Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc.

Joe Wells

Heriot-Watt University

Sébastien Carlier and Christian Haack helped with these overheads.
Overview.

- Basic concepts of types.
- Type polymorphism.
- Compositionality and principality.
- Case study: Type error slicing made possible by compositionality.
- Case study: Getting principal typings in the $\lambda$-calculus with polymorphism.
- Conclusion.
What are types?
What are types?

Types are used as *predicates* or *specifications* connected with some semantics.
What are types?

Types are used as *predicates* or *specifications* connected with some semantics.

Types are *usually* syntactic names of some of the conceivable predicates/specifications.
What are types?

- Types are used as *predicates* or *specifications* connected with some semantics.

- Types are *usually* syntactic names of some of the conceivable predicates/specifications.

- *Usually*, rules of a *type system* associate types with terms that satisfy or refine them.
What are types?

Types are used as *predicates* or *specifications* connected with some semantics.

Types are *usually* syntactic names of some of the conceivable predicates/specifications.

*Usually*, rules of a *type system* associate types with terms that satisfy or refine them.

Non-characterizers of types:
What are types?

- Types are used as *predicates* or *specifications* connected with some semantics.

- Types are *usually* syntactic names of some of the conceivable predicates/specifications.

- *Usually*, rules of a *type system* associate types with terms that satisfy or refine them.

- Non-characterizers of types:
  - Types can be used for *description* or *prescription*.
What are types?

- Types are used as *predicates* or *specifications* connected with some semantics.

- Types are *usually* syntactic names of some of the conceivable predicates/specifications.

- *Usually*, rules of a *type system* associate types with terms that satisfy or refine them.

- Non-characterizers of types:
  - Types can be used for *description* or *prescription*.
  - Types may be intended for reading by *humans* or *computers*.
What are types?

- Types are used as *predicates* or *specifications* connected with some semantics.

- Types are *usually* syntactic names of some of the conceivable predicates/specifications.

- *Usually*, rules of a *type system* associate types with terms that satisfy or refine them.

- Non-characterizers of types:
  - Types can be used for *description* or *prescription*.
  - Types may be intended for reading by *humans* or *computers*.
  - Types may be *easy* or *hard* to determine.
Why find types automatically?

Without type inference, explicit types might be needed at any program point. In the case of higher-order programming, they would get big, and it would be too tedious to type them.
Why find types automatically?

Without type inference, explicit types might be needed at any program point. In the case of higher-order programming, they would get big, and it would be too tedious to type them.

For programming flexibility, it is best to automatically calculate optimal types, because programmers might write type information that is not “most general”, preventing typable programs from being accepted and/or making modules reusable in fewer combinations.
Why find types automatically?

- Without type inference, explicit types might be needed at any program point. In the case of higher-order programming, they would get big, and it would be too tedious to type them.

- For programming flexibility, it is best to automatically calculate optimal types, because programmers might write type information that is not “most general”, preventing typable programs from being accepted and/or making modules reusable in fewer combinations.

- Programming language type systems are getting more and more complex (e.g., Cyclone, a “safe C”) and it is getting harder for programmers to supply the types.
An example program.

This Standard ML (SML) program:

```ml
fun twice f x = f (f x);
fun id z = z;
twice (twice id);
```
An example program.

This Standard ML (SML) program:

```ml
  fun twice f x = f (f x);
  fun id z = z;
  twice (twice id);
```

is the same as this λ-term:

```
  (@ t) (λf. λx. f (f x)) = (λt. (λi. t (t i)) (λy. y)) (λf. λx. f (f x))
```

Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc. – p.5/46
Example: Types.

Our example analyzed using the simply typed $\lambda$-calculus:

$$\lambda t \ ((o \rightarrow o) \rightarrow (o \rightarrow o)) \rightarrow (o \rightarrow o)$$

$$\lambda f \ (o \rightarrow o) \rightarrow (o \rightarrow o)$$

$$\lambda i \ (o \rightarrow o) \rightarrow (o \rightarrow o)$$

$$\lambda z \ o \rightarrow o$$

$$t \ (o \rightarrow o) \rightarrow (o \rightarrow o)$$

$$i \ o \rightarrow o$$

$$x \ o$$

Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc. – p.6/46
Example: Flow.

Our example analyzed using 0CFA [Shivers, 1991]:

Explaining Concepts in Compositional Type-Based Program Analysis:Principality, Intersection Types, Expansion, etc. – p.7/46
Type analysis is flow analysis.

Illustrating how the type and flow analyses are intertwined:
Overview.

- Basic concepts of types.
- **Type polymorphism.**
- Compositionality and principality.
- Case study: Type error slicing made possible by compositionality.
- Case study: Getting principal typings in the $\lambda$-calculus with polymorphism.
- Conclusion.
What is type polymorphism?

An important feature mitigating type system inflexibility is *type polymorphism*, which:
What is type polymorphism?

An important feature mitigating type system inflexibility is *type polymorphism*, which:

- Allows a program fragment to be viewed in different ways, depending on where its output is used or where its inputs come from.
What is type polymorphism?

An important feature mitigating type system inflexibility is *type polymorphism*, which:

- Allows a program fragment to be viewed in different ways, depending on where its output is used or where its inputs come from.

- Is *essential* for code reuse [Reynolds, 1974] and abstract data types [Mitchell and Plotkin, 1988].
What is type polymorphism?

An important feature mitigating type system inflexibility is *type polymorphism*, which:

- Allows a program fragment to be viewed in different ways, depending on where its output is used or where its inputs come from.

- Is *essential* for code reuse [Reynolds, 1974] and abstract data types [Mitchell and Plotkin, 1988].

