Sound and Complete Flow Typing with Unions, Intersections and Negations

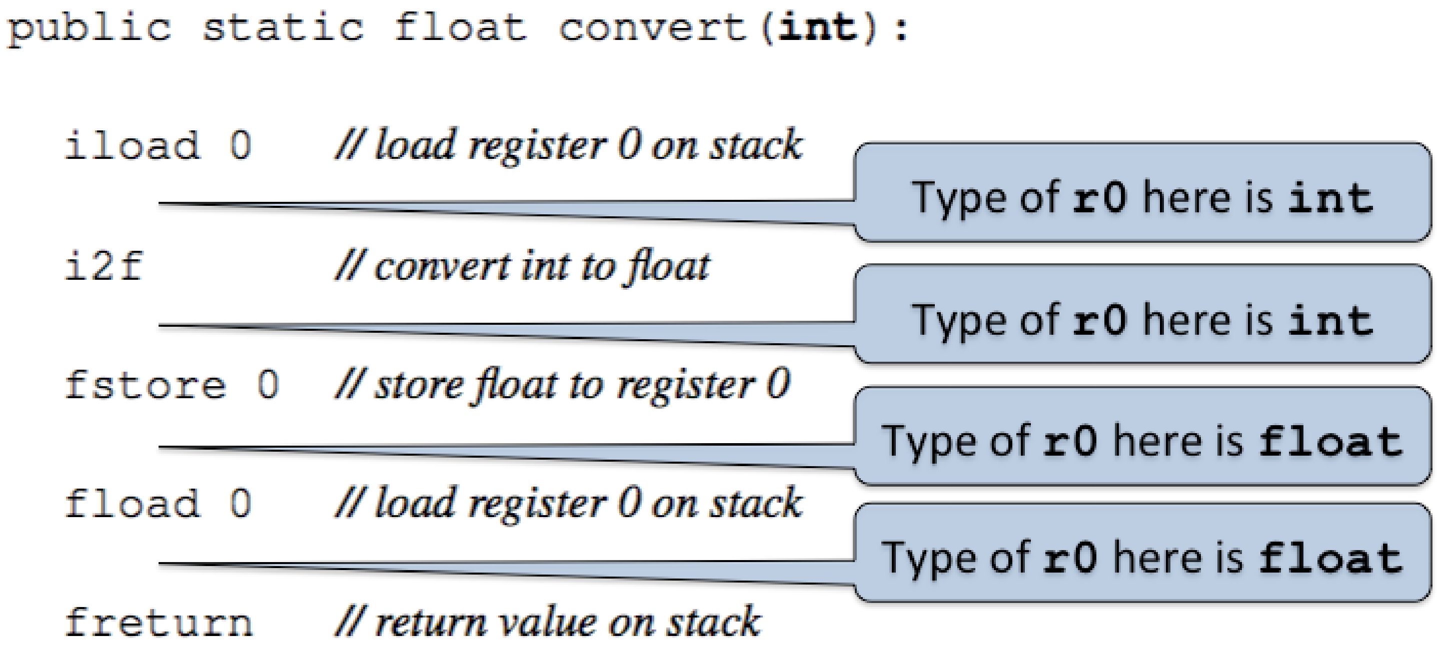
David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Flow Typing?

- **Defining characteristic**: *ability to retype variables*

- JVM Bytecode provides **widely-used** example:



```
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
```

