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
djp@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 declarative 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.

In this paper, we consider a relatively complex type system involving unions, intersections and negations developed previously. 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. We identify that the original system can, for the most part, be reworked to enable a natural expression using declarative 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.

CCS Concepts • **Theory of computation → Rewrite systems**; *Type theory*; • **Software and its engineering →** *Translator writing systems and compiler generators*;

Keywords Rewrite Systems, Type Systems, Type Theory

ACM Reference Format:
David J. Pearce. 2017. Rewriting for Sound and Complete Union, Intersection and Negation Types. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 14 pages.
https://doi.org/10.1145/3136040.3136042

GPCE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136042

1 Introduction

In the pursuit of efficient, flexible and correct programming language implementations the language designer can benefit from a wide range of tools to aid his/her craft. Examples of such tools include *parser-generators* [27, 54, 57], *language workbenches* [31, 48, 70, 74], *meta-programming frameworks* [15, 46], *program transformation systems* [23, 72], *IDE generator frameworks* [20, 32] and more [11, 40, 41, 66]. General purpose term rewrite systems have been used in the engineering of programming languages and related tools for some time [55]. Success has been achieved in a wide range of areas, such as: *program transformation* [17, 72], *program analysis* [49], *formal verification* [26, 56], and *domain-specific languages* [71]. Numerous industrial-strength rewriting tools have also been developed, with notable examples including: CafeOBJ [26], ELAN [14], Maude [21, 22], Stratego [72], ASF+SDF [16, 71] and Rascal [49] (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.

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 [66]. 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) [40]. 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 [28, 51].

In developing the Whiley Verifying Compiler [6, 60, 63, 64], we stumbled upon a type system whose implementation is well-suited to declarative rewriting. This is really the class of structural type systems involving *union*, *intersection* and *negation* types. Such systems are interesting as developing a *sound* and *complete* subtype operator is challenging [8, 19, 24, 35]. 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. In previous work, we developed such a system and established its correctness [59]. 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

Flow-sensitive typing — or, *flow typing* — is a relatively new typing discipline arising from prior work on data-flow analysis and related flow-sensitive static analyses. Flow typing has recently been popularised by its inclusion in a number of up-and-coming programming languages, including *Ceylon (Redhat)* [1], *Kotlin (JetBrains)* [3], *Flow (Facebook)* [2], *Typescript (Microsoft)* [13], *Groovy* [4], *Racket* [69] and, of course, *Whiley* [59].

Union types (e.g. $\mathtt{T}_1 \vee \mathtt{T}_2$) provide an alternative to *algebraic data types* (ADTs) or *sum types*. Consider the following example in Whiley:

```
// Return index of first occurrence of c in str, or null if none
function indexOf(int[] xs,int x)->int|null:
    ...
```

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 `NullPointerExceptions` [29, 33, 53].

A defining characteristic of flow typing is the ability to *retype* variables after runtime type tests. This is typically achieved through *intersections* (e.g. $\mathtt{T}_1 \wedge \mathtt{T}_2$) and *negations* (e.g. $\neg\mathtt{T}_1$). 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`**).[1]

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

1.2 Contributions

In this paper, we develop a type checker using declarative rewrite rules for a type system developed previously [59]. This system provided a partial formalisation of the 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. 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. Finally, we 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 [12, 44]).

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.

2.1 Overview

We now examine our type system involving union, intersection and negation types developed previously [59]. 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`:

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

Here, `any` represents the set of all values, `void` the empty set, `int` the set of all integers and $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ tuples with one or more elements. The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. 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 `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, $\mathsf{T}_1 \vee \mathsf{T}_2$ is indistinguishable from $\mathsf{T}_2 \vee \mathsf{T}_1$. Likewise, $\mathsf{T}_1 \vee \mathsf{T}_1$ is indistinguishable from T_1. These assumptions were not strictly necessary, but simplified the formal presentation.

Semantics. To better understand the meaning of types, it is helpful to give them a *semantic interpretation* [8, 19, 24, 35]. This is a set-theoretic model where *subtype* corresponds to *subset*. The *domain* $\mathbb{D}$ is the set of all values constructible from integers or tuples, inductively defined as follows:

$$\mathbb{D} = \mathbb{Z} \cup \left\{ (\mathsf{v}_1, \ldots, \mathsf{v}_n) \mid \mathsf{v}_1 \in \mathbb{D}, \ldots, \mathsf{v}_n \in \mathbb{D} \right\}$$

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

Definition 2.1 (Semantic Interpretation)**.** Every type T is characterized by the set of values it contains, given by $[\![\mathsf{T}]\!]$:

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

Essentially, $[\![\mathsf{T}]\!]$ determines the set of values to which the type T corresponds. It is important to distinguish the *syntactic* representation T from its *semantic* interpretation $[\![\mathsf{T}]\!]$. 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 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 $\mathtt{int}$ and $\neg\neg\mathtt{int}$ have distinct syntactic representations, but are semantically indistinguishable. Similarly for $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ and $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$.