- Is traditionally treated formally using “for all” (\(\forall\)) quantifiers [Girard, 1972] or “there exists” (\(\exists\)) quantifiers and/or by a notion of subtyping (\(T_1 \leq T_2\)).
Example: “for all” quantifiers.

\[ \forall a, b. (a \times b) \rightarrow (b \times a) = (fn (x^a, y^b) \Rightarrow (y^b, x^a)) \]

```plaintext
val swap : \forall a, b. (int \times bool) \rightarrow (bool \times int) = (fn (x^a, y^b) \Rightarrow (y^b, x^a));

val pair1 : int \times bool = (1, true);

val pair2 : real \times real = (2.7, 9.9);

(val swap : int \times bool \rightarrow bool \times int) pair1,

(val swap : real \times real \rightarrow real \times real) pair2;
```
Example: “for all” quantifiers.

\[ \forall a, b. (a \times b) \rightarrow (b \times a) = (fn(x^a, y^b) \Rightarrow (y^b, x^a)) \]

val swap \(\forall a, b. (a \times b) \rightarrow (b \times a) = (fn(x^a, y^b) \Rightarrow (y^b, x^a))\); 

val pair1 \(\text{int} \times \text{bool} = (1, \text{true})\); 

val pair2 \(\text{real} \times \text{real} = (2.7, 9.9)\); 

\((\text{swap}(\text{int} \times \text{bool}) \rightarrow (\text{bool} \times \text{int}))\) pair1, 

\((\text{swap}(\text{real} \times \text{real}) \rightarrow (\text{real} \times \text{real}))\) pair2;

Implicit typing discovers the types automatically [Damas and Milner, 1982] (at least for the above example).
Example: “for all” quantifiers.

\[ \forall a, b. (a \times b) \rightarrow (b \times a) = (\text{fn } (x^a, y^b) \Rightarrow (y^b, x^a)); \]

val swap \( ^{\text{int} \times \text{bool}} \) = (1, true);

val pair1 \( ^{\text{real} \times \text{real}} \) = (2.7, 9.9);

(val swap \( ^{\text{int} \times \text{bool}} \rightarrow \text{bool} \times \text{int} \) pair1,

to swap \( ^{\text{real} \times \text{real}} \rightarrow \text{real} \times \text{real} \) pair2);

- Implicit typing discovers the types automatically [Damas and Milner, 1982] (at least for the above example).

- In body of polymorphic function, the usage types are hidden behind type variables.
Example: “there exists” quantifiers.

val closure1 \(\text{int} \times (\text{int} \rightarrow \text{bool})\) = (5, \((\text{fn } x \Rightarrow x > 1)\));

val closure2 \(\text{bool} \times (\text{bool} \rightarrow \text{bool})\) = (true, \((\text{fn } x \Rightarrow \neg x)\));

val closure = if b then closure1 \(\exists a. a \times (a \rightarrow \text{bool})\)
else closure2 \(\exists a. a \times (a \rightarrow \text{bool})\);

val result \(\text{bool}\) = (\#2 closure)\(a \rightarrow \text{bool}\)(\#1 closure)\(a\);
Example: “there exists” quantifiers.

val closure1 \:\texttt{int} \times (\texttt{int} \to \texttt{bool}) = (5, (\texttt{fn x} \Rightarrow x > 1));

val closure2 \:\texttt{bool} \times (\texttt{bool} \to \texttt{bool}) = (\texttt{true}, (\texttt{fn x} \Rightarrow \texttt{not x}));

val closure = if \: b \: then \: closure1 \: \exists \: a . \: a \times (a \to \texttt{bool})
\:
else closure2 \: \exists \: a . \: a \times (a \to \texttt{bool});

val result \:\texttt{bool} = (\#2 \: \texttt{closure})^{a \to \texttt{bool}}(\#1 \: \texttt{closure})^a;

- Dual of universal quantifier.
Example: “there exists” quantifiers.

\[
\begin{align*}
\text{val } \text{closure}_1 &: \mathbb{N} \times (\mathbb{N} \to \mathbb{B}) = (5, (\lambda x . x > 1)) ; \\
\text{val } \text{closure}_2 &: \mathbb{B} \times (\mathbb{B} \to \mathbb{B}) = (\text{true}, (\lambda x . \neg x)) ; \\
\text{val } \text{closure} &= \text{if } b \text{ then } \text{closure}_1 \text{ else } \text{closure}_2 \\
\text{val result} &: \mathbb{B} = (\#2 \text{ closure})^{\mathbb{N} \to \mathbb{B}} (\#1 \text{ closure})^{\mathbb{N}} ;
\end{align*}
\]

- Dual of universal quantifier.

- Usage site does not know source types.
Polymorphism via intersection types.

Type polymorphism by *listing* usage types [Coppo, Dezani-Ciancaglini, and Venneri, 1980].
Polymorphism via intersection types.

- Type polymorphism by *listing* usage types [Coppo, Dezani-Ciancaglini, and Venneri, 1980].

- Example comparing $\forall$-quantified and intersection types:

  $\forall$-quantified types: $(\text{fn } x \Rightarrow x)^{\forall a. (a \rightarrow a)}$

  intersection types: $(\text{fn } x \Rightarrow x)^{(\text{int} \rightarrow \text{int}) \cap (\text{real} \rightarrow \text{real})}$

Example is semantically like $\forall a \in \{\text{int, real}\}. a \rightarrow a$, but the typing rules have significant practical differences.
Polymorphism via intersection types.

- Type polymorphism by *listing* usage types [Coppo, Dezani-Ciancaglini, and Venneri, 1980].

- Example comparing $\forall$-quantified and intersection types:

  - $\forall$-quantified types: $(\text{fn } x \Rightarrow x)^{\forall a.(a \rightarrow a)}$
  
  - intersection types: $(\text{fn } x \Rightarrow x)^{(\text{int} \rightarrow \text{int}) \cap (\text{real} \rightarrow \text{real})}$

  Example is semantically like $\forall a \in \{\text{int, real}\}.a \rightarrow a$, but the typing rules have significant practical differences.

