \geq \mathtt{T}_3^{+}$. This transformation is evidently safe (though not necessarily convergent). More importantly, Rule (3) would then apply *iff*

$\mathtt{T}_1^+ \equiv \mathtt{T}_3^+$. In fact, a useful property of positive atoms is that they are *unique* (i.e. $\mathtt{T}_1^+ \equiv \mathtt{T}_3^+ \iff \mathtt{T}_1^+ = \mathtt{T}_3^+$). Thus, with care, subtyping can be reduced here to an equality test and, hence, easily implemented in our rewrite language (i.e. because this is a built-in operator). For further discussion, see §4.4.

- *Canonical Not Required.* Rule 2 from Definition 6 is an interesting case. Its purpose is simply to ensure that canonical conjuncts *really are canonical*. For example, $\mathtt{any} \wedge \neg \mathtt{int}$ and $\mathtt{any} \wedge \neg \mathtt{int} \wedge \neg \mathtt{int}$ are equivalent, but the latter is not a canonical conjunct because of Rule 2. This rule was useful in the original presentation to aid the formalisation. However, from a practical perspective, *we do not require that canonical conjuncts are actually canonical!* Rather, we require only that they are not equivalent to `void`.

Finally, we note the issue of confluence and termination. These properties were implied by the soundness and completeness theorem obtained previously for the original system. However, as the system developed here differs somewhat from the original, we provide additional discussion of this as necessary.

3. Towards an Encoding

We now turn our attention to the encoding of types from our type system using declarative rewrite rules. At this stage, our goal is to introduce WyRL and give some indication as to how types can be encoded, rather than develop a complete solution. Our approach is framed in the context of the *Whiley Rewrite Language (WyRL)*, which is a standalone tool providing a domain-specific declarative rewrite language and code generator [89]. We choose this tool because it is familiar to us and is already part of the Whiley compiler. Furthermore, whilst originally developed for experimenting with different approaches to automated theorem proving, it turned out to be suitable for subtyping as well. Using this tool also makes it possible to perform a meaningful comparison with the existing (ad-hoc) subtype implementation in the compiler. We note, however, that the general approach taken here should be applicable to other similar tools (e.g. Spoofax [109, 70], etc). Indeed, we present a Rascal [72] implementation of our system later in §7 which is similar but more complete (i.e. can support arbitrary tuples rather than just pairs). Unfortunately, performing a meaningful performance comparison with this is not possible.

At this stage, we are not yet attempting to develop a complete encoding (in some sense). In the following section, we consider the more challenging problem of encoding the sound and complete subtype operator discussed in §2.2.

3.1. Basics

We begin with a cut-down encoding of types which provides a gradual introduction to the syntax. The fundamental building blocks in WyRL are *terms*. For example, consider the following (very simple) language of types:

```
term Any
term Void
define Primitive as Any | Void          // Primitive Types

term Not(Type)                          // Negation Types

define Type as Primitive | Not(Type)
```

This defines terms of the form `Any`, `Void`, `Not(Any)`, `Not(Not(Void))`, etc. Here, the `Not` term describes recursive structures of arbitrary depth. Rewrite rules can be defined over terms of this language as follows:

```
reduce Not(Primitive b):
    => Void, if b == Any
    => Any

reduce Not(Not(Type t)):
    => t
```

These implement two simplifications for terms in our language. Each rewrite rule consists of a pattern and one or more *cases* delineated by "`=>`". The first rule matches either `Not(Void)` or `Not(Any)` and employs a *conditional case* to distinguish them. Cases are tried in order of occurrence and, hence, the second case is applied only if the first is not. The second rule matches terms such as `Not(Not(Any))`, `Not(Not(Not(Void)))`, etc. The order of applications is unspecified and there are two valid applications of this rule to `Not(Not(Not(Void)))` (i.e. where `t` either binds to `Void` or `Not(Void)`) and we cannot determine which will be applied. In this case it doesn't matter which is applied first and, in general, *it is the programmer's responsibility to ensure their rewrites are confluent*.

3.2. Union and Intersection Types

We now encode unions and intersections using the pattern matching features of WyRL. In the rewrites above, the patterns were simple and the ordering of subterms not a consideration. However, intersection and union types may have an arbitrary number of subterms. WyRL supports *unordered* (i.e. associative-commutative) collections which correspond to sets and, hence, are suitable here. Furthermore, pattern matching over unordered collections is itself unordered.

The following illustrates a compound term with an unordered set of subterms:

```
term And{Type...}   // Intersection Type
```

Here, the "`...`" in "`Type...`" indicates *zero-or-more* occurrences. Since sets are unordered, `And{Any,Void}` is indistinguishable from `And{Void,Any}` (recall this matches the assumptions made in §2.1). Rewrites over sets employ unordered (i.e. associative-commutative) pattern matching:

```
reduce And{Primitive p, Type... ts}:
    => Void, if p == Void
    => Any, if |ts| == 0
    => And (ts)
```

The above matches any instance of `And` with *at least one subterm which is an instance of* `Primitive`. Thus, it will match `And{Any}` and `And{Not(Void),Any}`, but will not match `And{}` or `And{Not(Void)}`. Furthermore, there are two possible applications of this rule to the term `And{Any,Void}` and the order in which they will be applied is unspecified.

3.3. Pair Types

Whilst union and intersection types represent unordered collections of types, pairs represent ordered sequences. Fortunately, WyRL supports *ordered* collections of subterms which correspond roughly to lists or arrays and provide *ordered* pattern matching. The following illustrates:

```
term Pair[Type,Type]

reduce And{Pair[Type x1, Type x2],
           Pair[Type y1, Type y2],
           Type... rest}:
 => let x1y1 = And{x1,y1}
    let x2y2 = And{x2,y2}
    in And(Pair[x1y1,x2y2] ++ rest)
```

This rule reduces intersection of pair types to that of intersecting element types. Note that the "++" operator is overloaded for both *list consing* (as used here) and *list concatenation*.

4. Implementation

In the previous section, we now explored various aspects of WyRL and illustrated, roughly speaking, how a type system involving unions, intersections and negations can be represented. We now present our completed implementation of the type system outlined in §2.1 using declarative rewrite rules. The main purpose of these rules is to simplify a given type as much as possible and, using this, we trivially obtain the subtyping operator. This provides a useful contrast with the implementation of subtyping currently used in the Whiley compiler, which does not exploit rewriting and is an ad-hoc implementation. Our goal is to eventually replace this with an implementation based on rewriting, though there are still hurdles to overcome here. In particular, adding support for *recursive types* remains a significant hurdle (see §6.3 for more on this).

From our perspective, the benefits of rewriting are easier maintenance and extension. Indeed, maintaining our ad-hoc type system implementation over the years has proved challenging. This is evident from the large number of issues and bugs raised in relation to the implementation of subtyping (e.g. [111], issues `#840`, `#838`, `#829`, `#802`, `#783`, `#764`, `#696`, `#678`, `#645`, `#619`, `#585`, `#583`, `#561`, `#339`, `#50`). Whilst such issues are by no means impossible to fix, they often require extensive refactoring and can represent weeks or months of work each. Likewise, new features being added to the language have proved challenging when they involve the type system. For example, the recent addition of *reference lifetimes* required careful modifications to the type system [100]. In particular, the subtype relation had to be extended to support function types with lifetime arguments.

Whilst our ad-hoc subtyping implementation has proved unwieldy, it is reliable and efficient. In replacing it with an implementation based on rewriting, we must retain some efficiency as the type system is invoked many times during the compilation of even small programs. That said, some reduction in performance would be considered acceptable.

4.1. Preliminaries

We now examine each of the main pieces of our solution in turn. An important aspect here is the need to ensure termination of rewriting. In many cases, this is straightforward. However, in some cases, more careful reasoning is required. First, the syntax of types is as follows:

```
term Void
term Any
term Int

define Primitive as Any | Void | Int

term Pair[Type,Type]     // Pair Types
term Not(Type)           // Negation Types
term Or{Type...}         // Union Types
term And{Type...}        // Intersection Types

define Type as Primitive | Pair | Not | Or | And
```

These are largely as indicated from §3. Some basic rewrites over these types are given in Figure 3. The latter two are for flattening nested union and nested intersection types. Perhaps surprisingly, these do not correspond to any rules from the original presentation where, instead, a tacit (and unwritten) assumption was made that unions of unions were automatically flattened and, likewise, for intersections of intersections.

Finally, we note the above includes a term for `void`, despite this being equivalent to ¬`any`. This is convenient later where we need to identify ¬`void` as an (almost) negative atom (see §4.5). However, expressing this as ¬¬`any` triggers a reduction to `any` which is no longer a negative atom.

4.2. DNF Construction

As discussed already, an important first step when rewriting a general term is to move it into a type of the following form:

$$\bigvee_{i} \bigwedge_{j} \mathtt{T}^{*}_{i,j} \tag{7}$$

Figure 4 presents the rewrite rules corresponding to Definition 5. Again, these are mostly straightforward and employ set comprehensions to manage terms of arbitrary size.

```
reduce Not(Any):
    => Void

reduce Or{Void}:
    => Void

reduce Or{Or{Type... t1s}, Type... t2s}:
    => Or(t1s ++ t2s)

reduce And{And{Type... t1s}, Type... t2s}:
    => And(t1s ++ t2s)
```

Figure 3: Some simple rewrites applied to basic types.

```
reduce Not(Not(Type t)):                                    // Def 5(1)
  => t

reduce Not(Or{Type... ts}):                                 // Def 5(2)
  => let xs = { Not(t) | t in ts }
     in And(xs)

reduce Not(And{Type... ts}):                                // Def 5(3)
  => let xs = { Not(t) | t in ts }
     in Or(xs)

reduce And{Or{Type... xs}, Type... ys}:
  => let ys = { And(x ++ ys) | x in xs }
     in Or(ys)                                              // Def 5(4)

reduce Pair[Or{Type... xs},Type t]:
  => let ys = { Pair[r,t] | r in xs }
     in Or(ys)                                              // Def 5(5a)

reduce Pair[Type t, Or{Type... xs}]:
  => let ys = { Pair[t,r] | r in xs }
     in Or(ys)                                              // Def 5(5b)

reduce Pair[And{Type... xs},Type t]:
  => let ys = { Pair[r,t] | r in xs }
     in And(ys)                                             // Def 5(6a)

reduce Pair[Type t, And{Type... xs}]:
  => let ys = { Pair[t,r] | r in xs }
     in And(ys)                                             // Def 5(6b)

reduce Pair[Not(Type t1),Type t2]:
  => let lhs=Pair[Any,t2], rhs = Not(Pair[t1,t2])
     in And{lhs,rhs}                                        // Def 5(7a)

reduce Pair[Type t1, Not(Type t2)]:
  => let lhs = Pair[t1,Any], rhs = Not(Pair[t1,t2])
     in And{lhs,rhs}                                        // Def 5(7b)
```

Figure 4: Rewrite rules roughly equivalent to Definition 5.