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

Definition 2.2 (Subtype Soundness)**.** A subtype operator, $\leq$, is *sound* if $\mathsf{T}_1 \leq \mathsf{T}_2 \implies [\![\mathsf{T}_1]\!] \subseteq [\![\mathsf{T}_2]\!]$, for any T_1 and T_2.

Definition 2.3 (Subtype Completeness)**.** A subtype operator, $\leq$, is *complete* if, for any types T_1 and T_2, it holds that $[\![\mathsf{T}_1]\!] \subseteq [\![\mathsf{T}_2]\!] \implies \mathsf{T}_1 \leq \mathsf{T}_2$.

A subtype operator which exhibits both of these properties is said to be *sound* and *complete* [19, 34, 35]. For illustration, Figure 1 presents a typical set of subtyping rules used elsewhere [68, 69]. For example, $((\mathtt{int}, \mathtt{int}), \mathtt{int}) \leq \neg(\mathtt{int}, \mathtt{int})$ holds by S-Tup2. Likewise, under S-Union1 it holds that $(\mathtt{int}, \mathtt{any}) \vee (\mathtt{any}, \mathtt{int}) \leq (\mathtt{any}, \mathtt{any})$, etc.

$$\frac{}{\mathsf{T} \leq \mathtt{any}} \quad \frac{}{\mathtt{void} \leq \mathsf{T}} \qquad \text{(S-Any, S-Void)}$$

$$\frac{}{\mathtt{int} \leq \neg(\mathsf{T}_1, \ldots, \mathsf{T}_n)} \qquad \text{(S-Int1)}$$

$$\frac{}{(\mathsf{T}_1, \ldots, \mathsf{T}_n) \leq \neg\mathtt{int}} \qquad \text{(S-Int2)}$$

$$\frac{\forall \mathtt{i}.\mathsf{T}_\mathtt{i} \leq \mathsf{S}_\mathtt{i}}{(\mathsf{T}_1, \ldots, \mathsf{T}_n) \leq (\mathsf{S}_1, \ldots, \mathsf{S}_n)} \qquad \text{(S-Tup1)}$$

$$\frac{\mathtt{n} \neq \mathtt{m} \vee \exists \mathtt{i}.\mathsf{T}_\mathtt{i} \leq \neg\mathsf{S}_\mathtt{i}}{(\mathsf{T}_1, \ldots, \mathsf{T}_n) \leq \neg(\mathsf{S}_1, \ldots, \mathsf{S}_\mathtt{m})} \qquad \text{(S-Tup2)}$$

$$\frac{\forall \mathtt{i}.\mathsf{T}_\mathtt{i} \geq \mathsf{S}_\mathtt{i}}{\neg(\mathsf{T}_1, \ldots, \mathsf{T}_n) \leq \neg(\mathsf{S}_1, \ldots, \mathsf{S}_n)} \qquad \text{(S-Tup3)}$$

$$\frac{\forall \mathtt{i}.\mathsf{T}_\mathtt{i} \leq \mathsf{S}}{\mathsf{T}_1 \vee \ldots \vee \mathsf{T}_n \leq \mathsf{S}} \qquad \text{(S-Union1)}$$

$$\frac{\exists \mathtt{i}.\mathsf{T} \leq \mathsf{S}_\mathtt{i}}{\mathsf{T} \leq \mathsf{S}_1 \vee \ldots \vee \mathsf{S}_n} \qquad \text{(S-Union2)}$$

$$\frac{\exists \mathtt{i}.\mathsf{T}_\mathtt{i} \leq \mathsf{S}}{\mathsf{T}_1 \wedge \ldots \wedge \mathsf{T}_n \leq \mathsf{S}} \qquad \text{(S-Intersect1)}$$

$$\frac{\forall \mathtt{i}.\mathsf{T} \leq \mathsf{S}_\mathtt{i}}{\mathsf{T} \leq \mathsf{S}_1 \wedge \ldots \wedge \mathsf{S}_n} \qquad \text{(S-Intersect2)}$$

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

The rules of Figure 1 are sound with respect to Definition 2.1, but not complete. For example, neither of the following hold under Figure 1 (but are implied by Definition 2.1):

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

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

This problem of finding a sound and complete subtype operator has been studied extensively [8, 9, 19, 35, 44, 59].

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 $\mathtt{void}$ [59]. 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 [35, 59]:

$$\mathtt{T}_1 \leq \mathtt{T}_2 \iff \mathtt{T}_1 \wedge \neg\mathtt{T}_2 = \mathtt{void} \quad (1)$$

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

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

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 [59].