- Named “intersection types” because in traditional model theory, semantic denotations $\llbracket T_1 \rrbracket$ and $\llbracket T_2 \rrbracket$ are program fragment sets and $\llbracket T_1 \cap T_2 \rrbracket = \llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket$ (usually).
Example: Intersection types.

\[
\text{val swap} \left( (\text{int} \times \text{bool}) \to (\text{bool} \times \text{int}) \right) \cap (\text{real} \times \text{real}) \to (\text{real} \times \text{real}) = (\text{fn} (x \text{real}, y \text{real}) \Rightarrow (y \text{real}, x \text{real}));
\]

\[
\text{val pair1}^{\text{int} \times \text{bool}} = (1, \text{true});
\]

\[
\text{val pair2}^{\text{real} \times \text{real}} = (2.7, 9.9);
\]

\[
\text{swap}^{(\text{int} \times \text{bool}) \to (\text{bool} \times \text{int})} \text{pair1}, \quad \text{swap}^{(\text{real} \times \text{real}) \to (\text{real} \times \text{real})} \text{pair2};
\]
Example: Intersection types.

\[
\text{val swap} (\cap (\text{int} \times \text{bool}) \to (\text{bool} \times \text{int})) \\
= (\text{fn} (x : \text{int}, y : \text{real}) \to (y : \text{real}, x : \text{real}));
\]

\[
\text{val pair1}^{\text{int} \times \text{bool}} = (1, \text{true});
\]

\[
\text{val pair2}^{\text{real} \times \text{real}} = (2.7, 9.9);
\]

\[
(\text{swap}^{\text{int} \times \text{bool}} \to (\text{bool} \times \text{int})) \text{ pair1},
\]

\[
\text{swap}^{\text{real} \times \text{real}} \to (\text{real} \times \text{real}) \text{ pair2};
\]

All types can be discovered automatically [van Bakel, 1993; Jim, 1996; Kfoury and Wells, 1999]. (Also Ronchi Della Rocca [1988].)
Example: Intersection types.

\[
\text{val swap} \quad (\text{int} \times \text{bool}) \rightarrow (\text{bool} \times \text{int}) \quad \cap \\
(\text{real} \times \text{real}) \rightarrow (\text{real} \times \text{real})
\]

\[
\begin{align*}
\text{val pair1} & \quad \text{int} \times \text{bool} = (1, \text{true}); \\
\text{val pair2} & \quad \text{real} \times \text{real} = (2.7, 9.9);
\end{align*}
\]

\[
\begin{align*}
\text{swap} & \quad (\text{int} \times \text{bool}) \rightarrow (\text{bool} \times \text{int}) \quad \text{pair1}, \\
\text{swap} & \quad (\text{real} \times \text{real}) \rightarrow (\text{real} \times \text{real}) \quad \text{pair2};
\end{align*}
\]

- All types can be discovered automatically [van Bakel, 1993; Jim, 1996; Kfoury and Wells, 1999]. (Also Ronchi Della Rocca [1988].)

- Exposes usage types throughout.
Example: Union types.

\[
\begin{align*}
\text{val closure1} & \quad \text{int} \times (\text{int} \rightarrow \text{bool}) = (5, (\text{fn } x \Rightarrow x > 1)); \\
\text{val closure2} & \quad \text{bool} \times (\text{bool} \rightarrow \text{bool}) = (\text{true}, (\text{fn } x \Rightarrow \text{not } x)); \\
\text{val closure} & \quad = \text{if } b \text{ then closure1} (\bigcup \text{bool} \times (\text{bool} \rightarrow \text{bool})) \\
& \quad \quad \quad \text{else closure2} (\bigcup \text{bool} \times (\text{bool} \rightarrow \text{bool})); \\
\text{val result} & \quad = (\#2 \text{ closure}) \text{bool} \rightarrow \text{bool} (\#1 \text{ closure}) \text{bool}.
\end{align*}
\]
Example: Union types.

val closure1 \text{int} \times (\text{int} \to \text{bool}) = (5, (\text{fn x} \Rightarrow x > 1));

val closure2 \text{bool} \times (\text{bool} \to \text{bool}) = (\text{true}, (\text{fn x} \Rightarrow \neg x));

val closure = \text{if b then closure1} (\bigcup \text{int} \times (\text{int} \to \text{bool}) \bigcup \text{bool} \times (\text{bool} \to \text{bool}))
\text{else closure2} (\bigcup \text{int} \times (\text{int} \to \text{bool}) \bigcup \text{bool} \times (\text{bool} \to \text{bool}));

val result \text{bool} = (#2 closure) \text{int} \to \text{bool} \to \text{bool} (#1 closure) \text{bool};

\textbf{Dual of intersection types.}
Example: Union types.

\[
\text{val closure}_1^{\text{int} \times (\text{int} \to \text{bool})} = (5, (\text{fn } x \Rightarrow x > 1));
\]

\[
\text{val closure}_2^{\text{bool} \times (\text{bool} \to \text{bool})} = (\text{true}, (\text{fn } x \Rightarrow \text{not } x));
\]

\[
\text{val closure} = \text{if } b \text{ then closure}_1 \left( \bigcup \text{int} \times (\text{int} \to \text{bool}) \right) \text{ else closure}_2 \left( \bigcup \text{bool} \times (\text{bool} \to \text{bool}) \right);
\]

\[
\text{val result}^{\text{bool} \to \text{bool} \to \text{bool}} = (\#2 \text{ closure})^{\text{int} \to \text{bool}} (\#1 \text{ closure})^{\text{int} \to \text{bool}}.
\]

- Dual of intersection types.
- Exposes usage types throughout.
What is the rank of polymorphism?