4.3. Intersecting Positive Atoms

We now begin the more challenging process of encoding Figure 2. As highlighted already, this employs an intersection operator over positive atoms which requires special attention. This operator was defined as follows:

Definition 7 (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:*

$$
\begin{array}{lcll}
\mathtt{T}^+ \sqcap \mathtt{T}^+ & = & \mathtt{T}^+ & (1) \\
\mathtt{any} \sqcap \mathtt{T}^+ & = & \mathtt{T}^+ & (2) \\
\mathtt{T}^+ \sqcap \mathtt{any} & = & \mathtt{T}^+ & (3) \\
\mathtt{int} \sqcap (\mathtt{T}_1^+ \times \mathtt{T}_2^+) & = & \mathtt{void} & (4) \\
(\mathtt{T}_1^+ \times \mathtt{T}_2^+) \sqcap \mathtt{int} & = & \mathtt{void} & (5) \\
(\mathtt{T}_1^+ \times \mathtt{T}_2^+) \sqcap (\mathtt{S}_1^+ \times \mathtt{S}_2^+) & = & \mathtt{void}, \textbf{if } \exists_\mathtt{i}.(\mathtt{T}_\mathtt{i}^+ \sqcap \mathtt{S}_\mathtt{i}^+) = \mathtt{void} & (6) \\
 & = & (\mathtt{T}_1^+ \sqcap \mathtt{S}_1^+) \times (\mathtt{T}_2^+ \sqcap \mathtt{S}_2^+), \textbf{else} & (7)
\end{array}
$$

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

Although presented in the style of a function, this operator provides a natural recursive decomposition of terms which is suitable for rewriting. We begin with some additional terms:

```
define NegAtom as Not(PosAtom)
define PosAtom as Int | Any | Pair[PosAtom,PosAtom]

term Intersect[PosAtom,PosAtom]
```

Here, `PosAtom` and `NegAtom` refine the concept of a `Type` and, hence, any instance of `PosAtom` is also an instance of `Type` (though not necessarily vice-versa). The term `Intersect` has been introduced to represent the computation of the intersection operator. That is, we should consider an instance of `Intersect` to represent the computation in progress. Once the computation is complete, we are left either with `Void` or an instance of `PosAtom`. Finally, our definition of `Type` from before is extended to include the `Intersect` term.

Figure 5 presents the rewrite rules for `Intersect` terms. Most of these are fairly straightforward. The need for a condition in Rule *Def 7(1)* may seem surprising, but the pattern matching language of WyRL does not (currently) allow one to match e.g. `Intersect[Type t, Type t]`. Finally, we note that, aside from Rule *Def 7(7)*, all of these rules are trivially convergent as they reduce the overall size of the term being rewritten. However, Rule *Def 7(7)* does not decrease the overall size of the term. Nevertheless, termination follows because it pushes `Intersect` inwards and, hence, this cannot continue indefinitely.

4.4. Subtyping Positive Atoms

Figure 2 employs a subtype relation over positive atoms which also requires special attention. We could attempt to encode this using a special term (e.g. `IsSubtype`) as we did with `Intersect`. However, we choose to eliminate the subtype relation altogether by reworking Figure 2. Our inspiration comes from the known connection between subtyping and intersection (i.e. $\mathtt{T}_1 \leq \mathtt{T}_2 \iff \mathtt{T}_1 \wedge \neg \mathtt{T}_2 = \mathtt{void}$). Instead of creating a

```
reduce Intersect[PosAtom t1, PosAtom t2]:
  => t1, if t1 == t2                                          // Def 7(1)

reduce Intersect[Any, PosAtom t]:
  => t                                                        // Def 7(2)

reduce Intersect[Type t, Any]:
  => t                                                        // Def 7(3)

reduce Intersect[Int, Pair]:
  => Void                                                     // Def 7(4)

reduce Intersect[Pair, Int]:
  => Void                                                     // Def 7(5)

reduce Pair[Type t1, Type t2]:                                // Def 7(6)
  => Void, if t1 is Void || t2 is Void

reduce Intersect[Pair[Type x1,Type x2],Pair[Type y1,Type y2]]:
  => let lhs = Intersect[x1,y1], rhs = Intersect[x2,y2]
     in Pair[lhs,rhs]                                         // Def 7(7)
```

Figure 5: Rewrite rules equivalent to Definition 7

new term (i.e. `IsSubtype`), we just reuse an existing one (i.e. `Intersect`). To that end, we rework rules $(3)-(5)$ from Figure 2 as follows:

$$\begin{aligned} \mathtt{T}_\mathtt{x}^+ \wedge \neg\mathtt{T}_\mathtt{y}^+ \wedge \ldots &\implies \mathtt{void} && \textbf{if } \mathtt{T}_\mathtt{x}^+ = \mathtt{T}_\mathtt{y}^+ \quad (3)\\ &\implies \mathtt{T}_\mathtt{x}^+ \wedge \ldots && \textbf{if } \mathtt{T}_\mathtt{y}^+ = \mathtt{void} \quad (4)\\ &\implies \mathtt{T}_\mathtt{x}^+ \wedge \neg(\mathtt{T}_\mathtt{x}^+ \sqcap \mathtt{T}_\mathtt{y}^+) \wedge \ldots && (5) \end{aligned}$$

To understand the justification for this, first consider rule (3) which previously required $\mathtt{T}_\mathtt{x}^+ \leq \mathtt{T}_\mathtt{y}^+$. The intention of this rule is that "*if the negative contribution contains the positive contribution, then only the empty set remains*". For example $\mathtt{int} \wedge \neg\mathtt{any}$ should reduce to `void`. At first glance, the new rule (3) above does not appear to fire in this case (i.e. since $\mathtt{int} \neq \mathtt{any}$). However, rule (5) is now unconditional and applies first to give $\mathtt{int} \wedge \neg(\mathtt{int} \sqcap \mathtt{any})$ and, since this reduces to $\mathtt{int} \wedge \neg\mathtt{int}$, the updated rule (3) now fires as expected. Consider now rule (4), which previously required that $\mathtt{T}_\mathtt{x}^+ \sqcap \mathtt{T}_\mathtt{y}^+ = \mathtt{void}$. The intention of this rule is that "*if the negative contribution subtracts nothing from the positive contribution, then the positive contribution remains*". Since rule (5) now applies unconditionally then, if this is true, we know $\mathtt{T}_\mathtt{y}^+$ still becomes `void` through the application of rule (5).

We now consider the implications for making rule (5) unconditional. This previously required $\mathtt{T}_\mathtt{x}^+ \not\geq \mathtt{T}_\mathtt{y}^+$, the purpose of which was to ensure termination rather than correctness. Without it, a term such as $\mathtt{any} \wedge \neg\mathtt{int}$ could be matched to produce $\mathtt{any} \wedge \neg(\mathtt{any} \sqcap \mathtt{int})$ which then simply reduces back to its original form (i.e. $\mathtt{any} \wedge \neg\mathtt{int}$). Indeed, by construction, every term previously prohibited by the constraint that now matches under rule (5) will return to its original form. Thus, we have a problem of termination with our updated rules rather than correctness. We return to this shortly.

Finally, we consider rule (6) from Figure 2, which is no longer represented in our

updated rules. Recall from 2.3 that the purpose of this was to ensure canonical conjuncts really are canonical but that this is not strictly required for correct operation.

4.5. Canonicalisation

To address the issue of termination introduced by our updated rule (5), we observe that it need only fire once for each negative atom. For example, given $\mathtt{T}_1^+ \wedge \neg\mathtt{T}_2^+ \wedge \neg\mathtt{T}_3^+$, we need only reduce this to $\mathtt{T}_1^+ \wedge \neg(\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+) \wedge \neg(\mathtt{T}_1^+ \sqcap \mathtt{T}_3^+)$ once and we are done. To that end, we introduce new syntax as follows:

```
define Negible as Void | Not(Posible)
define Posible as Void | Int | Any | Pair[Posible,Posible]
                | Intersect

term Canonical{PosAtom,Negible...}
```

Again, `Canonical` is added to the definition of `Type`. The intuition for `Canonical` is that it represents a conjunct which either *is* a canonical conjunct or is *becoming* one. A "`Negible`" is something like a negative atom, but not quite. Firstly, it can be `Void` as `Not(Any)` is a `NegAtom`, but we cannot prevent its reduction to `Void` (recall Figure 3). Secondly, a `Negible` may contain one or more `Intersect` subcomponents. Either way, a `Negible` eventually becomes either a `NegAtom`, `Any` or `Void`.