Type of `r0` here is `int`

Type of `r0` here is `int`

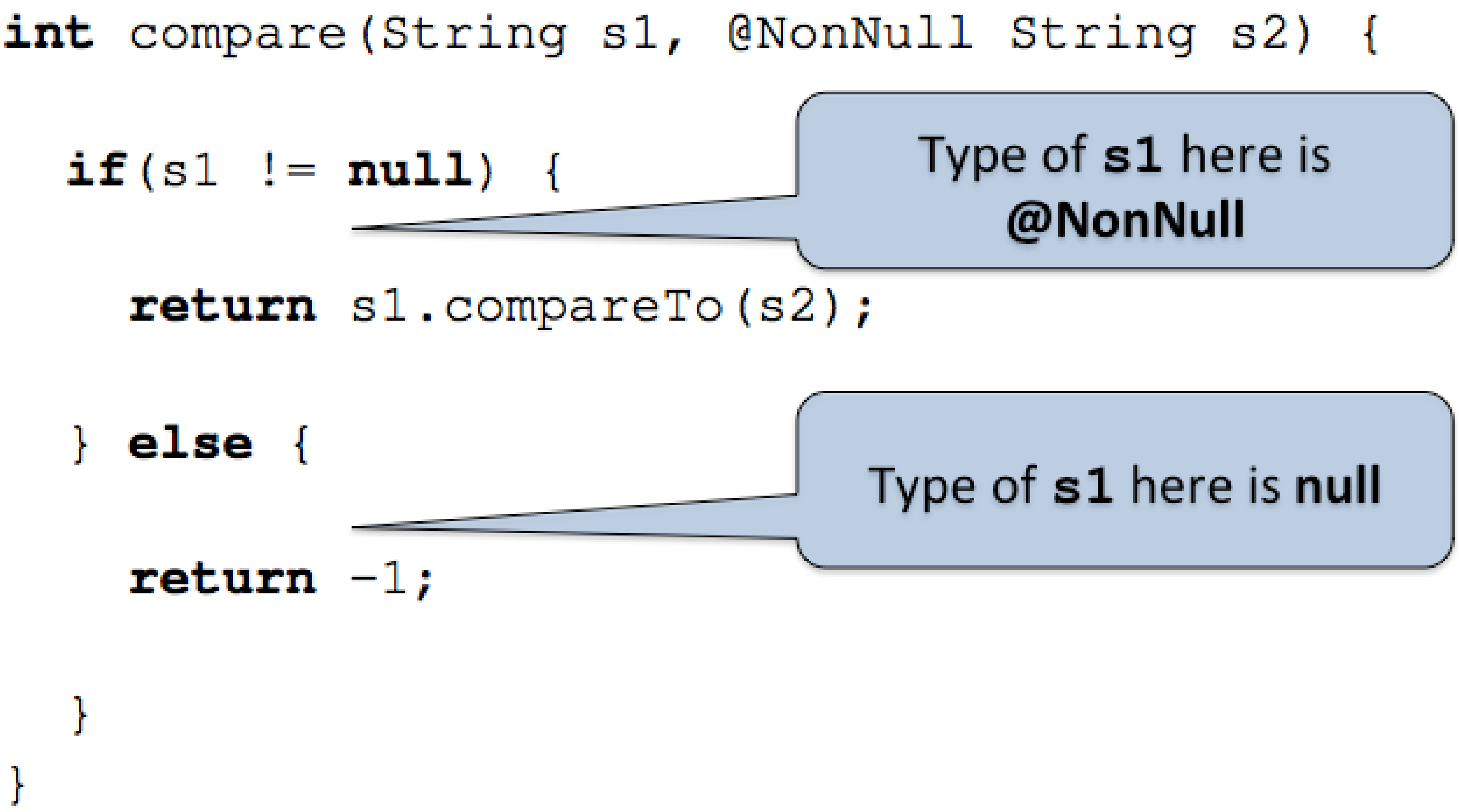
Type of `r0` here is `float`

Type of `r0` here is `float`

- Groovy 2.0 now includes flow-typing static checker

Another Example

- Non-null type checking provides another example:



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

Type of **s1** here is **@NonNull**

Type of **s1** here is **null**

- Many works in literature on this topic!