Rank is generally relative to some polymorphic type constructor \( C \), e.g., \( \cap \) or \( \forall \). Rank counts the number of “\( \rightarrow \)” occurrences an occurrence of \( C \) is inside the left argument of [Leivant, 1983].
What is the rank of polymorphism?

*Rank* is generally relative to some polymorphic type constructor $C$, e.g., $\cap$ or $\forall$. Rank counts the number of “→” occurrences an occurrence of $C$ is inside the left argument of [Leivant, 1983].

Examples:

<table>
<thead>
<tr>
<th>Type</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \rightarrow b$</td>
<td>0</td>
</tr>
<tr>
<td>$a \rightarrow (b \cap c \rightarrow b)$</td>
<td>1</td>
</tr>
<tr>
<td>$a \rightarrow (b \rightarrow (a \rightarrow e))$</td>
<td>2</td>
</tr>
<tr>
<td>$a \rightarrow (b \rightarrow d \cap a \rightarrow e)$</td>
<td>3</td>
</tr>
</tbody>
</table>
What is the rank of polymorphism?

- **Rank** is generally relative to some polymorphic type constructor $C$, e.g., $\cap$ or $\forall$. Rank counts the number of “$\rightarrow$” occurrences an occurrence of $C$ is inside the left argument of [Leivant, 1983].

- **Examples:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \rightarrow b$</td>
<td>0</td>
</tr>
<tr>
<td>$a \rightarrow (b \cap c \rightarrow b)$</td>
<td>1</td>
</tr>
<tr>
<td>$d \rightarrow (\forall a \rightarrow e)$</td>
<td>2</td>
</tr>
<tr>
<td>$a \rightarrow b \cap d \rightarrow e$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Rank-$k$ bounds how far into the future evaluation a type system can look in making distinctions when predicting behavior. The rank-$k$ restrictions of intersection types are decidable.
Typing power of intersection types.

F: System F.
\( \Lambda_k \): rank-\( k \) System F.
\( \cap \): intersection types.
\( \cap_k \): rank-\( k \) of \( \cap \).
Decidable.
Undecidable.

(Decision procedure complexity now known [Kfoury, Mairson, Turbak, and Wells, 1999].)
Flexibility of intersection types.

\[
M = \begin{pmatrix}
\text{fun self\_apply2 } z \Rightarrow (z \ z) \ z; \\
\text{fun apply } f \ x \Rightarrow f \ x; \\
\text{fun reverse\_apply } y \ g \Rightarrow g \ y; \\
\text{fun id } w \Rightarrow w; \\
\text{(self\_apply2 } \text{apply } \text{not } \text{true,} \\
\text{self\_apply2 } \text{reverse\_apply } \text{id } \text{false } \text{not});
\end{pmatrix}
\]

Program fragment \( M \) \textit{safely} computes \((\text{false, true})\).

Urzyczyn [1997] proved that \( M \) is not typable in \( F_\omega \), and \( F_\omega \) is the most powerful type system with “for all” quantifiers [Giannini, Honsell, and Ronchi Della Rocca, 1993].

\( M \) needs only rank-3 intersection types.
Program analysis with intersection types.

Intersection type systems have been developed for many kinds of program analysis aimed at justifying compiler optimizations to produce better machine code.
Program analysis with intersection types.

Intersection type systems have been developed for many kinds of program analysis aimed at justifying compiler optimizations to produce better machine code.

Flow [Banerjee, 1997].
Program analysis with intersection types.

Intersection type systems have been developed for many kinds of program analysis aimed at justifying compiler optimizations to produce better machine code.

- **Flow** [Banerjee, 1997].
- **Dead code** [Damiani and Giannini, 2000; Damiani, 2003].
Program analysis with intersection types.

Intersection type systems have been developed for many kinds of program analysis aimed at justifying compiler optimizations to produce better machine code.

- **Flow** [Banerjee, 1997].
- **Dead code** [Damiani and Giannini, 2000; Damiani, 2003].
- **Strictness** [Solberg et al., 1994; Jensen, 1998].
Program analysis with intersection types.

Intersection type systems have been developed for many kinds of program analysis aimed at justifying compiler optimizations to produce better machine code.

- **Flow** [Banerjee, 1997].
- **Dead code** [Damiani and Giannini, 2000; Damiani, 2003].
- **Strictness** [Solberg et al., 1994; Jensen, 1998].
- **Totality** [Solberg et al., 1994; Coppo et al., 2002].
Program analysis with intersection types.

Intersection type systems have been developed for many kinds of program analysis aimed at justifying compiler optimizations to produce better machine code.

- **Flow** [Banerjee, 1997].
- **Dead code** [Damiani and Giannini, 2000; Damiani, 2003].
- **Strictness** [Solberg et al., 1994; Jensen, 1998].
- **Totality** [Solberg et al., 1994; Coppo et al., 2002].

Intersection types seem to have the potential to be a general, flexible framework for many program analyses.
Overview.

- Basic concepts of types.
- Type polymorphism.
- **Compositionality and principality.**
- Case study: Type error slicing made possible by compositionality.
- Case study: Getting principal typings in the $\lambda$-calculus with polymorphism.
- Conclusion.
What is compositional analysis?

Compositional analysis means it always holds that the parts can be analyzed independently and the analysis results can be composed without reinspecting the parts.
What is compositional analysis?

- **Compositional analysis** means it always holds that the parts can be analyzed independently and the analysis results can be composed *without reinspecte*ng *the parts*.

- Compositional analysis results are always the best information for any possible usage context. If a part is unchanged and its analysis result is available, reanalyzing it can not help. Only new combinations need to be checked.
What is compositional analysis?

*Compositional analysis* means it always holds that the parts can be analyzed independently and the analysis results can be composed *without reinspecting the parts*.

Compositional analysis results are always the best information for any possible usage context. If a part is unchanged and its analysis result is available, reanalyzing it can not help. Only new combinations need to be checked.