Given the above representation of canonical conjuncts, we implement the rules of Figure 2 with the WyRL rules given in Figure 6. The main feature of these rules is the "switch over" from `And{...}` conjuncts to `Canonical{...}` conjuncts. This ensures a general conjunct `And{...}` is converted into a `Canonical` *at most once*, and is achieved by the two rules labelled *Fig 2(5a)* and *Fig 2(5b)*. These exploit a special `Root` term defined as follows:

```
term Root(Type)
```

The `Root` term is used to signal the outermost position, since WyRL has no support for expressing this. Rules *Fig 2(5a)* and *Fig 2(5b)* operate on the `Root` in order to ensure the type is in the appropriate form before introducing a `Canonical`. Without `Root`, they could incorrectly apply to e.g. `Not(And{PosAtom})` which is not yet in the appropriate DNF form. They also apply `Intersect` to trim each negative atom $\neg\mathtt{T}_y^+$ to enforce the invariant $\mathtt{T}_x^+ \geq \mathtt{T}_y^+$ (where $\mathtt{T}_x^+$ is the corresponding positive atom).

Finally, to test whether a type `t1` is a subtype of another type `t2` we reduce the term `Root(And{t1,Not(t2)})`. This reduction either produces a term `Root(Void)` or a term which cannot be further reduced and, hence, is not equivalent to void.

5. Experimental Results

We now present results from experiments comparing the rewrite-based subtype operator developed in this paper with the existing ad-hoc implementation found in the Whiley compiler. The latter does not exploit rewriting in any structured sense, though it does expand terms on demand and employs a binary relation (implemented as a matrix) for handling recursive types. In particular, the algorithm exhibits exponential behaviour in pathological cases. Whilst performance is not a primary motivation for using a rewrite-based operator, it is nevertheless a concern. That is, if performance was orders-of-magnitude slower, this might be prohibitive. In our experiments, we employ three datasets: firstly, the set of subtype tests performed by the Whiley Compiler when

```
reduce And{Void, Type... ts}:                                  // Fig 2(1)
    => Void

reduce And{PosAtom t1, PosAtom t2, Type... ts}:
    => And(Intersect[t1,t2]++ts)                               // Fig 2(2)

reduce Canonical{PosAtom t1,                                   // Fig 2(3)
          Not(PosAtom t2), Negible... ts}:
    => Void, if t1 == t2

reduce Canonical{PosAtom p, Not(Void) x,                       // Fig 2(4)
                 Negible... ts}:
    => Canonical(p++ts)

reduce Root(And{PosAtom p,NegAtom... ns}):                     // Fig 2(5a)
 => let rs = {Not(Intersect[p,*n]) | n in ns}
    in Root(Canonical(p++rs))

reduce Root(Or{And{PosAtom p, NegAtom... ns},                  // Fig 2(5b)
               Type... ts}):
 => let rs = {Not(Intersect[p,*n]) | n in ns}
    in Root(Or(Canonical(p++rs) ++ ts))
```

Figure 6: Rewrite rules for our updated notion of Figure 2. Here, `*n` returns the argument for the term bound to `n`. For example, if `n` was bound to `Not(Int)`, then `*n` returns `Int`.

executing its test suite; secondly, the set of subtype tests performed by the Whiley Compiler when building the Whiley Benchmark Suite (WyBench); finally, a series of randomly generated input sets.

5.1. Dataset I — Whiley Compiler Tests

The Whiley Compiler (WyC) ships with 524 valid test cases for checking correctness against the Whiley Language Specification [91]. When compiling these tests, a large number of subtype queries are performed (approx 16K). As each test represents a syntactically correct Whiley file, many of these queries have a positive outcome. However, a surprising number also have a negative outcome ($\sim 12\%$). This is because method selection (amongst other things) in the presence of overloaded methods can result in failing subtype tests.[4] Figure 7 illustrates one of the test cases from the test suite for reference.

The Whiley language contains a more expressive type system than that considered here. This includes unions, intersections, negations and integers as considered here, but also arrays, references, records, recursive types, functions, and various other primitives (e.g. **`bool`**, **`byte`**, etc). We eliminated from our benchmark suite those tests involving recursive types, references, and functions. For the remainder, we translated all extra primitives (e.g. **`byte`**) into **`int`**. Likewise, records were translated directly into pairs by simply removing field names and trimming additional fields. Finally, arrays (e.g.

[4] More specifically, in order to resolve a method call against two overloaded methods of the same name with the same number of parameters, the Whiley compiler performs subtype tests against their parameters to determine which one is the target.

```
public export method test() :
    int|int[] x
    //
    if 0 == 1:                  // CHECK: int ≥ int
        x = 1                   // CHECK: int|int[] ≥ int
    else:
        x = [1, 2, 3]           // CHECK: int|int[] ≥ int[]
    //
    assume x == [1,2,3]         // CHECK: int|int[] ≥ int[]
                                // CHECK: bool ≥ bool
```

Figure 7: An example (valid) test case from the Whiley Compiler test suite. Compilation of this test results in a range of subtype tests being performed, as indicated.

`int[]`) were translated into pairs (e.g. `int` × `int`). This left roughly 15K subtype tests remaining which formed the first series of this data set.

Whilst our first series here provides a representative set of subtype queries, it does contain a large number of repeated tests. In particular, a large number of tests between primitive types (e.g. `int` ≤ `int`). These occur due to the prevalent use of primitive types within the benchmarks (which is to be expected). For example, a statement "`int x = 1`" generates such a test to ensure the right-hand side is assignable to the left-hand side. A valid concern here is that a high proportion of such tests might mask the costs of more interesting queries. Whilst this would be representative of a real-world workload, it nevertheless seemed prudent to explore the effect further. Therefore, the second series of this data set eliminates this effect by simply removing all duplicates from the first series and comprises around 290 subtype queries.

5.2. Dataset II — Whiley Benchmark Suite

The Whiley Benchmark Suite consists of 26 small benchmarks totalling around 3KLOC of code [112]. These cover various problems including: *N-Queens*, *LZ77 compression*, *matrix multiplication*, *Conway's Game of Life*, *tic tac toe*, *merge sort*, etc. Table 1 provides more details of the benchmark suite, and Figure 8 illustrates part of a benchmark for reference. When compiling this benchmark suite, the Whiley Compiler performs roughly 6K subtype queries. After filtering and transforming these as before, there are around 5.5K remaining which constitute the first series in this data set. Again, in the second series, all duplicates are removed leaving only 88 subtype queries.

5.3. Dataset III — Random Types

Our third dataset corresponds to a series of randomly generated input sets. To generate each input set, we first produce a fixed set of types generated from a given "space" uniformly at random. Using this set we generate every possible pairing from one type in the set to another, where each pairing corresponds to a subtype test. An important step here is that of generating a fixed set uniformly at random from a given "space". To do this, we need a mechanism to describe a "space" of types. Such a description must provide a simple mechanism for counting the number of types in a given space as we want to pick types arbitrarily without enumerating the entire space. This is simply because they grow exponentially, and enumerating them quickly becomes impractical.

Name	Description	LOC
001_average	Average over integer array.	44
002_fib	Recursive Fibonacci generator.	17
003_gcd	Classical GCD algorithm.	40
004_matrix	Straightforward matrix multiplication.	139
006_queens	Classical N-Queens problem.	68
007_regex	Regular expression matching.	72
008_scc	Tarjan's algorithm for finding strongly connected components	191
009_lz77	LZ77 compression / decompression.	136
010_sort	Merge Sort	105
011_codejam	Solution for Google CodeJam problem.	118
012_cyclic	Cyclic buffer.	140
013_btree	Binary search tree with insertion / lookup.	174
014_lights	Traffic lights sequence generator.	64
015_cashtill	Simple change determination algorithm.	217
016_date	Gregorian dates.	87
017_math	Simple math algorithms.	191
018_heap	Binary heap data structure.	24
022_cars	Controlling cars on bridge problem.	54
023_microwave	Classical microwave state machine.	88
024_bits	Algorithms for bit arrays.	98
025_tries	String trie with lookup / insertion.	121
026_reverse	Reversing an array.	57
101_interpreter	WHILE language interpreter.	325
102_conway	Conway's Game of Life.	161
104_tictactoe	Tic-Tac-Toe.	261
107_minesweeper	Minesweeper.	257

Table 1: Description of the Whiley benchmark suite.

```
function decompress(byte[] data) -> byte[]:
  byte[] output = [0b;0]
  nat pos = 0
  //
  while (pos+1) < |data| where pos >= 0:
    byte header = data[pos]
    byte item = data[pos+1]
    // NOTE: calculating offset here suboptimal as can test
    // directly against 00000000b, but helps verification as later
    // know that offset! = 0.
    u8 offset = integer::toUnsignedInt(header)
    pos = pos + 2
    if offset == 0:
      output = append(output,item)
    else:
      u8 len = integer::toUnsignedInt(item)
      // NOTE: start >= 0 not guaranteed. If negative, we have
      // error case and implementation proceeds producing junk.
      int start = |output| - offset
      int i = start
      // NOTE: i >= 0 required to handle case of start < 0 by
      // allowing implementation to proceed regardless.
      while i >= 0 && i < (start+len) where i < |output|:
          item = output[i]
          output = append(output,item)
          i = i + 1
  // all done!
  return output
```

Figure 8: Illustrating part of the `009_lz77` benchmark which implements decompression.

