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 λ-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:

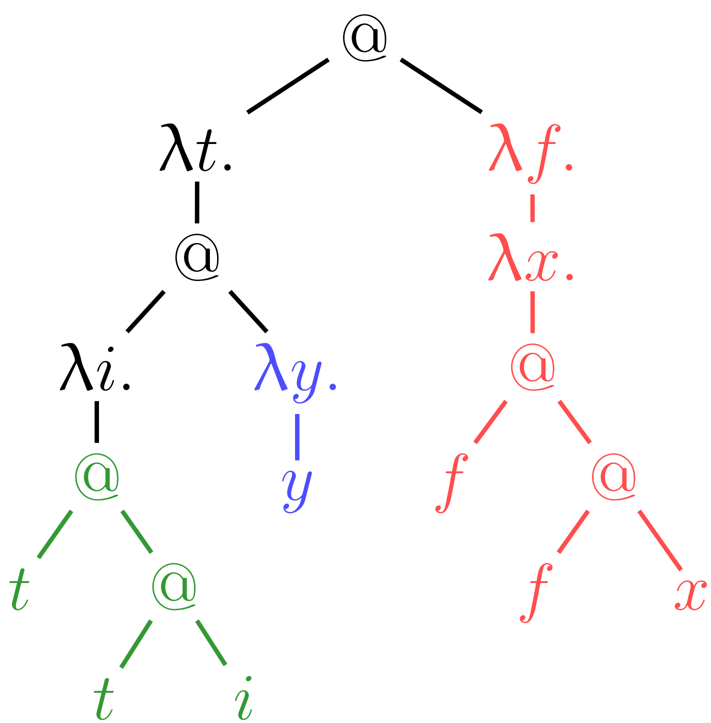
```
fun twice f x = f (f x);
fun id z = z;
twice (twice id);
```

An example program.

This Standard ML (SML) program:

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

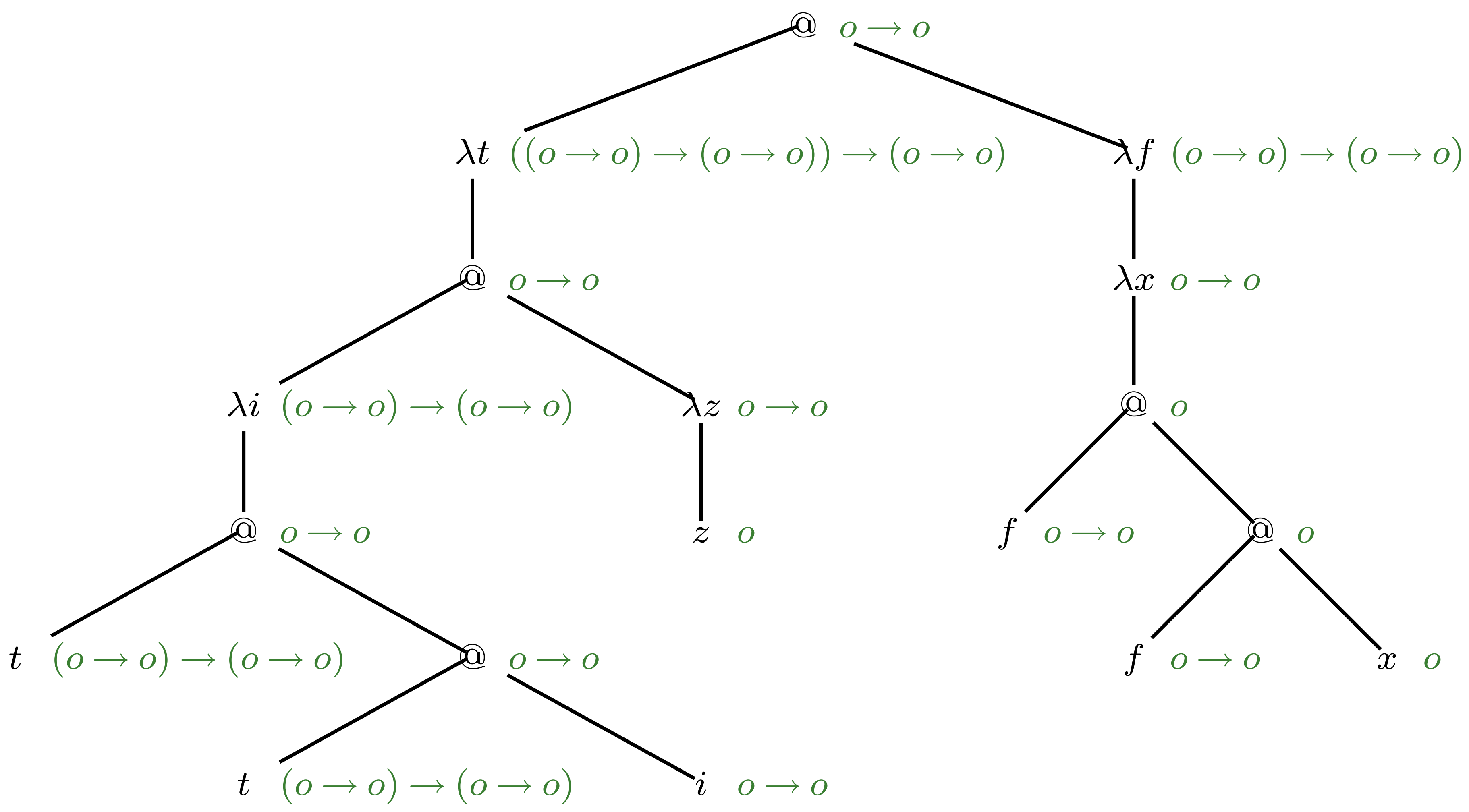
is the same as this λ-term:



$$= (\lambda t.(\lambda i.t\ (t\ i))\ (\lambda y.y))(\lambda f.\lambda x.f\ (f\ x))$$

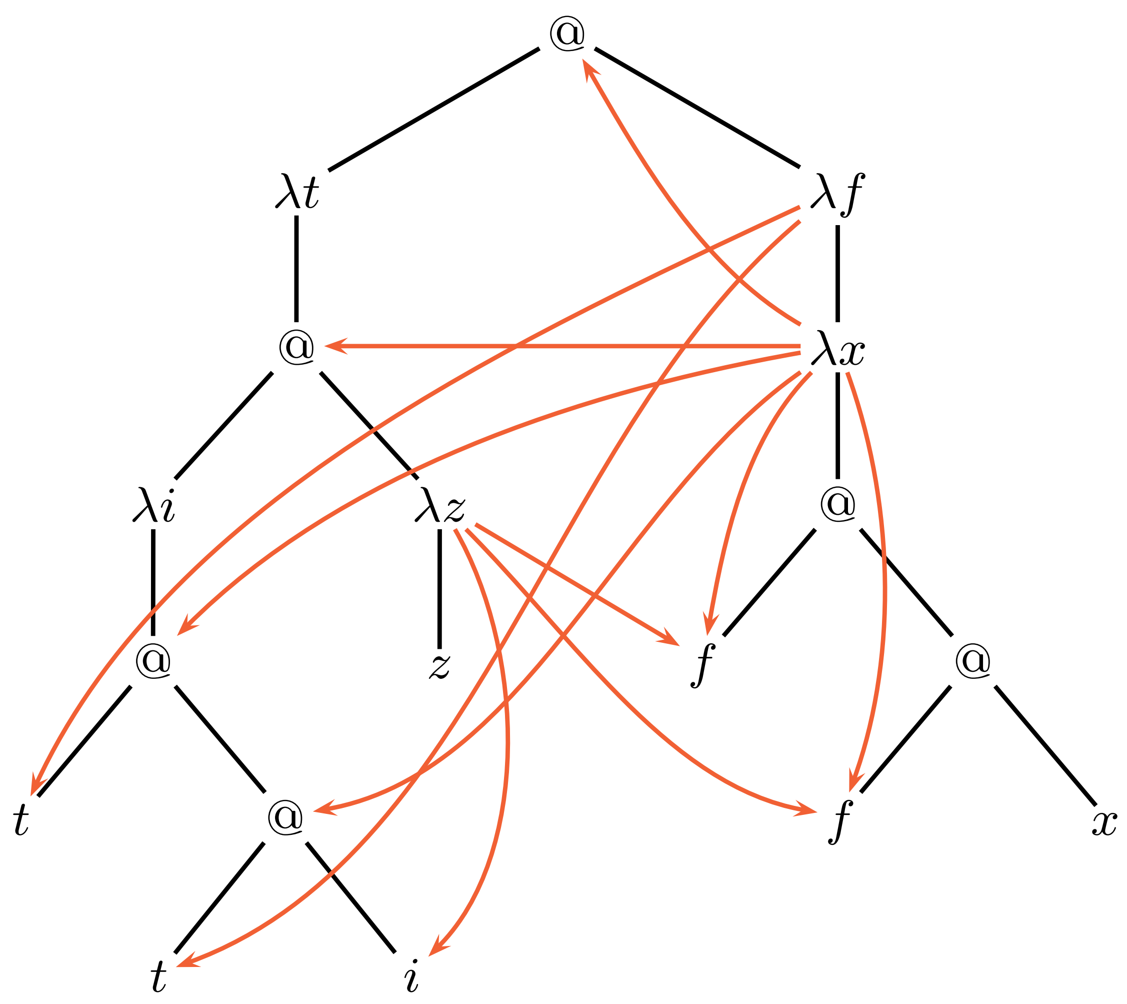
Example: Types.