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 2.4 (Type Atoms)**.** Let $\mathtt{T}^*$ denote a *type atom*:

$$\begin{array}{l} \mathtt{T}^* ::= \mathtt{T}^+ \mid \mathtt{T}^- \\ \mathtt{T}^- ::= \neg\mathtt{T}^+ \\ \mathtt{T}^+ ::= \mathtt{any} \mid \mathtt{int} \mid (\mathtt{T}^+_1, \ldots, \mathtt{T}^+_n) \end{array}$$

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

From Definition 2.4 we see that a negative atom is simply a negated positive atom. Furthermore, the elements of tuple 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 $\mathtt{T}$ into an equivalent of the following form:

$$\bigvee_i \bigwedge_j \mathtt{T}^*_{i,j} \quad (2)$$

This was achieved in our original presentation using a procedure, $\mathtt{DNF}(\mathtt{T})$ defined as follows:

Definition 2.5 (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 $\mathtt{T}'$.

$$\begin{array}{lll} \neg\neg\mathtt{T} & \Longrightarrow \mathtt{T} & (1) \\ \neg \bigvee_i \mathtt{T}_i & \Longrightarrow \bigwedge_i \neg\mathtt{T}_i & (2) \\ \neg \bigwedge_i \mathtt{T}_i & \Longrightarrow \bigvee_i \neg\mathtt{T}_i & (3) \\ \left(\bigvee_i \mathtt{S}_i\right) \wedge \bigwedge_j \mathtt{T}_j & \Longrightarrow \bigvee_i \left(\mathtt{S}_i \wedge \bigwedge_j \mathtt{T}_j\right) & (4) \\ (\ldots, \bigvee_i \mathtt{T}_i, \ldots) & \Longrightarrow \bigvee_i (\ldots, \mathtt{T}_i, \ldots) & (5) \\ (\ldots, \bigwedge_i \mathtt{T}_i, \ldots) & \Longrightarrow \bigwedge_i (\ldots, \mathtt{T}_i, \ldots) & (6) \\ (\ldots, \neg\mathtt{T}, \ldots) & \Longrightarrow (\ldots, \mathtt{any}, \ldots) \wedge \neg(\ldots, \mathtt{T}, \ldots) & (7) \end{array}$$

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

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 $(5-6)$ are somewhat non-standard and pull unions, intersections and negations *outside* of tuples. For example, type $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ is rewritten into a union of positive atoms $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$.

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

Definition 2.6 (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 2.6 makes sense if we consider $\mathtt{T}_1 \wedge \neg\mathtt{T}_2$ as $\mathtt{T}_1 - \mathtt{T}_2$; thus, in rule 1 the amount "subtracted" from the positive atom by any given negative atom is strictly less than the total. For example, $(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{any})$ is not permitted since this corresponds to `void`. Likewise, $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any})$ is not permitted either as it is more precisely represented as $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$. Rule 2 prohibits negative atoms from subsuming each other, such as in $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{int})$.

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. 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 (e.g. $\mathtt{int} \wedge \neg\mathtt{any} \Longrightarrow \mathtt{void}$). Rule 4 catches negative components which lie outside the domain (e.g. $\mathtt{int} \wedge \neg(\mathtt{int}, \mathtt{int}) \Longrightarrow \mathtt{int}$). Rule 5 covers negative components needing to be trimmed (e.g. $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any}) \Longrightarrow (\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$).

$$
\begin{array}{lcll}
\mathtt{void} \wedge \ldots & \Longrightarrow & \mathtt{void} & (1) \\
\mathtt{T}_i^+ \wedge \mathtt{T}_j^+ \wedge \ldots & \Longrightarrow & (\mathtt{T}_i^+ \sqcap \mathtt{T}_j^+) \wedge \ldots & (2) \\
\mathtt{T}_x^+ \wedge \neg \mathtt{T}_y^+ \wedge \ldots & \Longrightarrow & \mathtt{void} \quad \textbf{if } \mathtt{T}_x^+ \leq \mathtt{T}_y^+ & (3) \\
 & \Longrightarrow & \mathtt{T}_x^+ \wedge \ldots \quad \textbf{if } \mathtt{T}_x^+ \sqcap \mathtt{T}_y^+ = \mathtt{void} & (4) \\
 & \Longrightarrow & \mathtt{T}_x^+ \wedge \neg(\mathtt{T}_x^+ \sqcap \mathtt{T}_y^+) \wedge \ldots \quad \textbf{if } \mathtt{T}_x^+ \not\geq \mathtt{T}_y^+ & (5) \\
\neg \mathtt{T}_x^+ \wedge \neg \mathtt{T}_y^+ \wedge \ldots & \Longrightarrow & \neg \mathtt{T}_x^+ \wedge \ldots \quad \textbf{if } \mathtt{T}_x^+ \geq \mathtt{T}_y^+ & (6)
\end{array}
$$

Figure 2. Mechanism for constructing canonical conjuncts from arbitrary conjuncts of the form $\bigwedge_i \mathtt{T}_i^*$. 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 $\bigwedge_i \mathtt{T}_i^*$, not just the leftmost two (as the presentation might suggest).