Name	Description	Tests
`WyC_Tests_1`	All subtype queries generated when compiling the valid test suite.	14979
`WyC_Tests_2`	All unique subtype queries generated when compiling the valid test suite.	290
`WyBench_1`	All subtype queries generated when compiling the Whiley benchmark suite (WyBench).	5567
`WyBench_2`	All unique subtype queries generated when compiling the Whiley benchmark suite (WyBench).	88
`TestSuite_1_2`	The complete space $\mathcal{T}_{1,2} \times \mathcal{T}_{1,2}$	324
`TestSuite_2_1`	The complete space $\mathcal{T}_{2,1} \times \mathcal{T}_{2,1}$	196
`TestSuite_2_2`	The space $\delta \times \delta$, where δ is 100 types chosen uniformly at random from $\mathcal{T}_{2,2}$	10000
`TestSuite_3_1`	The complete space $\mathcal{T}_{3,1} \times \mathcal{T}_{3,1}$	900
`TestSuite_3_2`	The space $\delta \times \delta$, where δ is 100 types chosen uniformly at random from $\mathcal{T}_{3,2}$	10000

Table 2: Benchmarks. Column "Tests" reports the number of subtypes tests in each dataset

We denote a space of types by $\mathcal{T}_{d,w}$ where *d*=*depth* and *w*=*width*. Intuitively, depth corresponds to the maximum nesting level permitted on any type in the space and, likewise, width corresponds to the maximum number of subcomponents for any type (e.g. the number of terms in a union). Negations have width 1, whilst pairs have width 2 and, finally, unions and intersections have width ≥ 2. For example, we have the following spaces:

$$
\begin{aligned}
&\mathcal{T}_{0,0}=\{\texttt{int}, \texttt{any}\}\\
&\mathcal{T}_{1,1}=\mathcal{T}_{0,0} \cup \{\neg\texttt{int}, \neg\texttt{any}\} \cup \{(\texttt{int}), (\texttt{any})\}\\
&\mathcal{T}_{1,2}=\mathcal{T}_{1,1} \cup\\
&\qquad \{\texttt{int}\times\texttt{int}, \texttt{int}\times\texttt{any}, \texttt{any}\times\texttt{int}, \texttt{any}\times\texttt{any}\} \cup\\
&\qquad \{\texttt{int}\vee\texttt{int},\ \texttt{int}\vee\texttt{any},\ \texttt{any}\vee\texttt{int},\ \texttt{any}\vee\texttt{any}\} \cup\\
&\qquad \{\texttt{int}\wedge\texttt{int},\ \texttt{int}\wedge\texttt{any},\ \texttt{any}\wedge\texttt{int},\ \texttt{any}\wedge\texttt{any}\}
\end{aligned}
$$

Note that $\mathcal{T}_{0,0} = \mathcal{T}_{0,1} = \mathcal{T}_{1,0}$ since a type without depth cannot have width, and vice versa. Also, observe that $\mathcal{T}_{1,1}$ doesn't include any unions or intersections as these are deemed only to make sense at a width of two or greater. For reference, we note: $|\mathcal{T}_{2,2}| = 1010$, $|\mathcal{T}_{3,2}| = 3062322$ and $|\mathcal{T}_{3,3}| = 179011590$.

For a given space $\mathcal{T}_{d,w}$ we can: calculate $|\mathcal{T}_{d,w}|$ with relative ease; and, map each integer from 0 upto $|\mathcal{T}_{d,w}|$ to a unique type in the space. Thus, we can select a fixed number of types uniformly at random using Algorithm S from Knuth [73]. In particular, this can be done without holding the entire space in memory (as this is prohibitive) and without enumerating each type in the space (also prohibitive).

5.4. Experimental Methodology

We now document our experimental methodology. For all experiments, WyRL v0.4.7 and Whiley v0.3.40 were used.[5][6] To enable the comparison between the existing Whiley implementation (in Java) with our WyRL implementation, we translate the WyRL rules into Java. This is done automatically by the WyRL tool which is designed specifically to generate Java source to ensure the generated rewrites are efficient [92]. Furthermore, the Whiley subtype operator was extracted from the compiler to give us a standalone experimental platform containing both implementations.[7]

We also provide a comparison against our Rascal implementation (see §7). Unfortunately, making a useful performance comparison here seems difficult. This is because Rascal does not allow one to generate a standalone rewrite system which can be compiled and run and, instead, relies on using a REPL. As such, the best performance comparison we could make saw Rascal running an order of magnitude slower.[8] We note that timing was performed using Rascal's internal `cpuTime()` function with all runs for each benchmark executed in the same JVM.

For a given dataset, the time taken to perform all subtype tests contained therein was recorded. Each subtype test is (essentially) a single line of text which must be parsed and converted into the appropriate internal form for the given subtype operator. To eliminate this cost from our timing data, it was performed before timing began. Furthermore, each experiment employed 25 warm up runs (whose data points were discarded) followed by 50 timed runs, where each run includes every subtype test in the dataset. As such, our timing data does not reflect the performance that might be expected from a "cold" JVM but, rather, reflects "steady-state" performance [54]. This may not be representative for a use in command-line compiler working on small programs. Nevertheless, it is reasonable for an incremental IDE (such as Eclipse) or even in the context of a command-line compiler operating on large programs. The standard deviation across runs is determined and we report the *coefficient of variation* (i.e. standard deviation / mean) to indicate the amount of variance observed between runs. We also compared the outcomes (i.e. whether or not a given subtype test holds or not) between the two implementations to sanity check them and this identified a small number of bugs in the existing ad-hoc implementation.

Finally, the experimental machine was a MacBook Pro with an Intel 2.7 GHz Core i5 with 8GB of RAM, running MacOS 10.11.2 and Oracle's Hotspot JVM version 1.8.0_66. The JVM was executed without any additional command-line options (e.g. for specifying maximum heap size, etc).

5.5. Experimental Results

Table 2 clarifies the datasets employed in the experiments, and Table 3 reports the relative performance of the two implementations. Overall, we find these results to be encouraging. The first observation is that, as expected, the rewriting implementation is always slower than the existing ad-hoc implementation. This is not surprising given the extra complexity involved in implementing a general purpose rewrite system. In most cases, the rewriting implementation is roughly twice as slow as the existing ad-hoc implementation. However, performance appears to degrade quickly as the average

[5] `http://github.com/Whiley/WhileyRewriteLanguage`

[6] `http://github.com/Whiley/WhileyCompiler`

[7] `http://github.com/DavePearce/RewritingTypeSystem`

[8] One should not take this as an indication that Rascal is inefficient (indeed, we believe the opposite), only that making a fair comparison is difficult.

Name	Whiley /ms	Rewriting /ms	Rewrites	Rascal /ms
WyC_Tests_1	59.0 (0.07)	110.0 (0.05)	6.0	1216.9 (0.01)
WyC_Tests_2	6.0 (0.3)	11.0 (0.24)	4.0	102.7 (0.28)
WyBench_1	25.0 (0.04)	49.0 (0.07)	3.0	473.2 (0.03)
WyBench_2	6.0 (0.24)	7.0 (0.22)	3.0	51.8 (0.33)
TestSuite_1_2	6.0 (0.21)	9.0 (0.25)	5.0	88.0 (0.36)
TestSuite_2_1	4.0 (0.23)	8.0 (0.22)	3.0	59.2 (0.26)
TestSuite_2_2	50.0 (0.04)	235.0 (0.06)	13.0	7587.0 (0.01)
TestSuite_3_1	11.0 (0.27)	19.0 (0.23)	6.0	334.9 (0.16)
TestSuite_3_2	62.0 (0.05)	1132 (0.18)	37.0	42902.5 (0.03)

Table 3: Experimental Results. Columns "Whiley", "Rewriting" and "Rascal" report the mean runtime for each implementation and, in brackets, the coefficient of variation. Finally, column "Rewrites" reports the mean number of rewrites required per test for our rewriting implementation.

number of rewrites increases. Indeed, perhaps the most surprising finding overall is that the average number of rewrites tends to be relatively low, particularly for the real-world Whiley Compiler test suite and WyBench benchmarks. Finally, we note the variance between runs (as indicated by the coefficient of variation in Table 3) indicates moderate variance for the short running benchmarks (which is to be expected) and quite low variance for the long running benchmarks. Looking at the results for Rascal, we observe similar asymptotic behaviour. As discussed earlier, obtaining a fair comparison with Rascal was challenging. Nevertheless, the results agree on those benchmarks for which rewriting is challenging. More specifically, the relative ordering of benchmarks (in terms of runtime) is unchanged between Rascal and WyRL.

Of course, we must exercise caution when interpreting Table 3. Whilst the Whiley Compiler test suite and the WyBench benchmarks contain real Whiley programs, they are still relatively small in size. As such, it is not clear how representative of expected compiler workloads they are. One thing, however, is that the artificial type spaces (i.e. $\mathcal{T}_{d,w}$) are most likely *not* representative of real workloads. In particular, a space such as $\mathcal{T}_{2,2}$ is swamped with relatively large subtype queries. This contrasts with our expectation that the norm will be relatively small queries (i.e. since primitive types are more prevalent than others). That is, for a subtype query to generate a large number of rewrites, the types involved need to be fairly complex. For example, having a large number of nested pairs (records in Whiley). We expect that, whilst such types will certainly occur, most will be primitives or relatively shallow records, etc. The results in Table 3 for the WyBench suite seem to support this view. Nevertheless, type spaces such as $\mathcal{T}_{2,2}$ and $\mathcal{T}_{3,2}$ provide interesting insight into the rewriting system's performance.

6. Discussion

Having presented the implementation of our type system as declarative rewrite rules in WyRL, we now consider the implementation of tuples and recursive types.

6.1. Tuple Types

Our original presentation supported tuple types of the form $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$, rather than just pairs as presented thus far. As expected, tuples represent ordered sequences and can be represented using WyRL's ordered collections. The following illustrates:

```
term Tuple[Type...]
```

Whilst most rules involving `Tuple[...]` terms can be correctly encoded in WyRL not all, unfortunately, can be. The key difficulty lies in the corresponding rules for Definition 5. Specifically, we replace rules 5a-7b with the following:

$$(\ldots, \bigvee_i \mathtt{T_i}, \ldots) \implies \bigvee_i (\ldots, \mathtt{T_i}, \ldots) \quad (5)$$
$$(\ldots, \bigwedge_i \mathtt{T_i}, \ldots) \implies \bigwedge_i (\ldots, \mathtt{T_i}, \ldots) \quad (6)$$
$$(\ldots, \neg \mathtt{T}, \ldots) \implies (\ldots, \mathtt{any}, \ldots) \wedge \neg(\ldots, \mathtt{T}, \ldots) \quad (7)$$

These rules require matching an arbitrary position within a tuple. Unfortunately, WyRL is not sufficiently expressive to match an arbitrary position within a list. That is, for example, we cannot write the following translation of rule (5) from above:

```
reduce Tuple[Type... x, Or{Type... y}, Type... z]:
  => let rs={Tuple(x++[t]++z) | t in y}
     in Or(rs)
```

This is invalid because (at the time of writing) the use of unbounded matching (i.e. "`Type...`") is currently limited to the right-most position only. As we'll see in §7 pattern matching of this nature is possible in Rascal, thus leading to a more complete implementation in that system.