Our example analyzed using the simply typed λ-calculus:



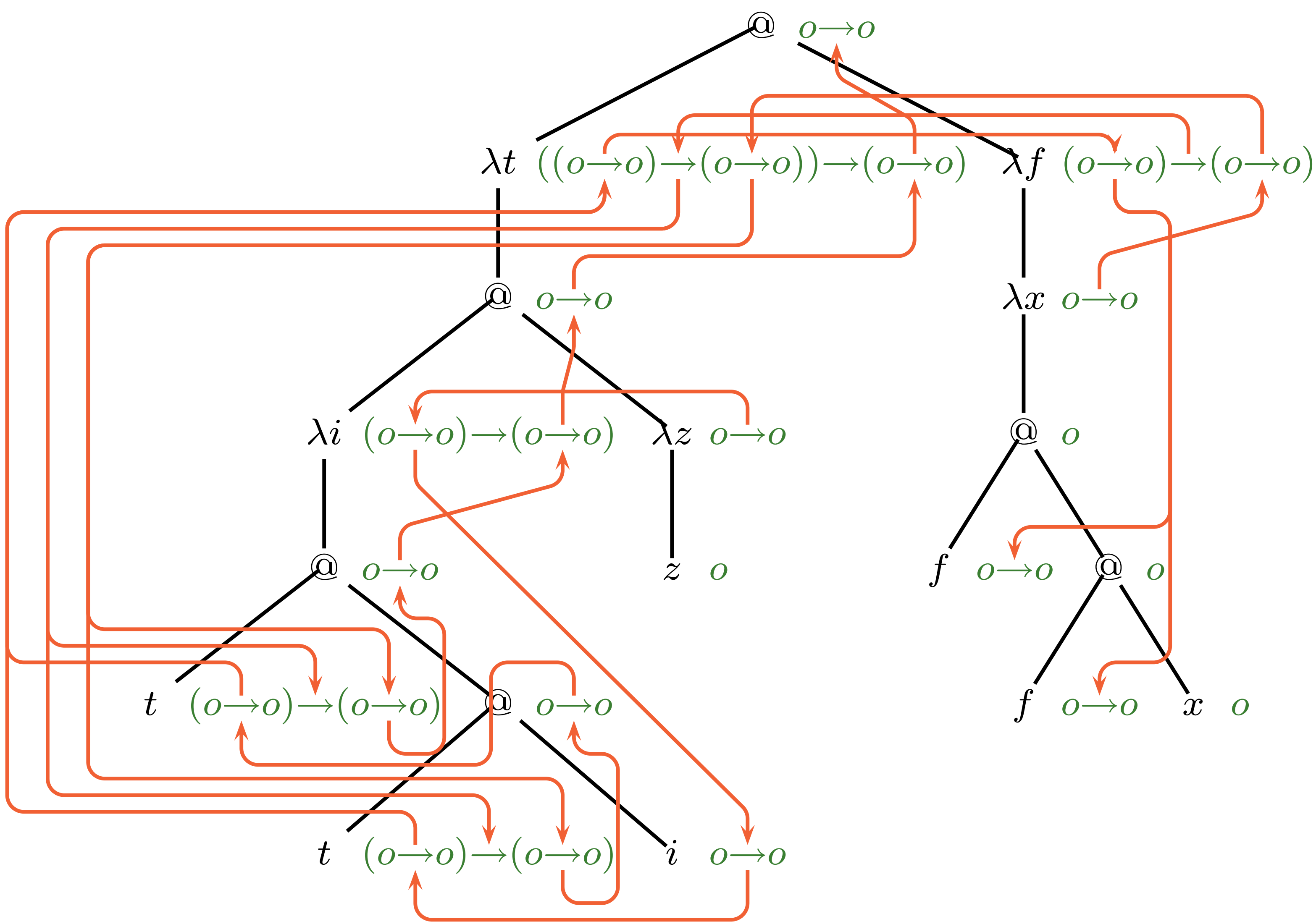
Example: Flow.

