Rewriting for Sound and Complete Union, Intersection and Negation Types

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

`@WhileyDave`
`http://whiley.org`
`http://github.com/Whiley`

Flow Typing

```
type LinkedList is null | { LinkedList next, int data }

function length(LinkedList list) → (int r)
// Length cannot be negative
ensures r >= 0:
    //
    if list is null:
        return 0
    else:
        return 1 + length(list.next)
```

- Whiley uses **flow typing** (as do others).
- Looks cool, but it is really a **nightmare**.
- Hard stuff: **negations**, **intersections**, **recursive types**.

An Interesting Type System

$$\texttt{T} \quad ::= \quad \texttt{any} \mid \texttt{void} \mid \texttt{int} \mid (\texttt{T}_1, \ldots, \texttt{T}_n) \mid \neg\texttt{T} \mid \texttt{T}_1 \wedge \texttt{T}_2 \mid \texttt{T}_1 \vee \texttt{T}_2$$

- **Equivalences.** E.g. $\texttt{any} \equiv \neg\texttt{void}$, $(\texttt{int} \wedge \texttt{any}) \equiv \texttt{int}$.
- **Subtyping**. e.g. $(\texttt{int} \wedge \neg\texttt{int}) \leq \texttt{void}$?

Q) How to implement *sound* and *complete* subtype operator?

$$\frac{}{\mathtt{T} \leq \mathtt{any}} \qquad \frac{}{\mathtt{void} \leq \mathtt{T}}$$

$$\frac{}{\mathtt{int} \leq \neg(\mathtt{T}_1,\ldots,\mathtt{T}_n)} \qquad \frac{}{(\mathtt{T}_1,\ldots,\mathtt{T}_n) \leq \neg\mathtt{int}}$$

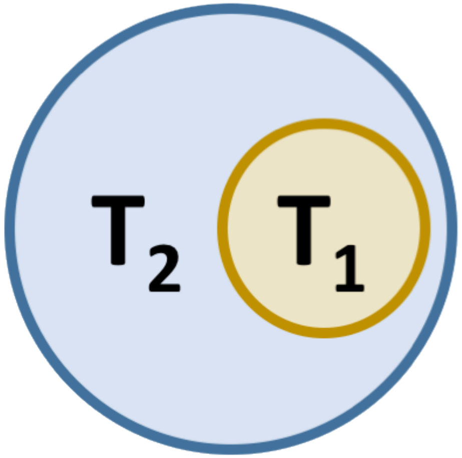
$$\frac{\forall \mathtt{i}.\mathtt{T}_\mathtt{i} \leq \mathtt{S}_\mathtt{i}}{(\mathtt{T}_1,\ldots,\mathtt{T}_n) \leq (\mathtt{S}_1,\ldots,\mathtt{S}_n)} \qquad \frac{\mathtt{n} \neq \mathtt{m} \vee \exists \mathtt{i}.\mathtt{T}_\mathtt{i} \leq \neg\mathtt{S}_\mathtt{i}}{(\mathtt{T}_1,\ldots,\mathtt{T}_n)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

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

$$\frac{\forall \mathtt{i}.\mathtt{T}_\mathtt{i} \leq \mathtt{S}}{\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n \leq \mathtt{S}} \qquad \frac{\exists \mathtt{i}.\mathtt{T} \leq \mathtt{S}_\mathtt{i}}{\mathtt{T} \leq \mathtt{S}_1 \vee \ldots \vee \mathtt{S}_n}$$

$$\frac{\exists \mathtt{i}.\mathtt{T}_\mathtt{i} \leq \mathtt{S}}{\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n \leq \mathtt{S}} \qquad \frac{\forall \mathtt{i}.\mathtt{T} \leq \mathtt{S}_\mathtt{i}}{\mathtt{T} \leq \mathtt{S}_1 \wedge \ldots \wedge \mathtt{S}_n}$$

Subtyping as Rewriting



$$\mathtt{T}_1 \leq \mathtt{T}_2 \iff \mathtt{T}_1 \land \neg \mathtt{T}_2 \equiv \mathtt{void}$$

Subtyping as Rewriting

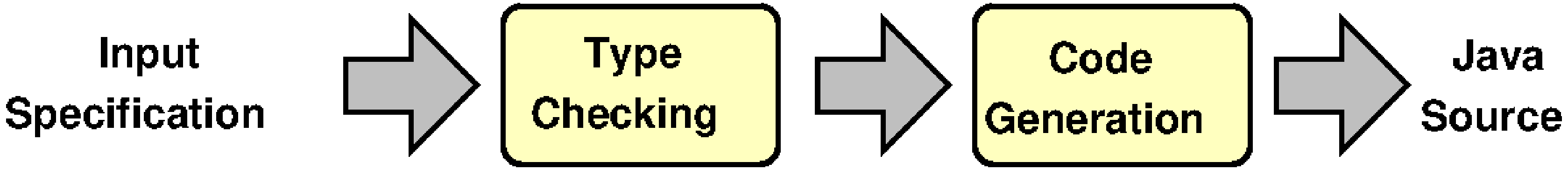
$$\bigvee_{\mathtt{i}} \bigwedge_{\mathtt{j}} \mathtt{T}^{*}_{\mathtt{i},\mathtt{j}}$$