6.2. Function Types

The original formalisation of the type system considered in this paper included tuple types, but not function types [87]. An interesting question is why they were omitted. The primary issue is around the need to provide a semantic interpretation of function types. In particular, this leads to a circularity between the type system and the subtype operator [52]. Castagna and Xu describe it thusly [23]:

> "*All definitions above run quite smoothly as long as basic and product types are the only atoms we consider (ie, the setting studied by Hosoya and Pierce [65]). But as soon as we add higher-order functions, that is, arrow types, the definitions above no longer work: If we take as $\mathcal{D}$ the set of all values, then this must include also λ-abstractions. Therefore, to define the semantic interpretation of types we need to define the type of λ-abstractions (in particular of the applications that may occur in their bodies) which needs the subtyping relation, which needs the semantic interpretation. We fall on a circularity.*"

To avoid this issue, the original formalisation of our type system simply chose tuples over function types. Nevertheless, the question remains whether or not function types can be encoded using declarative rewrite rules. In fact, it is not immediately clear how our system can be updated to accommodate this. Suppose we update our syntax (recall §2.1) with function types of the form $\mathtt{T_1} \rightarrow \mathtt{T_2}$. Whilst these appear similar to pairs they are, of course, contravariant in the parameter position and covariant in the return position. The standard subtyping rule clarifies this:

$$\frac{\mathtt{S_1} \leq \mathtt{T_1} \quad \mathtt{T_2} \leq \mathtt{S_2}}{\mathtt{T_1} \rightarrow \mathtt{T_2} \leq \mathtt{S_1} \rightarrow \mathtt{S_2}}$$

This contrasts, for example, with rule S-TUP1 from Figure 1. We can now begin updating Definition 5 as follows:

$$\begin{array}{l} \ldots \\ (\bigvee_i \mathtt{T_i}) \rightarrow \mathtt{T} \Longrightarrow \bigwedge_i (\mathtt{T_i} \rightarrow \mathtt{T}) \quad (8a) \\ \mathtt{T} \rightarrow (\bigvee_i \mathtt{T_i}) \Longrightarrow \bigvee_i (\mathtt{T} \rightarrow \mathtt{T_i}) \quad (8b) \\ (\bigwedge_i \mathtt{T_i}) \rightarrow \mathtt{T} \Longrightarrow \bigvee_i (\mathtt{T_i} \rightarrow \mathtt{T}) \quad (9a) \\ \mathtt{T} \rightarrow (\bigwedge_i \mathtt{T_i}) \Longrightarrow \bigwedge_i (\mathtt{T} \rightarrow \mathtt{T_i}) \quad (9b) \end{array}$$

Whilst the rules involving returns are identical as for pairs, those involving parameters differ. To understand why, consider that $(\mathtt{int} \vee \mathtt{bool}) \rightarrow \mathtt{int} \leq \mathtt{int} \rightarrow \mathtt{int}$ follows from our subtyping rule above. Then, the rewriting of $(\mathtt{int} \vee \mathtt{bool}) \rightarrow \mathtt{int}$ into $(\mathtt{int} \rightarrow \mathtt{int}) \wedge (\mathtt{bool} \rightarrow \mathtt{int})$ is consistent as the latter is, by construction, a subtype of $\mathtt{int} \rightarrow \mathtt{int}$. At this point it remains to complete our extension of Definition 5 (i.e. by specifying how negations are pulled out of function types). Likewise, we must update Definition 7. Since neither are immediately obvious, we leave them as future work.

6.3. Recursive Types

As discussed in §1.2, Whiley supports recursive types which are similar to the algebraic data types found in functional languages (e. g., Haskell, ML, etc). However, recursive types in Whiley build on unions for greater flexibility (i. e., as they are non-disjoint and structural). To illustrate subtyping of recursive types [86, 69], consider:

```
type List is null | { any data, List next }
type NonEmptyList is { any data, List next }
type IntList is {int data, IntList next} | null
```

This defines three variations on the basic concept of a linked list. As recursive types in Whiley always describe tree-like structures it is, for example, impossible for an instance of `List` to contain a cycle. Likewise, as data is immutable both `NonEmptyList` and `IntList` are implicit subtypes of `List`. Thus, any variable of type `NonEmptyList` or of type `IntList` can be passed into a function accepting a `List` as argument (e.g. `length()` from §1.2). As such, to enable this, the Whiley Compiler must reason about subtyping in the presence of recursion.

Contractiveness & Inhabitability. Another aspect of recursive types in Whiley is the need to check for *contractiveness* and *inhabitability*. The rough definition of a contractive type (adapted from Pierce [94]) would be that any infinite path through the graph representing the type must have infinite constructors (where the union, intersection and negation type combinators are not considered constructors). As a simple example, the following is rejected by the compiler:

```
type Invisible is Invisible | Invisible
```

This does not sensibly describe any type, as it never actually describes the structure of a value, and the compiler should report an error.

Whilst contractivity is a *necessary* requirement for well-formed recursive types, it is not *sufficient*. The following illustrates a contractive but *uninhabitable* type:

```
type InfList is { int data, InfList next }

function get(InfList l) -> (int d, InfList r):
   return l.data, l.next
```

Such a type would be perfectly reasonable in languages with lazy evaluation or implicit references [13] but, in Whiley, since all values are trees of *finite* depth it is impossible construct a value of type `InfList`.

As an aside, *non-recursive* uninhabitable types are also possible in the type system considered here. For example, $\texttt{int} \wedge \neg\texttt{int}$ is equivalent to `void`. Such types are concerning if programmers can accidentally write them without realising. However, this can be mitigated against by simply requiring that no type written at the source level is uninhabited (and, indeed, this is the approach taken in Whiley). Furthermore, detection of such types is relatively straightforward given an appropriate subtype operator.

Challenges. The handling of subtyping and checking for contractivity and inhabitability of recursive types present some real challenges for extending our rewriting-based implementation. To handle this properly requires some notion of *coinductive rewriting* [43]. To understand coinductive rewriting, we begin by considering the use of coinduction in a general sense. In a non-recursive setting, proving properties of a subtype operator is done in a well-established fashion using structural induction. Extending this to a subtype operator over recursive types requires some way of ensuring termination. That is, the subtype algorithm may recurse infinitely as recursive types describe infinite structures. Coinduction offers a simple solution which has, by now, been extensively used for reasoning about properties of recursive type systems [18, 53, 51, 5]. Compared with structural induction, however, structural *coinduction* is still not as widespread or as well understood. Kozen and Silva identified a useful motto regarding coinduction [77]:

> *"A property holds by induction if there is good reason for it to hold; whereas a property holds by coinduction if there is no good reason for it not to hold"*

In particular, with induction one establishes a base case and inductive step but, with coinduction, one need only establish the *coinductive step*. To further understand this, we now consider the problem of checking $\texttt{IntList} \leq \texttt{List}$ which, as we know, amounts to showing that $\texttt{IntList} \wedge \neg\texttt{List}$ reduces to `void`. For clarity, we do this in the context of our existing system. Hence, we define `List` and `IntList` as follows:

$$\begin{array}{rcl} \texttt{List} & \triangleq & \texttt{null} \vee (\texttt{any} \times \texttt{List}) \\ \texttt{IntList} & \triangleq & \texttt{null} \vee (\texttt{int} \times \texttt{IntList}) \end{array} \tag{8}$$

The derivation for rewriting this term begins as follows:

$$\begin{array}{rl} & \texttt{IntList} \wedge \neg\texttt{List} \\ \hookrightarrow & \big(\texttt{null} \vee (\texttt{int} \times \texttt{IntList})\big) \bigwedge \neg\big(\texttt{null} \vee (\texttt{any} \times \texttt{List})\big) \\ \hookrightarrow & \big(\texttt{null} \vee (\texttt{int} \times \texttt{IntList})\big) \bigwedge \neg\texttt{null} \wedge \neg(\texttt{any} \times \texttt{List}) \\ \hookrightarrow & \big(\texttt{null} \wedge \neg\texttt{null} \wedge \neg(\texttt{any} \times \texttt{List})\big) \bigvee \\ & \big((\texttt{int} \times \texttt{IntList}) \wedge \neg\texttt{null} \wedge \neg(\texttt{any} \times \texttt{List})\big) \\ \hookrightarrow & (\texttt{int} \times \texttt{IntList}) \wedge \neg(\texttt{any} \times \texttt{List}) \end{array}$$

At this point, we have made reasonably good progress in rewriting our original term to something approximating DNF. With some minor modifications, our WyRL implementation could certainly get this far. Consider the term we are rewriting here:

$$(\texttt{int} \times \texttt{IntList}) \wedge \neg(\texttt{any} \times \texttt{List}) \tag{9}$$

This is close to our original term, but represents the case when both lists have at least an integer head. Since neither `IntList` nor `List` are positive atoms, we cannot proceed further with our existing system.

Observe from above that knowing $\texttt{IntList} \wedge \neg \texttt{List} \Rightarrow \texttt{void}$ is enough to show that (9) above reduces to `void`. But, this is exactly what we were trying to show in the first place! In otherwords, we must apply a coinductive step. The following definition attempts to capture this succinctly:

Definition 8 (Coinductive Rewrite). *Let* $\texttt{T}_1$ *and* $\texttt{T}_3$ *be arbitrary terms. Then,* $\texttt{T}_1 \Longrightarrow \texttt{T}_3$ *if, for any intermediate term* $\texttt{T}_2$ *where* $\texttt{T}_1 \Longrightarrow \texttt{T}_2$ *and* $\texttt{T}_1 \neq \texttt{T}_2$*, we have* $\texttt{T}_2 \Longrightarrow \texttt{T}_3$ *whilst assuming* $\texttt{T}_1 \Longrightarrow \texttt{T}_3$.