Finally, rule 6 catches redundant negative components (e.g. $\ldots \wedge \neg(\mathtt{int}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \Longrightarrow \ldots \wedge \neg(\mathtt{int}, \mathtt{any})$).

2.3 Problem Statement

A key observation from Definition 2.6 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. Finally, such an encoding allows automatic generation of a type checker, thereby closing the gap between the type system design (i.e. our original presentation [59]) and its implementation (i.e. in the Whiley compiler). Indeed, the current implementation in the compiler bears little resemblance to our presentation.

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 [61]. We choose this tool simply because it is familiar to us and is already part of the Whiley compiler. However, the general approach taken here should be applicable to other similar tools (e.g. Spoofax [48, 75], Rascal [49], etc).

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:

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

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

This defines a language of 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 occurence 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, *we must carefully ensure our 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 Tuple Types

Whilst union and intersection types represent unordered collections of types, tuples 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 Tuple[Type...]

reduce And{Tuple[Type... x], Tuple[Type... y],
           Type... rest}:
 => Void, if |x| != |y|
 => let r = [And{x[i],y[i]} | i in 0..|x|]
    in And(Tuple(r) ++ rest)
```

This rule reduces intersection of tuple types to that of intersecting element types, where intersecting two tuples of different arity reduces to `Void`. The reduction rule employs a *list comprehension* to enable iterating over an arbitrary number of subterms. Thus, the term `And{Tuple[Int],Tuple[Int,Int]}` reduces to `Void`, whilst `And{Tuple[x1,x2],Tuple[y1,y2]}` reduces to `And{Tuple[And{x1,y1},And{x2,y2}]}`, etc.

3.4 Remarks

We have now explored various aspects of WyRL and illustrated, roughly speaking, how a type system involving unions, intersections and negations can be represented. What remains is to encode the rewrite rules from Definition 2.5 and Figure 2. There are two challenges: the *subtype* and *intersection* operators used in Figure 2. These are problematic as they represent non-trivial computation that cannot easily be described within WyRL itself. That is, WyRL has no support for writing arbitrary functions. Nevertheless, with some care, they can be encoded directly as rewrite rules within WyRL.

4 Implementation

We now present our completed implementation of the type system from §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, *recursive types* [47, 58]).

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. For example, there remain numerous long-standing open issues (e.g. [5], issues #645,#585,#561). Whilst such issues are by no means impossible to fix, they typically require extensive refactoring and can represent 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 [67].

4.1 Preliminaries

We begin with preliminaries before presenting the main pieces of our solution. First, the syntax of types is as follows:

```
term Void
term Any
term Int
define Primitive as Any | Void | Int

term Tuple[Type...]
term Not(Type)
term Or{Type...}
term And{Type...}
define Type as Primitive | Tuple | Not | Or
              | And | ...
```

These are largely as indicated from §3, although the "..." on the last line indicates space for additional types to be introduced later. 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.

```
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 2.5(1)
  => t

reduce Not(Or{Type... ts}):                 // Def 2.5(2)
  => let xs = { Not(t) | t in ts }
     in And(xs)

reduce Not(And{Type... ts}):                // Def 2.5(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 2.5(4)

reduce Tuple[Or{Type... xs},Type...ts]:
  => let ys = { Tuple(r++ts) | r in xs }
     in Or(ys)                              // Def 2.5(5a)

reduce Tuple[Type t1, Or{Type... xs}, Type...ts]:
  => let ys={ Tuple([t1,r]++ts) | r in xs }
     in Or(ys)                              // Def 2.5(5b)

reduce Tuple[And{Type... xs},Type...ts]:
  => let ys={ Tuple(r++ts) | r in xs }
     in And(ys)                             // Def 2.5(6a)

reduce Tuple[Type t1, And{Type... xs}, Type...ts]:
  => let ys={ Tuple([t1,r]++ts) | r in xs }
     in And(ys)                             // Def 2.5(6b)

reduce Tuple[Not(Type t1),Type...ts]:
  => let lhs=Tuple(Any++ts), rhs=Not(Tuple(t1++ts))
     in And{lhs,rhs}                        // Def 2.5(7a)