Compositional analysis is better for *dynamic*, *incremental*, and *modular* software assembly, but many type systems do not support compositional analysis.
Why compositional analysis?

- For efficiency, minimal analysis/compilation work needed on incremental changes. Old results for portions can be reliably reused.
Why compositional analysis?

- For efficiency, minimal analysis/compilation work needed on incremental changes. Old results for portions can be reliably reused.

- Reliability of incrementally modified systems. The analysis obtained by incremental changes (such as modifying one file and recompiling) should be identical to reanalyzing the entire system.
Why compositional analysis?

For efficiency, minimal analysis/compilation work needed on incremental changes. Old results for portions can be reliably reused.

Reliability of incrementally modified systems. The analysis obtained by incremental changes (such as modifying one file and recompiling) should be identical to reanalyzing the entire system.

Modern systems like Java and C♯ have broken the link needed by separate compilation between the compile-time and link-time environments, so it is better not to use any compile-time environment.
Why compositional analysis?

- For efficiency, minimal analysis/compilation work needed on incremental changes. Old results for portions can be reliably reused.

- Reliability of incrementally modified systems. The analysis obtained by incremental changes (such as modifying one file and recompiling) should be identical to reanalyzing the entire system.

- Modern systems like Java and C# have broken the link needed by separate compilation between the compile-time and link-time environments, so it is better not to use any compile-time environment.

- A network node without global knowledge can gradually learn more about other entities and predict possible failures as soon as sufficient information is available.
What is a typing?

A type system has *typing judgements* that assign *interesting properties* to *program fragments*. 
What is a typing?

A type system has *typing judgements* that assign *interesting properties* to *program fragments*.

Conventional typing judgements often look like this:

$$ A \vdash M : T $$
What is a typing?

A type system has *typing judgements* that assign *interesting properties* to *program fragments*.

Conventional typing judgements often look like this:

\[ A \vdash M : T \]

To encourage better thinking, we write this instead:

\[ M : \langle A \vdash T \rangle \]
What is a typing?

A type system has *typing judgements* that assign *interesting properties* to *program fragments*.

Conventional typing judgements often look like this:

\[ A \vdash M : T \]

To encourage better thinking, we write this instead:

\[ M : \langle A \vdash T \rangle \]
What is a typing?

A type system has *typing judgements* that assign *interesting properties* to *program fragments*.

Conventional typing judgements often look like this:

\[ A \vdash M : T \]

To encourage better thinking, we write this instead:

\[ M : \langle A \vdash T \rangle \]

A type system can thus be seen as a set of pairs of the form \((M : \Theta)\) where \(\Theta\) is usually of the form \(\langle A \vdash T \rangle\).
What is a principal typing?

Let $S$ be some type system.
What is a principal typing?

Let $S$ be some type system.

The statement $\Theta_1 \leq_S \Theta_2$ ("$\Theta_1$ is at least as strong as $\Theta_2$ in system $S$") means $M : \Theta_1$ implies $M : \Theta_2$ for every $M$. 
What is a principal typing?

Let $S$ be some type system.

The statement $\Theta_1 \leq_S \Theta_2$ ("$\Theta_1$ is at least as strong as $\Theta_2$ in system $S$") means $M : \Theta_1$ implies $M : \Theta_2$ for every $M$.

A typing $\Theta$ for term $M$ is principal exactly when $\Theta$ is at least as strong as all typings for $M$ [Wells, 2002].
**What is a principal typing?**

Let $S$ be some type system.

The statement $\Theta_1 \leq_S \Theta_2$ ("$\Theta_1$ is at least as strong as $\Theta_2$ in system $S$") means $M : \Theta_1$ implies $M : \Theta_2$ for every $M$.

A typing $\Theta$ for term $M$ is *principal* exactly when $\Theta$ is at least as strong as all typings for $M$ [Wells, 2002].

Do not confuse this with the *weaker* notion of "principal type" with fixed free variable type assumptions often mentioned for the Hindley/Milner (HM) type system (Haskell, OCaml, SML, etc.).
What is a principal typing?

Let $S$ be some type system.

The statement $\Theta_1 \leq_S \Theta_2$ ("$\Theta_1$ is at least as strong as $\Theta_2$ in system $S$") means $M : \Theta_1$ implies $M : \Theta_2$ for every $M$.

A typing $\Theta$ for term $M$ is principal exactly when $\Theta$ is at least as strong as all typings for $M$ [Wells, 2002].

Do not confuse this with the weaker notion of "principal type" with fixed free variable type assumptions often mentioned for the Hindley/Milner (HM) type system (Haskell, OCaml, SML, etc.).

Principal typings (PTs) allow compositional analysis.
What is a principal typing?

Let $S$ be some type system.

The statement $\Theta_1 \leq_S \Theta_2$ (“$\Theta_1$ is \textit{at least as strong as} $\Theta_2$ in system $S$”) means $M : \Theta_1$ implies $M : \Theta_2$ for every $M$.

A typing $\Theta$ for term $M$ is \textit{principal} exactly when $\Theta$ is at least as strong as all typings for $M$ [Wells, 2002].

Do not confuse this with the \textit{weaker} notion of “principal type” with fixed free variable type assumptions often mentioned for the Hindley/Milner (HM) type system (Haskell, OCaml, SML, etc.).

Principal typings (PTs) allow \textit{compositional} analysis.

Until Wells [2002], each system with PTs had its own definition via syntactic operations like \textit{substitution, subtyping, weakening}, etc.
Which systems have principal typings?

Many type systems with $\forall$-quantifiers (e.g., HM and System F) do not have PTs [Wells, 2002].
Which systems have principal typings?

Many type systems with $\forall$-quantifiers (e.g., HM and System F) do not have PTs [Wells, 2002].