The Whiley Programming Language

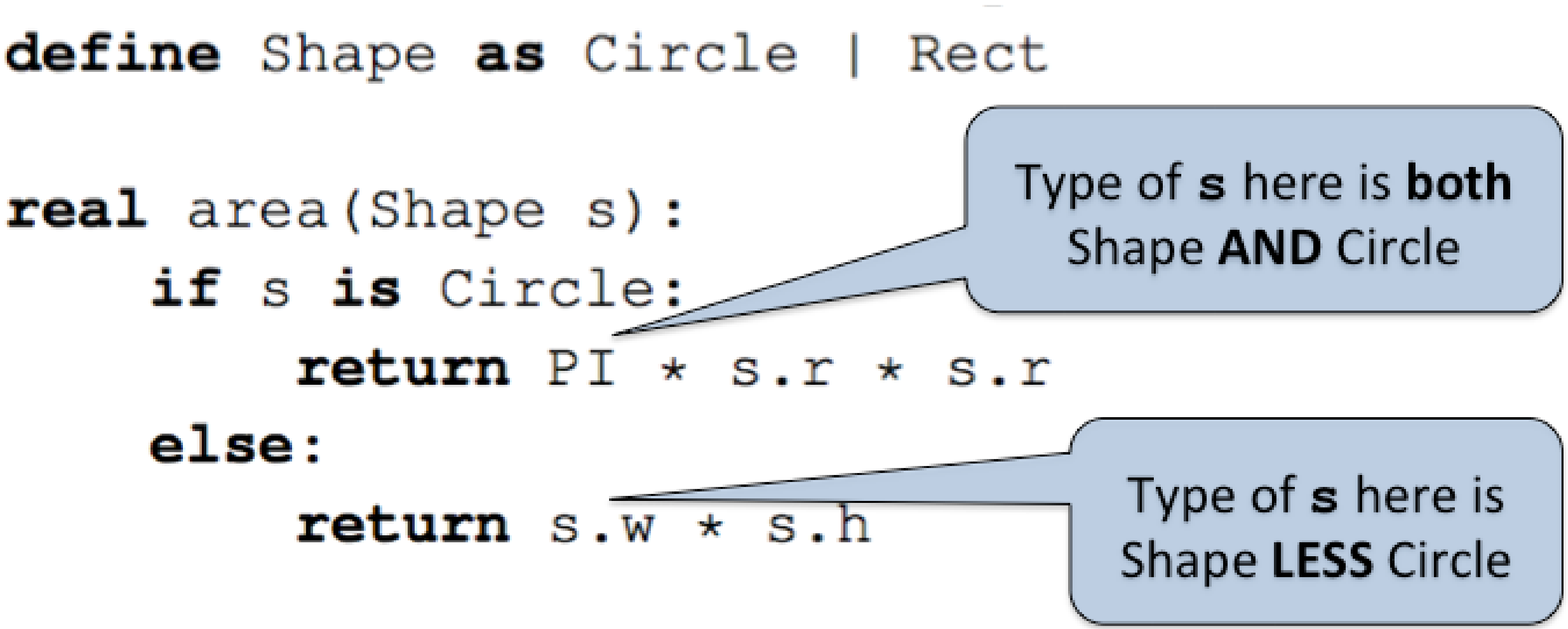
- Statically typed using a flow-type algorithm
- Look-and-feel of **dynamically-typed** language:

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

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

- **Question**: *how to implement flow-type checker?*

Intersection and Negation Types



```
define Shape as Circle | Rect

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

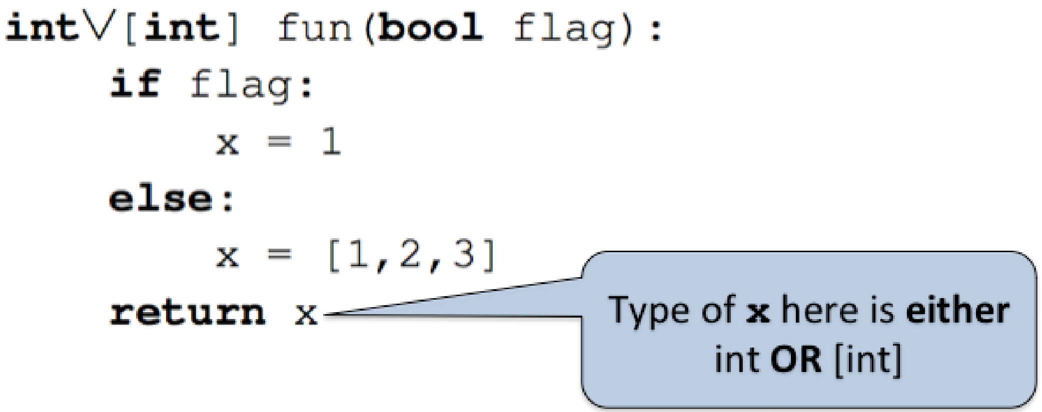
Type of **s** here is **both** Shape **AND** Circle

Type of **s** here is Shape **LESS** Circle

- **True Branch:** type of $\texttt{s}$ is $\texttt{Shape} \wedge \texttt{Circle} = \texttt{Circle}$
- **False Branch:** type of $\texttt{s}$ is $\texttt{Shape} - \texttt{Circle} = \texttt{Rect}$
- **NOTE:** can write $\texttt{T}_1 - \texttt{T}_2$ as $\texttt{T}_1 \wedge \neg\texttt{T}_2$

Union Types

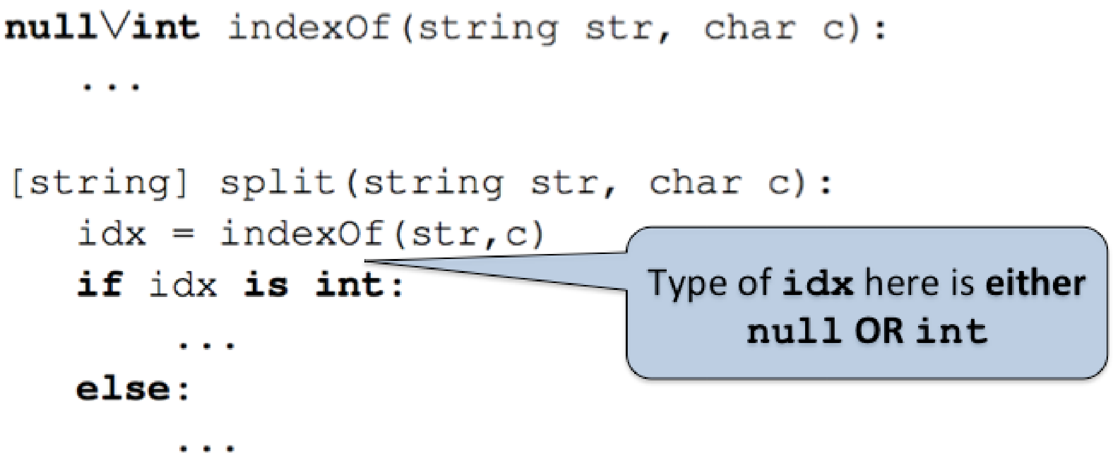
- Unions capture types of variables are **meet points**:



```
int∨[int] fun(bool flag):
    if flag:
        x = 1
    else:
        x = [1,2,3]
    return x
```

Type of **x** here is **either** int **OR** [int]

- Unions are useful for avoiding e.g. **null dereferences**:



```
null∨int indexOf(string str, char c):
   ...