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



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 λ-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.

$$\mathtt{val\ swap}^{\forall \mathtt{a},\mathtt{b}.(\mathtt{a}\times\mathtt{b})\to(\mathtt{b}\times\mathtt{a})} = (\mathtt{fn}\ (\mathtt{x}^{\mathtt{a}}, \mathtt{y}^{\mathtt{b}}) \Rightarrow (\mathtt{y}^{\mathtt{b}},\ \mathtt{x}^{\mathtt{a}}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},$$
$$\mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});$$

Example: "for all" quantifiers.

$$\mathtt{val\ swap}^{\forall a,b.(a\times b)\to(b\times a)} = (\mathtt{fn}\ (\mathtt{x}^{a}, \mathtt{y}^{b}) \Rightarrow (\mathtt{y}^{b},\ \mathtt{x}^{a}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},$$
$$\ \mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});$$

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

Example: “for all” quantifiers.

$$\mathtt{val\ swap}^{\forall a,b.(a\times b)\to(b\times a)} = (\mathtt{fn}\ (\mathtt{x}^{a}, \mathtt{y}^{b}) \Rightarrow (\mathtt{y}^{b},\ \mathtt{x}^{a}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$\begin{array}{l}(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},\\ \ \mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});\end{array}$$

- 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.

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

$$\texttt{val closure2}^{\texttt{bool}\times(\texttt{bool}\rightarrow\texttt{bool})} = (\texttt{true},\ (\texttt{fn x} \Rightarrow \texttt{not x}));$$

$$\begin{aligned}\texttt{val closure} = \texttt{if b then } & \texttt{closure1}^{\exists\texttt{a}.\texttt{a}\times(\texttt{a}\rightarrow\texttt{bool})}\\ \texttt{else } & \texttt{closure2}^{\exists\texttt{a}.\texttt{a}\times(\texttt{a}\rightarrow\texttt{bool})};\end{aligned}$$

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

Example: “there exists” quantifiers.

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

$$\mathtt{val\ closure2}^{\mathtt{bool}\times(\mathtt{bool}\rightarrow\mathtt{bool})} = (\mathtt{true},\ (\mathtt{fn\ x} \Rightarrow \mathtt{not\ x}));$$

$$\begin{aligned}\mathtt{val\ closure} = \mathtt{if\ b\ then\ closure1}^{\exists\mathtt{a}.\mathtt{a}\times(\mathtt{a}\rightarrow\mathtt{bool})} \\ \mathtt{else\ closure2}^{\exists\mathtt{a}.\mathtt{a}\times(\mathtt{a}\rightarrow\mathtt{bool})};\end{aligned}$$

$$\mathtt{val\ result}^{\mathtt{bool}} = (\#2\ \mathtt{closure})^{\mathtt{a}\rightarrow\mathtt{bool}}(\#1\ \mathtt{closure})^{\mathtt{a}};$$

- Dual of universal quantifier.