- **Rewriting** is key to implementing subtype relation.
- Roughly speaking, rewrite to **Disjunctive Normal Form**.

  $$\begin{aligned}&\big(\mathtt{int} \vee (\mathtt{int}, \mathtt{int})\big) \wedge (\mathtt{any}, \mathtt{any})\\ \rightarrow\ &\big(\mathtt{int} \wedge (\mathtt{any}, \mathtt{any})\big) \vee (\mathtt{int} \wedge \mathtt{any}, \mathtt{int} \wedge \mathtt{any})\\ \rightarrow\ &\mathtt{void} \vee (\mathtt{int}, \mathtt{int})\end{aligned}$$

- **See:** *“Sound and Complete Flow Typing with Unions, Intersections and Negations”*, VMCAI’13.

Whiley Rewrite Language (WyRL)



- **WyRL** is a Domain-Specific Rewrite Language.
- Used previously for developing Automated Theorem Prover.
- Generates **Java source** for efficiency.
- **See:** *“The Whiley Rewrite Language (WyRL)”*, SLE’15.

Encoding our Type System (Syntax)

$$\mathtt{T} \quad ::= \quad \mathtt{any} \mid \mathtt{void} \mid \mathtt{int} \mid (\mathtt{T}_1, \ldots, \mathtt{T}_n) \mid \neg\mathtt{T} \mid \mathtt{T}_1 \wedge \mathtt{T}_1 \mid \mathtt{T}_1 \vee \mathtt{T}_2$$

```
term Any
term Void
term Int
term Not(Type)
term Tuple[Type...]
term Neg(Type)
term And{Type...}
term Or{Type...}

define Type as Any | Void | Int | Not | Tuple | Neg | And | Or
```

- **E.g.** $\mathtt{int} \vee (\mathtt{int}, \mathtt{int})$ is $\mathtt{Or}\{\mathtt{Int}, \mathtt{Tuple}[\mathtt{Int}, \mathtt{Int}]\}$.
- **E.g.** $\mathtt{And}\{\mathtt{Any}, \mathtt{Int}\}$ identical to $\mathtt{And}\{\mathtt{Int}, \mathtt{Any}\}$.
- **E.g.** $\mathtt{And}\{\mathtt{Int}, \mathtt{Int}\}$ *not* identical to $\mathtt{Int}$

Encoding our Type System (Rewrite Rules)

Simple

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

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

Unordered (AC)

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

Comprehensions

```
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)
```

Challenges

$$
\begin{array}{llll}
\mathtt{void} \wedge \ldots & \Longrightarrow & \mathtt{void} & \\
\mathtt{T}_{\mathtt{i}}^{+} \wedge \mathtt{T}_{\mathtt{j}}^{+} \wedge \ldots & \Longrightarrow & (\mathtt{T}_{\mathtt{i}}^{+} \sqcap \mathtt{T}_{\mathtt{j}}^{+}) \wedge \ldots & \\
\mathtt{T}_{\mathtt{x}}^{+} \wedge \neg \mathtt{T}_{\mathtt{y}}^{+} \wedge \ldots & \Longrightarrow & \mathtt{void} & \textbf{if } \mathtt{T}_{\mathtt{x}}^{+} \leq \mathtt{T}_{\mathtt{y}}^{+} \\
 & \Longrightarrow & \mathtt{T}_{\mathtt{x}}^{+} \wedge \ldots & \textbf{if } \mathtt{T}_{\mathtt{x}}^{+} \sqcap \mathtt{T}_{\mathtt{y}}^{+} = \mathtt{void} \\
 & \Longrightarrow & \mathtt{T}_{\mathtt{x}}^{+} \wedge \neg (\mathtt{T}_{\mathtt{x}}^{+} \sqcap \mathtt{T}_{\mathtt{y}}^{+}) \wedge \ldots & \textbf{if } \mathtt{T}_{\mathtt{x}}^{+} \not\geq \mathtt{T}_{\mathtt{y}}^{+} \\
\neg \mathtt{T}_{\mathtt{x}}^{+} \wedge \neg \mathtt{T}_{\mathtt{y}}^{+} \wedge \ldots & \Longrightarrow & \neg \mathtt{T}_{\mathtt{x}}^{+} \wedge \ldots & \textbf{if } \mathtt{T}_{\mathtt{x}}^{+} \geq \mathtt{T}_{\mathtt{y}}^{+}
\end{array}
$$

- What is this *intersect operator* $\mathtt{T}_{\mathtt{x}}^{+} \sqcap \mathtt{T}_{\mathtt{y}}^{+}$ ... ?
- What is this *subtype operator* $\mathtt{T}_{\mathtt{x}}^{+} \geq \mathtt{T}_{\mathtt{y}}^{+}$ ... ?
- How to patten match on arbitrary list element?
- How do we ensure *confluence* and *termination*?