Whilst this may seem fairly strange at first, it is analogous to the standard approach taken to applying coinduction when proving properties of type systems (e.g. subtyping). Pierce gives a good overview of the subject [94], and numerous applications of this principle are found in the literature [3, 76, 31, 18, 53, 65, 69]. The key difference here, however, is its application in the context of rewriting.

The challenge, of course, in extending our system to employ coinduction is considerable. Not least, the underlying WyRL system needs significant updates to support this. Likewise, our declarative rewrite rules themselves need to be reworked (for example) to eliminate the concepts of positive and negative atoms. That is, whilst these concepts are helpful in reasoning about the non-recursive system, they no longer make sense in the context of a fully recursive system.

7. Rascal Encoding

Rascal is a more recent tool which was strongly influenced by ASF+SDF [72]. Like ASF+SDF, Rascal focuses on program analysis and transformation but, unlike ASF+SDF, Rascal provides a single language for expressing these concepts. Rascal also provides additional pattern matching primitives and features over ASF+SDF. For example, Rascal does not require concrete syntax be given. Rascal also supports imperative language features, such as I/O and side-effects and, somewhat curiously, during backtracking side-effects are even undone (though I/O obviously cannot be). Like most other comparable tools, Rascal provides no support for ensuring confluence. Finally, Rascal has been previously used to implement non-trivial type checkers: firstly, for the purposes of checking type constraints in Featherweight Generic Java (FGJ) [72]; secondly, Rascal's type checker is itself written in Rascal [64].

From our perspective here Rascal provides similar (and, in fact, much more) functionality to WyRL (including support for associative-commutative matching). As we will see, we can encode the language of types and the corresponding rewrite rules considered here *including with full support for tuples*.

7.1. Preliminaries

Following §4.1, we begin with the definition of data types and the simple rewrites from Figure 3.

```
data Type = Any() | Void() | Int() | Not(Type l)
          | Tuple(list[Type] ls) | And(set[Type] es)
          | Or(set[Type] es);
```

This is very comparable with the WyRL implementation, except supporting arbitrary tuples instead of just pairs. One difference is that we do not split out the primitives into their own type `Primitive`. This is because Rascal does not support inheritance

between data types. [9] To better understand this, consider the definitions of `Primitive` and `Type` from §4.1 on page 14. Here, `Any` is implicitly an instance of both `Primitive` and `Type`. This fluidity in WyRL provides a powerful mechanism for capturing subsets of terms to match over. In particular, `PosAtom` and `NegAtom` (recall §4.3) are structured subsets of `Type` used in Figure 5. However, in our Rascal encoding, we cannot easily employ such structured subsets. Instead, we work around this using explicit matching functions (e.g. `isPosAtom()`) which offer a comparable solution:

```
bool isPosAtom(Any()) = true;
bool isPosAtom(Int()) = true;

bool isPosAtom(Tuple([Type t, *RS]))
 = isPosAtom(t) && (size(RS) == 0 || isPosAtom(Tuple(RS)));

default bool isPosAtom(Type t) = false;
```

As we will see, these functions are subsequently used in rules as constraints rather than the explicit types used in WyRL. Looking at the above, we start to see the underlying approach of Rascal. That is, Rascal combines higher-order functions that manipulate terms with powerful pattern matching capabilities. This constrasts with WyRL which provides explicit rewrite rules as the fundamental building block.

We can now give the Rascal equivalents to the simple rewrite rules from Figure 3:

```
Type reduce(Not(Any())) = Void();
Type reduce(Or({Void()})) = Void();

Type reduce(Or({Or({*Type ls}),*Type rs}))
 = Or(ls + rs);

Type reduce(And({And({*Type ls}),*Type rs}))
 = And(ls + rs);
```

These are essentially identical to their WyRL counterparts, though the syntax differs slightly. Specifically, `{*Type ks}` matches a set of zero or more `Type`s.

7.2. DNF Construction

We now begin considering the rules for moving a given type into DNF form, as given in Definition 5 and which correspond (roughly speaking) to the WyRL reductions in Figure 4 on page 16.

```
Type reduce(Not(Not(Type t))) = t;            // Def 5(1)

Type reduce(Not(Or({*Type ts})))              // Def 5(2)
 = And({Not(t) | Type t <- ts});

Type reduce(Not(And({*Type ts})))             // Def 5(3)
 = Or({Not(t) | Type t <- ts});

Type reduce(And({Or({*Type xs}),*Type ys}))
 = Or({And({x}+ys) | Type x<-xs});            // Def 5(4)
```

[9] See: `http://stackoverflow.com/questions/44895428/inheritance-for-algebraic-data-types-in-rascal`.

```
Type intersect(Type lhs, Type rhs) {
  switch(<lhs,rhs>) {
    case <Int(),Int()>:                          // Def 7(1)
      return Int();
    case <Any(),Type>:                           // Def 7(2)
      return rhs;
    case <Type,Any()>:                           // Def 7(3)
      return lhs;
    case <Int(),Tuple([*Type])>:                 // Def 7(4)
      return Void();
    case <Tuple([*Type]),Int()>:
      return Void();
    case <Tuple([*Type ns]),Tuple([*Type ms])>: {
      if(size(ns) != size(ms)) {
       return Void();
      } else {
       list[Type] r=[intersect(ns[i],ms[i])
                     | i<-[0..size(ns)] ];
       if(Void() in r) {
         return Void();
       } else {
         return Tuple(r);
  } } }}
  fail;
}
```

Figure 9: Illustrating the implementation of Definition 7 in Rascal. Observe this is modified to support tuples.

Again, the above rules are strikingly similar to their WyRL counterparts, aside from some small syntactic differences. However, at this point, our Rascal implementation diverges further as, unlike in WyRL, we can provide full support for tuples:

```
Type reduce(Tuple([*AS,Or({*Type rs}),*BS]))
 = Or({Tuple(AS+[r]+BS) | Type r<-rs});

Type reduce(Tuple([*AS,And({*Type rs}),*BS]))
 = And({Tuple(AS+[r]+BS) | Type r<-rs});

Type reduce(Tuple([*AS,Not(Type r),*BS]))
 = And({Tuple(AS + [Any()] + BS),
        Not(Tuple(AS + [r] + BS))});
```

The key here is that Rascal supports arbitrary matching within a list, thus allowing us to pull out unions and push down intersections and negations for an *arbitrary* tuple element (recall discussion in §6.1). For example, in the above we have `[*AS,Or({*Type rs}),*BS]` which matches a list containing an `Or` term with arbitrary terms before and after it. Since this is not currently possible in WyRL, it represents a useful improvement.

7.3. Subtype and Intersection Operators

Another useful advantage offered by Rascal is that we can encode the intersection and subtype operators of Figure 2 more directly and thereby avoid a separate `Intersect` term altogether. As such, the following corresponds more precisely to Figure 4 from the original presentation [87]:

```
bool isSubtype(Int(),Int()) = true;

bool isSubtype(Type t,Any()) = true;

bool isSubtype(Tuple([*LS]),Tuple([*RS]))
 = size(LS) == size(RS) &&
     (true | it && isSubtype(LS[i],RS[i])
           | int i <- [0..size(LS)]);

default bool isSubtype(Type t1,Type t2) = false;
```

Whilst these are mostly straightforward, the rule for tuples warrants attention. Specifically, this performs a map/reduce style operation which recursively invokes `isSubtype(_,_)` on corresponding elements from each tuple.

The implementation of the intersection operator is given in Figure 9 and corresponds roughly with the WyRL rules in Figure 5. This illustrates some of Rascal's powerful pattern matching in play, as the **`switch`** matches over pairs of types. A list comprehension is used to handle the more complex case for tuples.

7.4. Subtyping Positive Atoms

Unlike with the WyRL implementation, we can encode the rules of Figure 2 directly in Rascal.

```
Type reduce(And({Void(),*Type})) = Void();                  //(1)

Type reduce(And({Type i, Type j, *RS}))                     //(2)
= And({intersect(i,j)}+RS) when isPosAtom(i) && isPosAtom(j);

Type reduce(And({Type i,Not(Type j),*RS})) = Void()         //(3)
when isPosAtom(i) && isPosAtom(j) && isSubtype(i,j);

Type reduce(And({Type i,Not(Type j),*RS})) = And({i}+RS)    //(4)
when isPosAtom(i) && isPosAtom(j) && intersect(i,j)==Void();

Type reduce(And({Type i,Not(Type j),*RS}))                  //(5)
 = And({i,Not(intersect(i,j))}+RS)
when isPosAtom(i) && isPosAtom(j) && !isSubtype(i,j)
     && !isSubtype(j,i);
```

Here, we finally see the use of constraints, namely `isPosAtom()` to capture structured subtypes. Overall, the ability to call functions directly from rewrite rules simplifies the presentation considerably.

8. Related Work

The primary contribution of this paper is a demonstration that the subtype operator for a class of complex type systems can be encoded using a rewrite system. By complex, we mean that there is a large gap between the syntactic expression of a type and

its meaning. This gap arises from the presence of algebraic operators over types (i.e. union, intersection and negation) which enable reasoning through symbolic manipulation. That is, determining whether two types are equivalent (for example) requires manipulating one into the other (or, as we do, manipulating both into a normal form). In contrast, the type system for a typical mainstream language does not have such a gap. Java is one example, where subtyping does not involve much by the way of algebraic manipulation. Rather, its complexities arise from: determining the *Least Upper* or *Greatest Lower* bound of types in the inheritance hierarchy [78]; handling complex generic substitutions in the presence of existentials [4].