Example: "there exists" quantifiers.

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

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

$$\begin{aligned}\texttt{val closure} = \texttt{if b then closure1}^{\exists\texttt{a}.\texttt{a}\times(\texttt{a}\to\texttt{bool})}\\ \texttt{else closure2}^{\exists\texttt{a}.\texttt{a}\times(\texttt{a}\to\texttt{bool})};\end{aligned}$$

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

- 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: $(\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{x})^{\forall \mathtt{a}.(\mathtt{a} \to \mathtt{a})}$

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

  Example is semantically like $\forall \mathtt{a} \in \{\mathtt{int}, \mathtt{real}\}.\mathtt{a} \to \mathtt{a}$, 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: $(\mathtt{fn}\ \mathtt{x} \Rightarrow \mathtt{x})^{\forall \mathtt{a}.(\mathtt{a} \to \mathtt{a})}$

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

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

- Named “intersection types” because in traditional model theory, semantic denotations $[\![T_1]\!]$ and $[\![T_2]\!]$ are program fragment sets and $[\![T_1 \cap T_2]\!] = [\![T_1]\!] \cap [\![T_2]\!]$ (usually).

Example: Intersection types.

$$\mathtt{val\ swap}^{\left(\substack{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})\\ \cap(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\right)}$$

$$= (\mathtt{fn}\ (\mathtt{x}^{\substack{\mathtt{int}\\ \mathtt{real}}}, \mathtt{y}^{\substack{\mathtt{bool}\\ \mathtt{real}}}) \Rightarrow (\mathtt{y}^{\substack{\mathtt{bool}\\ \mathtt{real}}},\ \mathtt{x}^{\substack{\mathtt{int}\\ \mathtt{real}}}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},$$
$$\ \mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});$$

Example: Intersection types.

$$\mathtt{val\ swap}^{\left(\substack{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})\\ \cap(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\right)}$$

$$= (\mathtt{fn}\ (\mathtt{x}^{\substack{\mathtt{int}\\ \mathtt{real}}}, \mathtt{y}^{\substack{\mathtt{bool}\\ \mathtt{real}}}) \Rightarrow (\mathtt{y}^{\substack{\mathtt{bool}\\ \mathtt{real}}}, \mathtt{x}^{\substack{\mathtt{int}\\ \mathtt{real}}}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},$$
$$\mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});$$

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

Example: Intersection types.

$$\mathtt{val\ swap}^{\left(\substack{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})\\ \cap(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\right)}$$

$$= (\mathtt{fn}\ (\mathtt{x}^{\substack{\mathtt{int}\\ \mathtt{real}}}, \mathtt{y}^{\substack{\mathtt{bool}\\ \mathtt{real}}}) \Rightarrow (\mathtt{y}^{\substack{\mathtt{bool}\\ \mathtt{real}}}, \mathtt{x}^{\substack{\mathtt{int}\\ \mathtt{real}}}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},$$
$$\mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});$$

- 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.

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

$$\mathtt{val\ closure2}^{\mathtt{bool}\times(\mathtt{bool}\rightarrow\mathtt{bool})} = (\mathtt{true},\ (\mathtt{fn\ x} \Rightarrow \mathtt{not\ x}));$$

$$\begin{aligned}\mathtt{val\ closure} = \mathtt{if\ b\ then\ closure1}&\left(\bigcup \begin{smallmatrix}\mathtt{int}\times(\mathtt{int}\rightarrow\mathtt{bool})\\ \mathtt{bool}\times(\mathtt{bool}\rightarrow\mathtt{bool})\end{smallmatrix}\right)\\ \mathtt{else\ closure2}&\left(\bigcup \begin{smallmatrix}\mathtt{int}\times(\mathtt{int}\rightarrow\mathtt{bool})\\ \mathtt{bool}\times(\mathtt{bool}\rightarrow\mathtt{bool})\end{smallmatrix}\right);\end{aligned}$$