The popular W algorithm [Damas and Milner, 1982] for HM is not compositional and compositional analysis for HM can not use HM typings for intermediate results.
Which systems have principal typings?

Many type systems with \( \forall \)-quantifiers (e.g., HM and System F) do not have PTs [Wells, 2002].

The popular W algorithm [Damas and Milner, 1982] for HM is not compositional and compositional analysis for HM can not use HM typings for intermediate results. Fortunately, a restricted rank-2 intersection type system [Damas, 1985] types the same terms and has PTs.
Which systems have principal typings?

Many type systems with ∀-quantifiers (e.g., HM and System F) do not have PTs [Wells, 2002].

The popular W algorithm [Damas and Milner, 1982] for HM is not compositional and compositional analysis for HM can not use HM typings for intermediate results. Fortunately, a restricted rank-2 intersection type system [Damas, 1985] types the same terms and has PTs.

Getting PTs usually needs types or type constraints that closely follow the language semantics. For the λ-calculus, adding intersection types can generally gain PTs (e.g., [Margaria and Zacchi, 1995]).
Implications of not having PTs.

For example, HM’s lack of principal typings means an HM analysis algorithm must do one of these:
Implications of not having PTs.

For example, HM’s lack of principal typings means an HM analysis algorithm must do one of these:

- Be incomplete (failing on some typable terms).
Implications of not having PTs.

For example, HM’s lack of principal typings means an HM analysis algorithm must do one of these:

- Be incomplete (failing on some typable terms).
- Be noncompositional (not strictly bottom-up). For example, the W algorithm [Damas and Milner, 1982] is noncompositional because for \((\text{let } x = M \text{ in } N)\) it first analyzes \(M\) and then uses the result in analyzing \(N\).
Implications of not having PTs.

For example, HM’s lack of principal typings means an HM analysis algorithm must do one of these:

- Be incomplete (failing on some typable terms).
- Be noncompositional (not strictly bottom-up). For example, the W algorithm [Damas and Milner, 1982] is noncompositional because for \((\text{let } x = M \text{ in } N)\) it first analyzes \(M\) and then uses the result in analyzing \(N\).
- Not use HM typings for intermediate results. E.g., the typing of \((xx)\) in the Chap. 1 system of Damas [1985]:

\[
\langle (x : a, x : a \rightarrow b) \vdash b \rangle
\]

This is essentially intersection types, i.e.:

\[
\langle (x : a \cap (a \rightarrow b)) \vdash b \rangle
\]

Essentially the same was done by Shao and Appel [1993] and Bernstein and Stark [1995].
Overview.

- Basic concepts of types.
- Type polymorphism.
- Compositionality and principality.

**Case study:** Type error slicing made possible by compositionality.

**Case study:** Getting principal typings in the λ-calculus with polymorphism.

Conclusion.
Case study: Type error slicing.

I now will show by examples a case study where doing an analysis compositionally made things much easier.
Case study: Type error slicing.

I now will show by examples a case study where doing an analysis compositionally made things much easier.

The system does *type error slicing* [Haack and Wells, 2004], which means it analyzes a untypable term and outputs a minimal untypable *slice* of the term to explain the type error.
Case study: Type error slicing.

I now will show by examples a case study where doing an analysis compositionally made things much easier.

The system does type error slicing [Haack and Wells, 2004], which means it analyzes a untypable term and outputs a minimal untypable slice of the term to explain the type error.

The system I will describe uses a type system that types the same terms as HM, but uses intersection types instead of “for all” quantifiers internally, so it is compositional. This made it much easier to generate and solve constraints.
Type error example.

```haskell
val average = fn weight => fn list =>
  let val iterator = fn (x,(sum,length)) =>
    (sum + weight x, length + 1)
  in
    foldl iterator (0,0) list
  end
in
  sum div length end

val find_best = fn weight => fn lists =>
  let
    val average = average weight
    val iterator = fn (list,(best,max)) =>
      let
        val avg_list = average list
        in
          if avg_list > max then
            (list,avg_list)
          else
            (best,max)
          end
      end
  in
    foldl iterator (nil,0) lists
  end
in
  best end

val find_best_simple = find_best 1
```

Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc. – p.29/46
Wrong type error location.

val average = fn weight => fn list =>
    let val iterator = fn (x, (sum, length)) =>
        (sum + weight x, length + 1)
    val (sum, length) = foldl iterator (0, 0) list
    in sum div length end end

val find_best = fn weight => fn lists =>
    let val average = average weight
    val iterator = fn (list, (best, max)) =>
        let val avg_list = average list
        in if avg_list > max then
            (list, avg_list)
        else
            (best, max)
        end
    val (best, _) = foldl iterator (nil, 0) lists
    in best end

val find_best_simple = find_best 1
val average = fn weight => fn list =>
    let val iterator = fn (x,(sum,length)) =>
        (sum + weight x, length + 1)
        val (sum,length) = foldl iterator (0,0) list
    in sum div length end

val find_best = fn weight => fn lists =>
    let val average = average weight
        val iterator = fn (list,(best,max)) =>
            let val avg_list = average list
                in if avg_list > max then
                    (list,avg_list)
                else
                    (best,max)
            end
        val (best,_) = foldl iterator (nil,0) lists
    in best end

val find_best_simple = find_best 1
Correct type error location.

```haskell
val average = fn weight => fn list =>
  let val iterator = fn (x,(sum,length)) =>
      (sum + weight * x, length + 1)
  in
    val (sum,length) = foldl iterator (0,0) list
    in
      sum div length
  end

val find_best = fn weight => fn lists =>
  let val average = average weight
    in
      val iterator = fn (list,(best,max)) =>
        let val avg_list = average list
        in
          if avg_list > max then
            (list,avg_list)
          else
            (best,max)
        end
      end
    in
      val (best,_) = foldl iterator (nil,0) lists
      in
        best
    end

val find_best_simple = find_best 1
```
Type error slice.

type constructor clash,
endpoints: function vs. int

```scala
.. val average = fn weight =>
   (.. weight (..) ..)

.. val find_best = fn weight =>
   (.. average weight ..)

.. find_best 1 ..)
```
A possible fix.

type constructor clash,
endpoints: function vs. int

(.. val average = fn weight =>
  (.. weight (..) ..)
)

.. val find_best = fn weight =>
  (.. average weight ..)