reduce Tuple[Type t1, Not(Type t2), Type...ts]:
  => let lhs = Tuple([t1,Any]++ts),
         rhs = Not(Tuple([t1,t2]++ts))
     in And{lhs,rhs}                        // Def 2.5(7b)
```

Figure 4. Rewrite rules roughly equivalent to Definition 2.5.

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:

$$\bigvee_i \bigwedge_j \mathtt{T}^*_{i,j} \tag{3}$$

Figure 4 presents the rewrite rules corresponding to Definition 2.5. Again, these are mostly straightforward and employ set comprehensions to manage terms of arbitrary size. However, there is one shortcoming which represents an important limitation. Specifically, we can only translate rules (5) – (7) for *specific cases* rather than for the *general case*. For example, *Def 2.5(5a)* from Figure 4 matches the case $(\bigvee_i \mathtt{T}_i, \ldots)$, whilst *Def 2.5(5b)* matches $(\mathtt{T}, \bigvee_i \mathtt{T}_i, \ldots)$. Unfortunately, WyRL is not sufficiently expressive (at the time of writing) to match an arbitrary position within a list. That is, for example, we cannot write the following translation of rule (5):

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

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 4.1 (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:

$$\begin{array}{lll}
\mathtt{T}^+ \sqcap \mathtt{T}^+ & = \mathtt{T}^+ & (1)\\
\mathtt{any} \sqcap \mathtt{T}^+ & = \mathtt{T}^+ & (2)\\
\mathtt{T}^+ \sqcap \mathtt{any} & = \mathtt{T}^+ & (3)\\
\mathtt{int} \sqcap (\mathtt{T}^+_1, \ldots, \mathtt{T}^+_n) & = \mathtt{void} & (4)\\
(\mathtt{T}^+_1, \ldots, \mathtt{T}^+_n) \sqcap \mathtt{int} & = \mathtt{void} & (5)\\
(\mathtt{T}^+_1, \ldots, \mathtt{T}^+_n) \sqcap (\mathtt{S}^+_1, \ldots, \mathtt{S}^+_m) & & \\
\qquad & = \mathtt{void}, \textbf{if } \mathtt{n} \neq \mathtt{m} & (6)\\
\qquad & = \mathtt{void}, \textbf{if } \exists \mathtt{i}.\mathtt{T}^+_\mathtt{i} \sqcap \mathtt{S}^+_\mathtt{i} = \mathtt{void} & (7)\\
\qquad & = (\mathtt{T}^+_1 \sqcap \mathtt{S}^+_1, \ldots, \mathtt{T}^+_n \sqcap \mathtt{S}^+_n), \textbf{else} & (8)
\end{array}$$

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 | Tuple[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 4.1(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]`. Furthermore, rule *Def 4.1(7)* is not an exact translation of its counterpart from Definition 4.1 (as that cannot be expressed). However, the net effect achieved is the same.

```
reduce Intersect[PosAtom t1, PosAtom t2]:
  => t1, if t1 == t2                          // Def 4.1(1)

reduce Intersect[Any, PosAtom t]:
  => t                                        // Def 4.1(2)

reduce Intersect[Type t, Any]:
  => t                                        // Def 4.1(3)

reduce Intersect[Int, Tuple]:
  => Void                                     // Def 4.1(4)

reduce Intersect[Tuple,Int]:
  => Void                                     // Def 4.1(5)

reduce Intersect[Tuple[Type... n],Tuple[Type... m]]:
  => Void, if |n| != |m|                      // Def 4.1(6)

reduce Tuple[Type... ns]:                     // Def 4.1(7)
  => Void, if some { t in ns | t is Void }

reduce Intersect[Tuple[Type... n],Tuple[Type... m]]:
  => let es = [Intersect[n[i],m[i]] | i in 0..|n|]
     in Tuple(es), if |n|==|m|                // Def 4.1(8)
```

Figure 5. Rewrite rules equivalent to Definition 4.1

4.4 Subtyping Positive Atoms

Figure 2 employs a subtype operator 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 operator altogether by reworking Figure 2. Our inspiration comes from the known connection between subtyping and intersection (i.e. $\mathsf{T}_1 \leq \mathsf{T}_2 \iff \mathsf{T}_1 \wedge \neg \mathsf{T}_2 = \mathtt{void}$). Instead of creating a 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:

$$\begin{aligned} \mathsf{T}^+_x \wedge \neg \mathsf{T}^+_y \wedge \ldots &\implies \mathtt{void} && \text{if } \mathsf{T}^+_x = \mathsf{T}^+_y && (3)\\ &\implies \mathsf{T}^+_x \wedge \ldots && \text{if } \mathsf{T}^+_y = \mathtt{void} && (4)\\ &\implies \mathsf{T}^+_x \wedge \neg(\mathsf{T}^+_x \sqcap \mathsf{T}^+_y) \wedge \ldots && && (5) \end{aligned}$$

To understand the justification for this, first consider rule (3) which previously required $\mathsf{T}^+_x \leq \mathsf{T}^+_y$. 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 $\mathsf{T}^+_x \sqcap \mathsf{T}^+_y = \mathtt{void}$. 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 T^+_y still becomes `void` through the application of rule (5).

We now consider the implications for making rule (5) unconditional. This previously required $\mathsf{T}^+_x \not\geq \mathsf{T}^+_y$, the purpose of this was to ensure confluence 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 confluence with our updated rules rather than correctness. We return to address this shortly.

Finally, we consider rule (6) from Figure 2, which is no longer represented in our updated rules. The purpose of this was to ensure 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 would be reduced by rule (6). 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`. As such, rule (6) from Figure 2 is not actually required.

4.5 Canonicalisation

To address the issue of confluence introduced by our updated rule (5), we observe that it need only fire once for each negative atom. For example, given $\mathsf{T}^+_1 \wedge \neg \mathsf{T}^+_2 \wedge \neg \mathsf{T}^+_3$, we need only reduce this to $\mathsf{T}^+_1 \wedge \neg(\mathsf{T}^+_1 \sqcap \mathsf{T}^+_2) \wedge \neg(\mathsf{T}^+_1 \sqcap \mathsf{T}^+_3)$ 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
       | Tuple[Posible...] | Intersect

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

Again, `Canonical` is added to the definition of `Type`. The intuition is that `Canonical` 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