[string] split(string str, char c):
   idx = indexOf(str,c)
   if idx is int:
       ...
   else:
       ...
```

Type of **idx** here is **either** **null OR int**

Syntax of Types

- A **syntactic** definition of types being considered:

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

- Made some **simplifying assumptions**:
  - Intersections and Unions are **unordered** (e.g. $\texttt{T}_1 \vee \texttt{T}_2$ is *syntactically identical* $\texttt{T}_2 \vee \texttt{T}_1$)
  - Duplicates are **removed** from Intersections and Unions (e.g. $\texttt{T}_1 \vee \texttt{T}_1$ is *syntactically identical* to $\texttt{T}_1$)
  - Will often write `void` as **short-hand** for $\neg\texttt{any}$

- **Note:** above defines a subset of types in Whiley

Semantics of Types

- A **semantic** definition of types being considered:

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

- Some **equivalences** between types are implied:

$$[\![\texttt{int} \vee \neg \texttt{int}]\!] = [\![\texttt{any}]\!]$$

$$[\![(\texttt{T}_1 \vee \texttt{T}_2, \texttt{int})]\!] = [\![(\texttt{T}_1, \texttt{int}) \vee (\texttt{T}_2, \texttt{int})]\!]$$

- Such types are *syntactically distinct*, but *semantically identical*

Soundness and Completeness

Definition (Subtype Soundness)

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

Definition (Subtype Completeness)

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

- Any **complete** subtyping algorithm must cope with equivalent types

  For example, must be able to show that $\mathtt{int} \wedge \neg\mathtt{int} \leq (\mathtt{int}, \mathtt{int})$

- *But, do we need completeness?*

Subtype Rules (Sound, but not Complete)

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

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

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

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

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

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

- **Comparable to**: S.Tobin-Hochstadt and M.Felleisen. The design and implementation of typed Scheme. In *Proceedings of POPL*, 2008.

Towards a Sound & Complete Subtype Algorithm...

- Developing a complete algorithm is challenging!
- Tried many **modifications** on previous rules ... without success
- Equivalences between types are the **main difficulty**
- Problem previously shown as **decidable**:

  A.Frisch,G.Castagna and V.Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 2008.

- But, this does not present **easily implementable** algorithm...

Atoms

- Let $\mathtt{T}^*$ denote a **type atom**, defined as follows:

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

- Atoms are **canonical by construction**

- **Sound and complete** subtyping for atoms is straightforward:

$$\frac{}{\mathtt{T}^+ \leq \mathtt{T}^+} \qquad \frac{}{\mathtt{T}^+ \leq \mathtt{any}}$$

$$\frac{\forall \mathtt{i} \in \{1, \ldots, \mathtt{n}\}. \mathtt{T}_\mathtt{i}^+ \leq \mathtt{S}_\mathtt{i}^+}{(\mathtt{T}_1^+ \ \ldots, \mathtt{T}_\mathtt{n}^+) \leq (\mathtt{S}_1^+, \ldots, \mathtt{S}_\mathtt{n}^+)}$$