.. find_best 1 ..)
A possible fix.

type constructor clash,
endpoints: function vs. int

(\text{.. \ val average = fn weight =>}
(\text{.. weight} \times (\text{..}) ..)

\text{.. \ val find\_best = fn weight =>}
(\text{.. average weight ..})

\text{.. find\_best 1 ..})
Another possible fix.

type constructor clash,
endpoints: function vs. int

(\cell{.. \text{val} \ \text{average} = \text{fn} \ \text{weight} -> }
 (\cell{.. \ \text{weight} (\cell{..}) \ ..})

\cell{.. \ \text{val} \ \text{find\_best} = \text{fn} \ \text{weight} -> }
 (\cell{.. \ \text{average} \ \text{weight} \ ..})

\cell{.. \ \text{find\_best \ 1} \ ..})
Another possible fix.

type constructor clash, endpoints: function vs. int

(.. val average = fn weight => 
  (.. weight (..) ..)

.. val find_best = fn weight => 
  (.. average weight ..)

.. find_best (fn x => x) ..)
Yet another possible fix.

type constructor clash,
endpoints: function vs. int

(. val average = fn weight =>
 (. weight (.) ..)

(. val find_best = fn weight =>
 (. average weight ..)

(. find_best 1 ..)
Yet another possible fix.

type constructor clash,
endpoints: function vs. int

(\[..\] val average = fn weight =>
  (\[..\] weight (\[..\]) ..)

\[..\] val find_best = fn weight =>
  (\[..\] average (fn x => weight * x) ..)

\[..\] find_best 1 ..)
Overview.

• Basic concepts of types.
• Type polymorphism.
• Compositionality and principality.
• Case study: Type error slicing made possible by compositionality.

**Case study:** Getting principal typings in the $\lambda$-calculus with polymorphism.

• Conclusion.
Case study: Intersection type inference.

I will now present some details on how to actually do compositional type inference for a system that has type polymorphism.
Case study: Intersection type inference.

I will now present some details on how to actually do compositional type inference for a system that has type polymorphism.

This involves inferring types using both ordinary function types and intersection types to provide polymorphism.
Case study: Intersection type inference.

I will now present some details on how to actually do compositional type inference for a system that has type polymorphism.

This involves inferring types using both ordinary function types and intersection types to provide polymorphism.

The key mechanism to understand is expansion, which is presented here via a well chosen example.
A problematic type inference example.

Consider typing this example $\lambda$-term:

$$M = \underbrace{(\lambda x.x (\lambda y.y z))}_{N} \underbrace{(\lambda f.\lambda x.f (f x))}_{P}$$
A problematic type inference example.

Consider typing this example \( \lambda \)-term:

\[
M = (\lambda x. x (\lambda y. y z)) (\lambda f. \lambda x. f (f x))
\]

In an intersection type system, the usual principal typings of \( N \) and \( P \) are:

\[
N : \langle (z: a) \vdash T_1 \rightarrow c \rangle \quad \text{where} \quad T_1 = ((a \rightarrow b) \rightarrow b) \rightarrow c
\]

\[
P : \langle () \vdash T_2 \rangle \quad \text{where} \quad T_2 = ((e \rightarrow f) \cap (d \rightarrow e)) \rightarrow (d \rightarrow f)
\]
A problematic type inference example.

Consider typing this example \( \lambda \)-term:

\[
M = (\lambda x.x (\lambda y.y z)) (\lambda f.\lambda x.f (f x))
\]

In an intersection type system, the usual principal typings of \( N \) and \( P \) are:

\( N : \langle (z : a) \vdash T_1 \rightarrow c \rangle \) where \( T_1 = ((a \rightarrow b) \rightarrow b) \rightarrow c \)

\( P : \langle () \vdash T_2 \rangle \) where \( T_2 = ((e \rightarrow f) \cap (d \rightarrow e)) \rightarrow (d \rightarrow f) \)

To type \( M \), we must find derivable judgements such that:

\( N : \langle (z : T'') \vdash T \rightarrow T' \rangle \)

\( P : \langle () \vdash T \rangle \)

They ought to be obtainable from the principal typings.
Can we unify the example types? (1)

Can we unify $T_1$ and $T_2$ merely by substitution?

$$T_1 = (((a \rightarrow b) \rightarrow b) \rightarrow c$$
$$T_2 = (((e \rightarrow f) \cap (d \rightarrow e)) \rightarrow (d \rightarrow f)$$

Problem: clash between $\rightarrow$ and $\cap$. 

Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc. – p.40/46
Can we unify the example types? (1)

Can we unify $T_1$ and $T_2$ merely by substitution?

\[
T_1 = ((a \rightarrow b) \rightarrow b) \rightarrow c \\
T_2 = ((e \rightarrow f) \cap (d \rightarrow e)) \rightarrow (d \rightarrow f)
\]

Problem: clash between $\rightarrow$ and $\cap$.

Could we use $T \cap T = T$ to make the intersection go away?
Can we unify the example types? (2)

If using $T \cap T = T$, we now have 3 types to unify together:

```
Expand the diagram here...
```
Can we unify the example types? (2)

If using $T \cap T = T$, we now have 3 types to unify together:

Oh, no! We cannot solve $a \rightarrow b = b$ (without recursive types).

Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc. – p.41/46
Solving the example with expansion.