```
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,                      // Fig 2(4)
          Not(Void) x, 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,                    // Fig 2(5b)
              NegAtom... ns}, 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.

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 $\neg \mathsf{T}_y^+$ to enforce the invariant $\mathsf{T}_x^+ \geq \mathsf{T}_y^+$ (where T_x^+ is the corresponding positive atom).

Finally, to test whether `t1` is a subtype of `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. 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 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 [62]. 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.

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 tuples by simply removing field names. Finally, arrays (e.g. **`int[]`**) were translated into unit tuples (e.g. **`(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` $\leq$ `int`). To reduce this effect, the second series of this data set is simply the first with all duplicates removed and comprises around 290 subtype queries.

5.2 Dataset II — Whiley Benchmark Suite

The Whiley Benchmark Suite consists of 26 small benchmarks totalling around 2.5KLOC of code [7]. These cover various problems including: *N-Queens*, *LZ77 compression*, *matrix multiplication*, *Conway's Game of Life*, *tic tac toe*, *merge sort*, etc. 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.

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 elements in a tuple, or terms in a union). Negations have width 1, whilst tuples have width ≥ 1 and, finally, unions and intersections have width ≥ 2. For example, we have the following spaces:

$\mathcal{T}_{0,0}$={`int`, `any`}
$\mathcal{T}_{1,1}=\mathcal{T}_{0,0} \cup \{\neg$`int`, $\neg$`any`$\} \cup \{$(`int`), (`any`)$\}$
$\mathcal{T}_{1,2}=\mathcal{T}_{1,1} \cup$
{(`int`, `int`), (`int`, `any`), (`any`, `int`), (`any`, `any`)} ∪
{`int`∨`int`, `int`∨`any`, `any`∨`int`, `any`∨`any`} ∪
{`int`∧`int`, `int`∧`any`, `any`∧`int`, `any`∧`any`}

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 [50]. 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. 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 [63]. Furthermore, the Whiley subtype operator was extracted from the compiler to give us a standalone experimental platform containing both implementations. [2]

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 [36]. 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. Finally, 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 (which was not surprising).

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 1 clarifies the datasets employed in the experiments, and 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 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.

Of course, we must exercise caution when interpreting Table 1. 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 our expectation that the norm will be relatively small queries (i.e. since primitive types are more prevalent than others). Nevertheless, these type spaces provide some interesting insight into the rewriting system's performance.

6 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

[2] http://github.com/DavePearce/RewritingTypeSystem

Table 1. Experimental Results. Column "Tests" reports the number of subtypes tests in each dataset. Columns "Whiley" and "Rewriting" 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.

Name	Description	Tests	Whiley /ms	Rewriting /ms	Rewrites
WyC_Tests_1	All subtype queries generated when compiling the valid test suite.	14979	59.0 (0.07)	110.0 (0.05)	6.0
WyC_Tests_2	All unique subtype queries generated when compiling the valid test suite.	290	6.0 (0.3)	11.0 (0.24)	4.0
WyBench_1	All subtype queries generated when compiling the Whiley benchmark suite (WyBench).	5567	25.0 (0.04)	49.0 (0.07)	3.0
WyBench_2	All unique subtype queries generated when compiling the Whiley benchmark suite (WyBench).	88	6.0 (0.24)	7.0 (0.22)	3.0
TestSuite_1_2	The complete space $\mathcal{T}_{1,2} \times \mathcal{T}_{1,2}$	324	6.0 (0.21)	9.0 (0.25)	5.0
TestSuite_2_1	The complete space $\mathcal{T}_{2,1} \times \mathcal{T}_{2,1}$	196	4.0 (0.23)	8.0 (0.22)	3.0
TestSuite_2_2	The space $\delta \times \delta$, where δ is 100 types chosen uniformly at random from $\mathcal{T}_{2,2}$	10000	50.0 (0.04)	235.0 (0.06)	13.0
TestSuite_3_1	The complete space $\mathcal{T}_{3,1} \times \mathcal{T}_{3,1}$	900	11.0 (0.27)	19.0 (0.23)	6.0
TestSuite_3_2	The space $\delta \times \delta$, where δ is 100 types chosen uniformly at random from $\mathcal{T}_{3,2}$	10000	62.0 (0.05)	1132 (0.18)	37.0

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 [52]; handling complex generic substitutions in the presence of existentials [10].