- Above can be **extended** to negative atoms as well

Disjunctive Normal Form

- 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 $\mathtt{T}'$.

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

- Examples:

$$\neg(\mathtt{T}_1 \wedge \mathtt{T}_2) \Longrightarrow \neg\mathtt{T}_1 \vee \neg\mathtt{T}_2$$

$$\mathtt{T}_1 \wedge (\mathtt{T}_2 \vee \mathtt{T}_3) \Longrightarrow (\mathtt{T}_1 \wedge \mathtt{T}_2) \vee (\mathtt{T}_1 \wedge \mathtt{T}_3)$$

$$(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any}) \Longrightarrow (\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$$

Canonical Conjuncts

Definition (Canonical Conjunct)

Let $\mathtt{T}^{\wedge}$ denote a *canonical conjunct*. Then, $\mathtt{T}^{\wedge}$ is a type of the form $\mathtt{T}_1^+ \wedge \neg\mathtt{T}_2^+ \wedge \ldots \wedge \neg\mathtt{T}_n^+$ where:

1. For every negation $\neg\mathtt{T}_k^+$, we have $\mathtt{T}_1^+ \neq \mathtt{T}_k^+$ and $\mathtt{T}_1^+ \geq \mathtt{T}_k^+$.
2. For any two distinct negations $\neg\mathtt{T}_k^+$ and $\neg\mathtt{T}_m^+$, we have $\mathtt{T}_k^+ \not\geq \mathtt{T}_m^+$.

- **Key Property**: Let $\mathtt{T}_1$ and $\mathtt{T}_2$ be canonical conjuncts, then $[\![\mathtt{T}_1]\!] = [\![\mathtt{T}_2]\!] \implies \mathtt{T}_1 = \mathtt{T}_2$

- **Key Observation**: any conjunct of atoms can be expressed as a canonical conjunct:

$$\neg(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any}) \implies (\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any}) \wedge \neg((\mathtt{int}, \mathtt{int}), \mathtt{any})$$

$$(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{any}) \implies \mathtt{void}$$

$$(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{int}) \implies (\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{any}, \mathtt{int})$$

Canonicalised Disjunctive Normal Form

Definition (DNF$^+$)

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

- **Key Property:** $[\![\mathtt{T}]\!] = \emptyset \iff \mathtt{DNF}^+(\mathtt{T}) = \mathtt{void}$

- **Examples:**

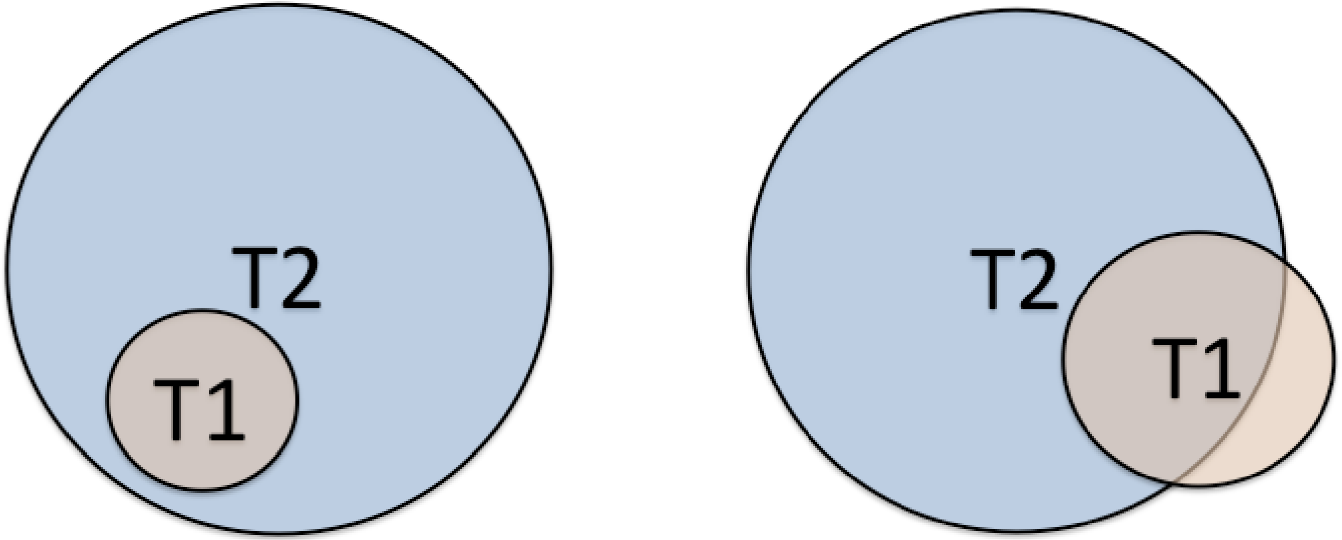
$$\mathtt{DNF}^+\big(\neg(\mathtt{int} \wedge (\mathtt{int},\mathtt{int}))\big) = (\mathtt{any} \wedge \neg\mathtt{int}) \vee (\mathtt{any} \wedge \neg(\mathtt{int},\mathtt{int}))$$

$$\mathtt{DNF}^+\big((\mathtt{int},\mathtt{any}) \wedge (\mathtt{int} \vee \neg(\mathtt{any},\mathtt{any}))\big) = \mathtt{void} \vee \mathtt{void}$$

- **Note:** $\mathtt{DNF}^+(\mathtt{T})$ is *not* a canonical form of $\mathtt{T}$

A Sound & Complete Subtype Algorithm

- **Surprise:** can encode subtype tests as types!



Definition (Subtyping)

Let $\mathtt{T}_1$ and $\mathtt{T}_2$ be types. Then, $\mathtt{T}_1 \leq \mathtt{T}_2$ is defined as $\mathtt{DNF}^+(\mathtt{T}_1 \wedge \neg \mathtt{T}_2) = \mathtt{void}$.

Conclusions

- Sound & Complete Subtyping over Unions, Intersections and Negations is **challenging**!
- Several sound (but not complete) algorithms have been presented
- Frisch *et al.* showed completeness was **decidable**
- We now have an **easily implementable** algorithm!
- *So ... does a polynomial time algorithm exist?*

http://whiley.org