We argue that the class of type systems considered here is important in its own right, as evident from the considerable amount of work on such systems. Amadio and Cardelli were perhaps the first to develop a subtype operator for such a system involving unions and recursion [3]. Later, the work of Damm [31] and similarly Aiken and Wimmers [2] considered recursive subtyping with union and intersection types. More recently, the XDuce system of Hosoya and Pierce for representing XML schemas [65]. Frisch *et al.* developed CDuce as an extension of XDuce with function types [51, 10, 52]. Dardha *et al.* employ an expressive type system which includes negations, intersections and (implicitly) unions [32]. Work has also progressed in the other direction, by retro-fitting such operators onto existing languages. Büchi and Weck introduce *compound types* (similar to intersection types) to overcome limitations caused by a lack of multiple inheritance in Java [21]. Likewise Igarashi and Nagira introduce union types into Java [67], whilst, Plümicke employ intersection types in the context of type inference [95]. More recently, Dunfield presented a mechanism for "compiling out" unrestricted intersection and union types (a process they refer to as *elaboration*) [39]. For example, unrestricted unions are translated into disjoint unions, whilst implicit coercions to / from union types become tagging / untagging operations.

8.1. Rewriting

The use of rewriting to help programming language designers has, of course, a strong history. An early and influential tool here is ASF+SDF, which extended the Syntax Definition Formalism (SDF) with a term rewriting capability [105]. Rules are either interpreted or compiled to native C. ASF+SDF is naturally suited to the development of domain-specific languages, source code analysis and source code transformations and has found application in industry, particularly to support the prototyping and development of domain-specific languages [17].

Whilst WyRL was chosen as the underlying rewrite system here, the resulting reduction system is essentially language agnostic. We expect that it could, most likely, be encoded in a number of other systems (of which the Rascal implementation from §7 illustrates one example). However, we are not aware of any published attempt to explore type systems of the kind considered here in the context of rewriting. Rather prior work in this direction focuses primarily on relatively standard type systems (i.e. as found in mainstream languages) which do not require algebraic manipulation.

Veritas. The Veritas workbench allows one to first describe a type system and then automatically generate a soundness proof and efficient type checker [59, 56]. The emphasis is on reducing the risk of introducing errors when converting the specification of a type system into an efficient type checker. The authors argue that such specifications are typically presented only as typing judgements in academic papers (for example) which are then converted by hand into efficient type checkers. As such, Veritas provides a domain specific language for expressing type syntax and judgements in a very

similar style to that commonly used in type system presentations. The authors note this specification language is geared towards relatively simple type systems and, for example, doesn't support types with arbitrary components (e.g. tuples). As such, it does not provide the necessary machinery for handling the union, intersection and negation types considered here. To prove soundness, Veritas employs off-the-shelf automated theorem provers with domain-specific proof strategies targeted for type systems. For generating type checkers, certain optimisations can be applied in some cases (e.g. for eliminating transitivity subtyping rules). Thus, we see a clear benefit from separating the declarative specification of a type system from its underlying implementation as, for example, different optimisations and execution strategies can be employed and empirically evaluated. Finally, the process of translating declarative type judgements into first-order logic suitable for an automated theorem prover appears challenging. In particular, the authors performed separate studies into the effects of different translation strategies, finding they can significantly affect prover performance [58]. Likewise, they compared the effect of different provers (though finding less difference here) [57].

Spoofax. Spoofax is another language workbench aimed at allowing DSL designers to easily go from idea to implementation [109, 70]. In particular, Spoofax allows one to go from a language specification to a functioning editor in Eclipse with relative ease. The generated IDE plugin includes many of the services to which one is accustomed including, amongst other things, *syntax highlighting* and *semantically-aware completion* based on name and type analysis. For specifying syntax, Spoofax supports a variant on SDF which, for example, allows encoding context-free languages [63]. For specifying rewrites, Spoofax provides the Stratego transformation language [106]. In many ways, Stratego is similar to the WyRL language used here and, as with WyRL, no support is giving for ensuring confluence. A key feature of Stratego is the ability to represent rewrite *strategies* within the language itself (i.e. where rewrite rules and their applications are first-class entities) and, thus, one can easily experiment with different approaches. For example, one can specify that a "bottom up" strategy is used where rewrites are applied to children before their parents, etc. From our perspective, SDF+Stratego should allow one to encode the language of types and the corresponding rewrite rules considered here (though this remains to be shown). One issue is the lack of explicit support for the unordered (i.e. associative-commutative) collections found in WyRL which, instead, must be manually encoded in a somewhat cumbersome fashion. From the perspective of performance, it is difficult to compare Spoofax with WyRL given the former's embedding within Eclipse as a plugin. Finally, we note that Visser *et al.* later extended Spoofax with declarative languages for expressing different semantic components, such as for name resolution, operational semantics and typing rules, etc [107]. From this an interpreter can be automatically derived, along with Coq definitions upon which one can establish formal properties (e.g. type soundness).

Others. CafeOBJ is a formal specification language which inherits many features from the OBJ3 specification language [35, 36]. CafeOBJ is designed for developing industrial-strength models of software systems and verifying properties of them. CafeOBJ supports both equational and behavioural reasoning, where equations are regarded as left-to-right rewrite rules which are applied exhaustively. CafeOBJ is particularly well-suited to modelling and reasoning about event-based systems, such as verifying safety properties for security protocols [84]. ELAN provides a framework for developing deduction systems based on term rewriting, such as theorem provers, logic programming languages, constraint solvers, decision procedures, etc [14]. Like Stratego, ELAN en-

ables sophisticated user-defined strategies to control rewriting by making rules and strategies first-class entities. ELAN is perhaps unusual in having a strong focus on search, rather than deterministic computation [82]. This stems from its goal of supporting the development of deduction systems, which typically perform an exhaustive search for *unsatisfiability* (i.e. contradiction).

Another system with a strong focus on formal verification of models is Maude [27]. Again, Maude supports user-defined rewrite strategies through reflection. An unusual feature is the inclusion of an LTL model checker for checking reachability and invariants. Another interesting aspect of the Maude ecosystem is the number of tools which provide additional services, such as termination checking [40], inductive theorem proving [29], debugging [98], etc. Finally, TOM extends Java with constructs for describing term rewrite systems, allowing it to leverage the existing features of Java [7]. This contrasts with our use of `native` methods to describe general computation, as TOM extensions and Java methods interact seamlessly.

8.2. Flow Typing.

The ultimate motivation behind our work is flow typing, since Whiley's type system incorporates union, intersection and negation types primarily for this reason. Flow typing is often considered in the context of typing programs written in untyped languages. For example, Tobin-Hochstadt and Felleisen consider the problem of typing previously untyped Racket (aka Scheme) programs using a flow-typing algorithm [103, 102]. Their approach retypes variables within expressions dominated by type tests, and is referred to as *occurrence typing* [71]. However, they employ only union types and do not consider intersections or negations, instead preferring to conservatively approximate the intersection of two types in specific situations. The work of Dardha *et al.* employs a more expressive type system which includes negations, intersections and (implicitly) unions [32]. However, like the work of Frisch *et al.* [52], this does not detail an actual subtyping algorithm and, unfortunately, elides numerous technical details. The work of Guha *et al.* focuses on flow-sensitive type checking for JavaScript [61] and assumes programmer annotations are given for parameters. Their approach 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. Finally, Bierman *et al.* considers retyping after type tests for a core calculus supporting union, intersection and negation types [12].