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 [9]. Later, the work of Damm [24] and similarly Aiken and Wimmers [8] considered recursive subtyping with union and intersection types. More recently, the XDuce system of Hosoya and Pierce for representing XML schemas [44]. Frisch *et al.* developed CDuce as an extension of XDuce with function types [12, 34, 35]. Dardha *et al.* employ an expressive type system which includes negations, intersections and (implicitly) unions [25]. 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 [18]. Likewise Igarashi and Nagira introduce union types into Java [45], whilst, Plümicke employ intersection types in the context of type inference [65].

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 using other systems (and, indeed, an encoding in Rascal can be found in the accompanying technical report). 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.

The **Veritas** workbench allows one to first describe a type system and then automatically generate a soundness proof and efficient type checker [37, 40]. 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 [39]. Likewise, they also compared the effect of using different provers (though finding less difference here) [38].

Spoofax is another language workbench aimed at allowing DSL designers to easily go from idea to implementation [48, 75]. 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 [42]. For specifying rewrites, Spoofax provides the Stratego transformation language [72]. 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 [73]. From this an interpreter can be automatically derived, along with Coq definitions upon which one can establish formal properties (e.g. type soundness).

Rascal is a more recent tool which was strongly influenced by ASF+SDF [49]. 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). Finally, like most other tools considered here, Rascal provides no direct support for ensuring confluence.

From our perspective here Rascal provides similar (and, in fact, much more) functionality to WyRL (including support for associative-commutative matching). We can, for example, encode the language of types and the corresponding rewrite rules considered here (our accompanying technical report illustrates such an encoding). Unfortunately, making a useful performance comparison with WyRL seems, unfortunately, difficult or impossible. 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 several orders of magnitude slower (even on e.g. $\mathcal{T}_{1,2}$).[3] Nevertheless, Rascal offers useful advantages here. For example, we can encode the intersection and subtype operators of Figure 2 more directly and thereby avoid the `Intersect` term. Likewise Rascal supports arbitrary matching within a list, thus allowing a translation of rules (5) – (7) from Definition 2.5 for the general case (i.e. rather than for specific cases, as discussed in §4.2). However, Rascal also has some disadvantages and, in particular, does not support inheritance between data types. Consider the definitions of `Primitive` and `Type` from §4.1. 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 (e.g. `PosAtom` and `NegAtom` from §4.3 are structured subsets of `Type` used in Figure 5). In our Rascal encoding, we work around this using explicit matching functions (e.g. `isPosAtom()`) which offer a comparable solution.

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) [49]; secondly, Rascal's type checker is itself written in Rascal [43].

7 Conclusion

We have explored the use of declarative rewrite rules to implement subtyping for a system with unions, intersections and negations. 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. 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. In the future, we are interested in extending the system to more closely resemble that found in Whiley. Support for recursive types would be valuable (though requires coinductive rewriting [30]).

[3] 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.

References

[1] Ceylon Homepage. Retrieved 2017 from http://ceylon-lang.org/

[2] Facebook Flow. Retrieved 2017 from https://flowtype.org/

[3] Kotlin Homepage. Retrieved 2017 from http://kotlinlang.org/

[4] What's new in Groovy 2.0? Retrieved 2017 from http://www.infoq.com/articles/new-groovy-20

[5] Whiley Compiler Respository on GitHub. Retrieved 2017 from http://github.com/Whiley/WhileyCompiler

[6] The Whiley Programming Language. Retrieved 2017 from http://whiley.org

[7] The Whiley Benchmark Suite. Retrieved 2017 from http://github.com/Whiley/WyBench

[8] Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture (FPCA)*. 31–41.

[9] Roberto M. Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems* 15 (1993), 575–631. Issue 4.

[10] Nada Amin and Ross Tate. 2016. 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)*. ACM Press, 838–848.

[11] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. 2006. A framework for implementing pluggable type systems. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 57–74.

[12] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. 51–63.

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

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

[15] J. Bos, M. Hills, P. Klint, T. Storm, and J. Vinju. 2011. Rascal: From Algebraic Specification to Meta-Programming. In *Proceedings Workshop on Algebraic Methods in Model-based Software Engineering, AMMSE (EPTCS)*, Vol. 56. 15–32.