Instead, we do expansion [Coppo, Dezani-Ciancaglini, and Venneri, 1980] on the typing of \( N \) to solve the problem:

\[
N : \langle (z : a) \vdash ((a \to b) \to b) \to c \rangle
\]

\[
\downarrow
\]

\[
N : \langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c \rangle
\]
Solving the example with expansion.

Instead, we do expansion \cite{Coppo:1980} on the typing of $N$ to solve the problem:

\[
N : \langle (z : a) \vdash ((a \rightarrow b) \rightarrow b) \rightarrow c \rangle \\
\downarrow
\]

\[
N : \langle (z : a_1 \cap a_2) \vdash (((a_1 \rightarrow b_1) \rightarrow b_1) \cap ((a_2 \rightarrow b_2) \rightarrow b_2) \rightarrow c) \rangle
\]
Solving the example with expansion.

Instead, we do **expansion** [Coppo, Dezani-Ciancaglini, and Venneri, 1980] on the typing of $N$ to solve the problem:

$$N : \langle (z : a) ⊢ ((a → b) → b) → c \rangle \rightarrow c$$

$$\downarrow$$

$$N : \langle (z : a_1 \cap a_2) ⊢ (((a_1 → b_1) → b_1) \cap ((a_2 → b_2) → b_2) → c \rangle \rightarrow c\rangle$$

$$P : \langle () ⊢ (e → f) \cap (d → e) \rightarrow (d → f) \rangle$$
Solving the example with expansion.

Instead, we do expansion [Coppo, Dezani-Ciancaglini, and Venneri, 1980] on the typing of $N$ to solve the problem:

\[
N : \langle \text{ } (z : a) \vdash ( ((a \rightarrow b) \rightarrow b) \rightarrow c ) \rightarrow c \rangle
\]

\[
\downarrow
\]

\[
N : \langle (z : a_1 \cap a_2) \vdash ( ((a_1 \rightarrow b_1) \rightarrow b_1) \cap (a_2 \rightarrow b_2) \rightarrow b_2 ) \rightarrow c ) \rightarrow c \rangle
\]

\[
P : \langle \text{ } () \vdash ( e \rightarrow f ) \cap ( d \rightarrow e ) \rightarrow (d \rightarrow f) \rangle
\]

Then we apply this substitution (dotted lines above):

\[
S_f = (e := a_1 \rightarrow b_1, \ f := b_1, \ d := a_2 \rightarrow a_1 \rightarrow b_1,
\]

\[
b_2 := a_1 \rightarrow b_1, \ c := (a_2 \rightarrow a_1 \rightarrow b_1) \rightarrow b_1)
\]
Huh? What did you just do?

But how precisely did expansion go from the 1st to the 2nd typing for $N$?
Huh? What did you just do?

But how precisely did expansion go from the 1st to the 2nd typing for $N$?

Expansion simulated *in types* a transformation on the typing derivation for $N$ that inserted a use of the intersection-introduction typing rule at a deeply nested position.
Huh? What did you just do?

But how precisely did expansion go from the 1st to the 2nd typing for $N$?

Expansion simulated *in types* a transformation on the typing derivation for $N$ that inserted a use of the intersection-introduction typing rule at a deeply nested position.

Recently this has become much easier to understand due to a new definition using *expansion variables* (E-variables) [Kfoury and Wells, 1999; Carlier et al., 2004], which I will now show you.
How to do expansion with E-variables.

Applying $S = (e := (((a := a_1), b := b_1) \cap ((a := a_2), b := b_2))):$

Explaining Concepts in Compositional Type-Based Program Analysis: Principality, Intersection Types, Expansion, etc. – p.44/46
How to do expansion with E-variables.

Applying $S = (e := (((a := a_1), b := b_1) \cap ((a := a_2), b := b_2)))$:

Effect on typings:

$$
\langle (z : e a) \vdash (e ((a \to b) \to b) \to c) \to c \rangle
$$

$$
\xrightarrow{[S]} \langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c) \to c \rangle
$$
Overview.

- Basic concepts of types.
- Type polymorphism.
- Compositionality and principality.
- Case study: Type error slicing made possible by compositionality.
- Case study: Getting principal typings in the $\lambda$-calculus with polymorphism.

Conclusion.
Conclusion.

*Types* can be used for many *program analyses* and are already equivalent to *flow analysis*.
Conclusion.

- Types can be used for many program analyses and are already equivalent to flow analysis.

- Type polymorphism is vital, and can be obtained via either “for all” quantifiers or intersection types.
Conclusion.

Types can be used for many program analyses and are already equivalent to flow analysis.

Type polymorphism is vital, and can be obtained via either “for all” quantifiers or intersection types.

Compositional analysis is more suitable for a number of scenarios that are becoming more common, and principal typings enable compositional analysis.
Conclusion.

Types can be used for many program analyses and are already equivalent to flow analysis.

Type polymorphism is vital, and can be obtained via either “for all” quantifiers or intersection types.

Compositional analysis is more suitable for a number of scenarios that are becoming more common, and principal typings enable compositional analysis.

Getting compositionality is hard with “for all” quantifiers, so there may be motivation to learn intersection types and similar technologies.
Conclusion.

- **Types** can be used for many *program analyses* and are already equivalent to *flow analysis*.

- **Type polymorphism** is vital, and can be obtained via either “for all” quantifiers or intersection types.

- **Compositional analysis** is more suitable for a number of scenarios that are becoming more common, and *principal typings* enable compositional analysis.

- Getting compositionality is hard with “for all” quantifiers, so there may be motivation to learn *intersection types* and similar technologies.

- Doing compositional analysis with intersection types requires *expansion*. This is now much better understood and can be done with *E-variables*. 

Explaining Concepts in Compositional Type-Based Program Analysis:Principality, Intersection Types, Expansion, etc. – p.46/46
References