The JVM Bytecode Verifier [79] is perhaps the most widely-used example of a flow typing system. Since locals and stack locations are untyped, the Verifier employs flow typing to ensure type safety. The verifier retypes variables after assignments, but does not retype them after `instanceof` tests. The following illustrates:

```
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 [78]. This tracks the type of a variable at each program point, allowing it easily to handle the above example.

To handle meet points in the control-flow graph, the Bytecode verifier computes the least upper bound of the types for each variable, rather than supporting explicit unions. This approach is conservative and means some programs will fail to verify that might otherwise be expected to pass. The problem here is that Java's subtype relation does not form a complete lattice [78]. 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`. Finally, several works on formalising the bytecode verifier have proposed the use of intersection types as an alternative solution [55, 97].

A reasonable amount of work on *non-null type systems* employs flow typing in various guises (e.g. [96, 49, 41, 80, 46]). The objective of such systems is to prevent variables which may hold **`null`** from being dereferenced. Non-null systems are interesting because they require variables be retyped after conditionals, as 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`.

Fähndrich and Leino discuss a system for checking non-null qualifiers in the context of C# [46]. 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 [41]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `v!=`**`null`** checks [96]. The JACK tool for verifying `@NonNull` type annotations extends the bytecode verifier with *type aliasing* [80]. This enables the system to retype a variable `x` as `@NonNull` in the body of an **`if`**`(x!=`**`null`**`)` conditional. The system does not employ union or intersection types directly but, instead, calculates them directly as needed (which is possible since the language of types is relatively simple).

Finally, we note some connection between flow typing and the use of *strong updates* in imperative languages [1]. Here, a strong update changes the type of a variable based on the value assigned to it at a given point. However, such retyping does not require the use of intersection and negation types as it is not connected with runtime type tests. Despite this, an interesting connection between flow typing and strong updates can be found in CQual [49, 48]. This is a flow-sensitive qualifier inference supporting numerous type qualifiers, including those for synchronisation and file I/Om which sup-

ports strong updates. However, CQual does not account for the effects of conditionals and, hence, retyping is impossible.

9. Conclusion

We have explored the use of declarative rewrite rules to implement subtyping for a system with unions, intersections and negations in the context of type checking. The starting point was our existing (paper-and-pencil) formalisation of this system. Whilst not designed with rewriting in mind, close parallels were apparent on reflection. The translation into our rewrite language, WyRL, was non-trivial and, indeed, a perfect translation was (just) beyond its expressive power. Specifically, we had to replace arbitrary tuples with pairs (§6.1) and rework the mechanisms for intersection (§4.3) and subtyping (§4.4) of atoms from our original formulation. We also looked at performance compared with the existing ad-hoc implementation found in the Whiley compiler. Overall, the results here were encouraging and we expect further gains are possible since WyRL has not been aggressively optimised. For example, there are situations where it performs more work than needed, such as deep cloning terms unnecessarily. We must caution, however, that it remains unclear how well our benchmark suites reflect expected real-world workloads and further research is required here. In particular, whilst WyBench is the most representative of the three benchmark suites considered, it is clear from Table 1 that they are still relatively small in size. Unfortunately, no larger real-world benchmark suite of Whiley programs exists at this time. We also presented an implementation of our system in Rascal though, unfortunately, a fair performance comparison was difficult. Nevertheless, this provides evidence that the general approach explored in this paper will apply beyond the specific system (WyRL) considered.

Finally, we discussed how our system might be extended to support arbitrary tuples, function types, and recursive types (though the latter remains challenging). It would also be interesting to consider how other features, such as parametric polymorphism, could be supported. Likewise, we note that other type systems (such as that of Castagna and Xu [23]) might benefit from the use of declarative rewriting and this would be interesting to explore. Finally, our experimental results do not consider subtyping for other domains, such as XML. For the latter we might expect different results given that XML structures can be very large with many levels of nesting. As such, this domain may pose some interesting challenges.

Acknowledgements. The author would like to thank the anonymous reviewers for many extremely helpful and insightful comments on early drafts.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15:575–631, 1993.

[4] Nada Amin and Ross Tate. Java and Scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 838–848. ACM Press, 2016.

[5] Davide Ancona and Andrea Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, volume 8586 of *LNCS*, pages 282–307. Springer-Verlag, 2014.

[6] 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)*, pages 57–74, 2006.

[7] E. Balland, P. Brauner, R. Kopetz, P. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Proceedings the Conference on Rewriting Techniques and Applications (RTA)*, pages 36–47, 2007.

[8] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In *Proceedings of the Conference on Theoretical Aspects of Computer Software (TACS)*, pages 651–674, 1991.

[9] C. Barrett and C. Tinelli. CVC3. In *Proceedings of Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.

[10] 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, 2003.

[11] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, volume 8586, pages 257–281. Springer-Verlag, 2014.

[12] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 105–116, 2010.

[13] Marcello M. Bonsangue, Jurriaan Rot, Davide Ancona, Frank S. de Boer, and Jan J. M. M. Rutten. A coalgebraic foundation for coinductive union types. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 62–73. Springer-Verlag, 2014.

[14] P. Borovanský, C. Kirchner, H. Kirchner, and P. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[15] J. Bos, M. Hills, P. Klint, T. Storm, and J. Vinju. Rascal: From algebraic specification to meta-programming. In *Proceedings Workshop on Algebraic Methods in Model-based Software Engineering, AMMSE*, volume 56 of *EPTCS*, pages 15–32, 2011.

[16] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - A functional language with dependent types. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 5674 of *LNCS*, pages 73–78, 2009.

[17] M. Brand, A. Deursen, P. Klint, S. Klusener, and E. Meulen. Industrial applications of ASF+SDF. In *Proceedings of the Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 9–18, 1996.

[18] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 1210 of *LNCS*, pages 63–81. Springer-Verlag, 1997.

[19] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):1–56, Winter 2006.

[20] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 174–177, 2009.

[21] 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.

[22] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 198–199, 2005.

[23] Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 94–106. ACM Press, 2011.

[24] Ceylon homepage. http://ceylon-lang.org/.

[25] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 227–247, 2007.

[26] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *Proceedings of the Conference on Automated Software Engineering (ASE)*, pages 485–488. ACM Press, 2007.

[27] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[28] M. Clavel, F. Durán, S. Eker, P. Lincoln, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *Proceedings the Conference on Rewriting Techniques and Applications (RTA)*, pages 76–87, 2003.

[29] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618—1650, 2006.

[30] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

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

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

[33] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[34] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[35] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST. 1998.

[36] Razvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 285(2):289–318, 2002.

[37] Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 60–72. ACM Press, 2017.

[38] Charles Donnelly and Richard Stallman. *Bison Manual: Using the YACC-compatible Parser Generator, for Version 1.29*. Free Software Foundation, Inc., 2000.

[39] Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.

[40] F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012.

[41] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[42] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Proc. GTTSE*, pages 422–436. Springer-Verlag, 2005.

[43] Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky, and Alexandra Silva. A coinductive framework for infinitary rewriting and equational reasoning. In *Proceedings the Conference on Rewriting Techniques and Applications (RTA)*, pages 143–159, 2015.

[44] Sebastian Erdweg, Tijs Van Der Storm, M. Voelter, M. Boersma, R. Bosman, William Cook, A. Gerritsen, A. Hulshout, S. Kelly, Alex Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, Van Der Vlist, K. B., G. Wachsmuth, Van Der Woning, and J. M. *The State Of The Art In Language Workbenches. Conclusions From The Language Workbench Challenge*. Springer-Verlag, 2013.

[45] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *SPLASH/OOPSLA Companion*, pages 307–309. ACM Press, 2010.

[46] 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)*, pages 302–312. ACM Press, 2003.

[47] Facebook flow. https://flowtype.org/.

[48] 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, 1999.

[49] 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, 2002.

[50] Martin Fowler. Language workbenches: the killer-app for doman specific languages?, 2005. https://www.martinfowler.com/articles/languageWorkbench.html.

[51] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proceedings of the ACM/IEEE Symposium on Logic In Computer Science (LICS)*, pages 137–146. IEEE Computer Society Press, 2002.

[52] 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.

[53] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2002.

[54] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 57–76. ACM Press, 2007.

[55] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proceedings of the confererence on Computer and Communications Security (CCS)*, pages 49–58, 1998.

[56] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Using Vampire in soundness proofs of type systems. In *Proceedings of the Vampire Workshop*, pages 33—-51, 2015.

[57] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Automating proof steps of progress proofs: Comparing Vampire and Dafny. In *Proceedings of the Vampire Workshop*, volume 44, pages 33–45, 2016.

[58] Sylvia Grewe, Sebastian Erdweg, Michael Raulf, and Mira Mezini. Exploration of language specifications by compilation to first-order logic. In *Proceedings of the Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 104–117. ACM Press, 2016.

[59] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. Type systems for the masses: deriving soundness proofs and efficient checkers. In *Proceedings of the ACM conference Onward!*, pages 137–150. ACM Press, 2015.

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

[61] 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.

[62] Görel Hedin and Eva Magnusson. Jastadd - a Java-based system for implementing front ends. *Electronic Notes in Computer Science*, 44(2):59–78, 2001.

[63] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF (reference manual). *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[64] Mark Hills, Paul Klint, and Jurgen J. Vinju. Program analysis scenarios in Rascal. In *Proceedings of the Workshop on Rewriting Logic and Its Applications (WRLA)*, pages 10–30. Springer-Verlag, 2012.

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

[66] 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, 2006.

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

[68] JetBrains. Meta programming system. http://www.jetbrains.com/mps.

[69] T. Jones and D. J. Pearce. A mechanical soundness proof for subtyping over recursive types. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP)*, page article 1, 2016.

[70] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 444–463. ACM Press, 2010.

[71] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 296–309. ACM Press, 2016.

[72] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. pages 168–177. IEEE Computer Society Press, 2009.

[73] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Second Edition, Addison-Wesley, Reading, 1981.

[74] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 311–331. Springer-Verlag, 2012.

[75] Kotlin homepage. http://kotlinlang.org/.

[76] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 419–428, 1993.

[77] Dexter Kozen and Alexandra Silva. Practical coinduction. *Mathematical Structures in Computer Science*, pages 1–21, 2016.

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

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

[80] 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.

[81] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proceedings of the confererence on Compiler Construction (CC)*, pages 73–88. Springer-Verlag, 2004.

[82] José Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81(7-8):721–781, 2012.

[83] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[84] Kazuhiro Ogata and Kokichi Futatsugi. Rewriting-based verification of authentication protocols. *Electronic Notes in Computer Science*, 71:208–222, 2004.

[85] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 425–436. ACM Press, 2011.

[86] D. J. Pearce. A calculus for constraint-based flow typing. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP)*, page Article 7, 2013.

[87] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–354, 2013.

[88] D. J. Pearce. Integer range analysis for Whiley on embedded systems. In *Proceedings of the IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 26–33, 2015.

[89] D. J. Pearce. The Whiley Rewrite Language (WyRL). In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 161–166, 2015.

[90] D. J. Pearce. Rewriting for Sound and Complete Union, Intersection and Negation Types. In *Proceedings of the Conference on 'Generative Programming: Concepts & Experience (GPCE)*, pages 117–130, 2017.

[91] D. J. Pearce. The Whiley Language Specification, Updated, 2016.

[92] D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 238–248, 2013.

[93] D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing Whiley. *Science of Computer Programming*, pages 191–220, 2015.

[94] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[95] Martin Plümicke. Intersection types in Java. In *Proceedings of the conference on Principles and Practices of Programming in Java (PPPJ)*, pages 181–188. ACM Press, 2008.

[96] 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)*, pages 334–554, 2001.

[97] 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.

[98] A. Riesco, A. Verdejo, and N. Martí-Oliet. A complete declarative debugger for Maude. In *Proceedings of the Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 216–225, 2010.

[99] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. Efficient software model checking of soundness of type systems. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 493–504. ACM Press, 2008.

[100] Sebastian Schweizer. Lifetime analysis for Whiley. Master's thesis, Department of Computer Science, University of Kaiserslautern, 2016.

[101] The Coq Development Team. The Coq proof assistant reference manual version 7.2. Technical Report 255, INRIA, 2002.

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

[103] 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.

[104] Mark van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen

Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Proceedings of the confererence on Compiler Construction (CC)*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[105] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. AMAST. 1996.

[106] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings the Conference on Rewriting Techniques and Applications (RTA)*, pages 357–362, 2001.

[107] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proceedings of the ACM conference Onward!*, pages 95–111. ACM Press, 2014.

[108] Markus Voelter and Vaclav Pech. Language modularity with the MPS language workbench. In *Proceedings of the International Conference of Software Engineering (ICSE)*, pages 1449–1450. IEEE Computer Society Press, 2012.

[109] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014.

[110] The Whiley Programming Language, http://whiley.org, accessed 2017.

[111] Whiley compiler respository on GitHub, http://github.com/whiley/whileycompiler, accessed 2017.

[112] The Whiley Benchmark Suite, http://github.com/whiley/wybench, accessed 2017.