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

[17] M. Bravenboer, A. van Dam, K. Olmos, , and E. Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae* 69, 1-2 (Winter 2006), 1–56.

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

[19] Castagna and Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 198–199.

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

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

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

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

[24] Flemming M. Damm. 1994. Subtyping with Union Types, Intersection Types and Recursive Types. In *Proc. TACS*. LNCS, Vol. 789. 687–706.

[25] Ornela Dardha, Daniele Gorla, and Daniele Varacca. 2012. *Semantic Subtyping for Objects and Classes*. Technical Report. Laboratoire PPS, Université Paris Diderot.

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

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

[28] Torbjörn Ekman and Görel Hedin. 2005. Modular Name Analysis for Java Using JastAdd. In *Proceedings of the Conference on Generative and Transformational Techniques in Software Engineering (GTTSE)*. Springer-Verlag, 422–436.

[29] T. Ekman and G. Hedin. 2007. Pluggable Checking and Inferencing of Non-Null Types for Java. *Journal of Object Technology* 6, 9 (2007), 455–475.

[30] Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky, and Alexandra Silva. 2015. A Coinductive Framework for Infinitary Rewriting and Equational Reasoning. In *Proceedings the Conference on Rewriting Techniques and Applications (RTA)*. 143–159.

[31] 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. 2013. *The State Of The Art In Language Workbenches. Conclusions From The Language Workbench Challenge*. Springer-Verlag.

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

[33] M. Fähndrich and K. R. M. Leino. 2003. 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)*. ACM Press, 302–312.

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

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

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

[37] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. 2015. Using Vampire in Soundness Proofs of Type Systems. In *Proceedings of Vampire Workshop*. 33—-51.

[38] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. 2016. Automating Proof Steps of Progress Proofs: Comparing Vampire and Dafny. In *Proceedings of the Vampire Workshop*, Vol. 44. 33–45.

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

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

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

[42] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. 1989. The Syntax Definition Formalism SDF – reference manual –. *ACM SIGPLAN Notices* 24, 11 (1989), 43–75.

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

[44] Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology* 3, 2 (2003), 117–148.

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

[46] JetBrains. Meta Programming System. Retrieved 2017 from http://www.jetbrains.com/mps

[47] T. Jones and D. J. Pearce. 2016. A Mechanical Soundness Proof for Subtyping over Recursive Types. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP)*. article 1.

[48] Lennart C. L. Kats and Eelco Visser. 2010. 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)*. ACM Press, 444–463.

[49] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. IEEE Computer Society Press, 168–177.

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

[51] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Proceedings of the Conference on Software Language Engineering (SLE)*. Springer-Verlag, 311–331.

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

[53] C. Male, D.J. Pearce, A. Potanin, and C. Dymnikov. 2008. Java Bytecode Verification for @NonNull Types. In *Proceedings of the confererence on Compiler Construction (CC)*. 229–244.

[54] Scott McPeak and George C. Necula. 2004. Elkhound: A Fast, Practical GLR Parser Generator. In *Proceedings of the confererence on Compiler Construction (CC)*. Springer-Verlag, 73–88.

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

[56] Kazuhiro Ogata and Kokichi Futatsugi. 2004. Rewriting-Based Verification of Authentication Protocols. *Electronic Notes in Computer Science* 71 (2004), 208–222.

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

[58] D. J. Pearce. 2013. A Calculus for Constraint-Based Flow Typing. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP)*. Article 7.

[59] D. J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 335–354.

[60] D. J. Pearce. 2015. Integer Range Analysis for Whiley on Embedded Systems. In *Proceedings of the IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*. 26–33.

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

[62] D. J. Pearce. Updated, 2016. The Whiley Language Specification. (Updated, 2016).

[63] D. J. Pearce and L. Groves. 2013. Whiley: a Platform for Research in Software Verification. In *Proceedings of the Conference on Software Language Engineering (SLE)*. 238–248.

[64] D. J. Pearce and L. Groves. 2015. Designing a Verifying Compiler: Lessons Learned from Developing Whiley. *Science of Computer Programming* (2015), 191–220.

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

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

[67] Sebastian Schweizer. 2016. *Lifetime Analysis for Whiley*. Master's Thesis. Department of Computer Science, University of Kaiserslautern.

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

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

[70] 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. 2001. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Proceedings of the confererence on Compiler Construction (CC) (LNCS)*, Vol. 2027. Springer-Verlag, 365–370.

[71] Arie van Deursen, Jan Heering, and Paul Klint (Eds.). 1996. *Language Prototyping: An Algebraic Specification Approach*.

[72] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings the Conference on Rewriting Techniques and Applications (RTA)*. 357–362.

[73] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabrieël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Proceedings of the ACM conference Onward!* ACM Press, 95–111.

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

[75] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (2014), 35–43.