Experimental Results (Part 1)

Name	Tests	Whiley /ms	Rewriting /ms	Rewrites
`WyC_Tests_1`	14979	59.0 (0.07)	110.0 (0.05)	6.0
`WyC_Tests_2`	290	6.0 (0.3)	11.0 (0.24)	4.0
`WyBench_1`	5567	25.0 (0.04)	49.0 (0.07)	3.0
`WyBench_2`	88	6.0 (0.24)	7.0 (0.22)	3.0

- **WyC_Tests.** All subtype queries generated when compiling **524** Whiley programs used in test suite (14979 tests, 290 unique).

- **WyBench.** All subtype queries generated when compiling **26** Whiley programs used in WyBench benchmark suite (5567 tests, 88 unique).

Random Type Generation

Type Space

Denote a space of types by $\mathcal{T}_{d,w}$ where *d=depth* and *w=width*.

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

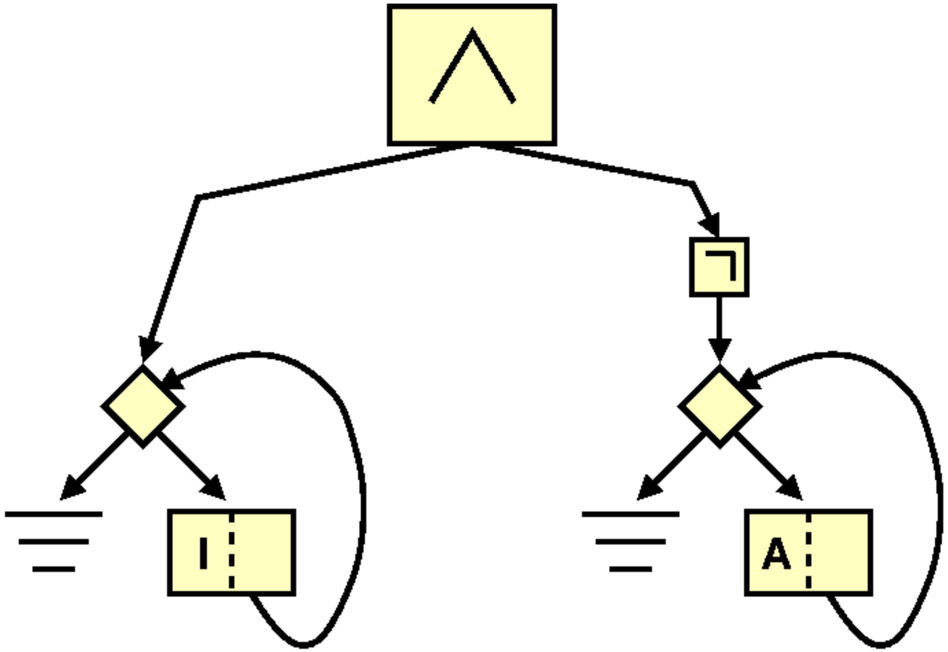
- $\mathcal{T}_{0,0} = \mathcal{T}_{0,1} = \mathcal{T}_{1,0}$
- $|\mathcal{T}_{2,2}| = 1010$, $|\mathcal{T}_{3,2}| = 3062322$ and $|\mathcal{T}_{3,3}| = 179011590$.
- $\mathcal{T}_{d,w}$ is countable, can calculate $|\mathcal{T}_{d,w}|$ easily

Experimental Results (Part 2)

Name	Tests	Whiley /ms	Rewriting /ms	Rewrites
`TestSuite_1_2`	324	6.0 (0.21)	9.0 (0.25)	5.0
`TestSuite_2_1`	196	4.0 (0.23)	8.0 (0.22)	3.0
`TestSuite_2_2`	10000	50.0 (0.04)	235.0 (0.06)	13.0
`TestSuite_3_1`	900	11.0 (0.27)	19.0 (0.23)	6.0
`TestSuite_3_2`	10000	62.0 (0.05)	1132 (0.18)	37.0

- **TestSuite_1_2**. The complete space $\mathcal{T}_{1,2} \times \mathcal{T}_{1,2}$.
- **TestSuite_2_1**. The complete space $\mathcal{T}_{2,1} \times \mathcal{T}_{2,1}$.
- **TestSuite_2_2**. The space $\delta \times \delta$, where δ is 100 types chosen uniformly at random from $\mathcal{T}_{2,2}$.
- **TestSuite_3_1**. The complete space $\mathcal{T}_{3,1} \times \mathcal{T}_{3,1}$.
- **TestSuite_3_2**. The space $\delta \times \delta$, where δ is 100 types chosen uniformly at random from $\mathcal{T}_{3,2}$.

Recursive Types



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

type IsVoid is IntList & !AnyList
```

- **Q)** What to do with this beast? *Coinductive Rewriting?*

Conclusion

- Complex type system arising from **flow typing** (+ elsewhere)
- **Declarative** Implementation in WyRL (some issues)
- Empirical evaluation is **encouraging**
- Have **Rascal** implementation (resolves issues)
- Recursive types ... ?

`https://github.com/DavePearce/RewritingTypeSystem`

http://whiley.org

@WhileyDave
http://github.com/DavePearce/Whiley