$$\mathtt{val\ result}^{\mathtt{bool}} = (\#2\ \mathtt{closure})^{\begin{smallmatrix}\mathtt{int}\rightarrow\mathtt{bool}\\ \mathtt{bool}\rightarrow\mathtt{bool}\end{smallmatrix}}(\#1\ \mathtt{closure})^{\begin{smallmatrix}\mathtt{int}\\ \mathtt{bool}\end{smallmatrix}};$$

Example: Union types.

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

$$\mathtt{val\ closure2}^{\mathtt{bool}\times(\mathtt{bool}\to\mathtt{bool})} = (\mathtt{true},\ (\mathtt{fn\ x} \Rightarrow \mathtt{not\ x}));$$

$$\begin{aligned}\mathtt{val\ closure} = \mathtt{if\ b\ then\ closure1}&\left(\cup\begin{smallmatrix}\mathtt{int}\times(\mathtt{int}\to\mathtt{bool})\\ \mathtt{bool}\times(\mathtt{bool}\to\mathtt{bool})\end{smallmatrix}\right)\\ \mathtt{else\ closure2}&\left(\cup\begin{smallmatrix}\mathtt{int}\times(\mathtt{int}\to\mathtt{bool})\\ \mathtt{bool}\times(\mathtt{bool}\to\mathtt{bool})\end{smallmatrix}\right);\end{aligned}$$

$$\mathtt{val\ result}^{\mathtt{bool}} = (\#2\ \mathtt{closure})^{\begin{smallmatrix}\mathtt{int}\to\mathtt{bool}\\ \mathtt{bool}\to\mathtt{bool}\end{smallmatrix}}(\#1\ \mathtt{closure})^{\begin{smallmatrix}\mathtt{int}\\ \mathtt{bool}\end{smallmatrix}};$$

- Dual of intersection types.

Example: Union types.

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

$$\mathtt{val\ closure2}^{\mathtt{bool}\times(\mathtt{bool}\to\mathtt{bool})} = (\mathtt{true},\ (\mathtt{fn\ x} \Rightarrow \mathtt{not\ x}));$$

$$\begin{aligned}\mathtt{val\ closure} = \mathtt{if\ b\ then\ closure1}&^{\left(\bigcup \begin{smallmatrix}\mathtt{int}\times(\mathtt{int}\to\mathtt{bool})\\ \mathtt{bool}\times(\mathtt{bool}\to\mathtt{bool})\end{smallmatrix}\right)}\\ \mathtt{else\ closure2}&^{\left(\bigcup \begin{smallmatrix}\mathtt{int}\times(\mathtt{int}\to\mathtt{bool})\\ \mathtt{bool}\times(\mathtt{bool}\to\mathtt{bool})\end{smallmatrix}\right)};\end{aligned}$$

$$\mathtt{val\ result}^{\mathtt{bool}} = (\#2\ \mathtt{closure})^{\begin{smallmatrix}\mathtt{int}\to\mathtt{bool}\\ \mathtt{bool}\to\mathtt{bool}\end{smallmatrix}}(\#1\ \mathtt{closure})^{\begin{smallmatrix}\mathtt{int}\\ \mathtt{bool}\end{smallmatrix}};$$

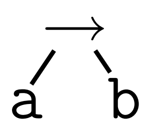
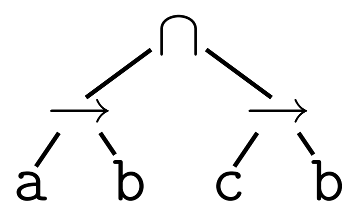
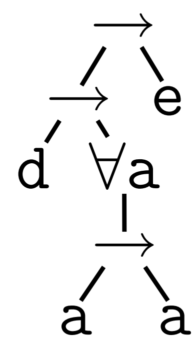
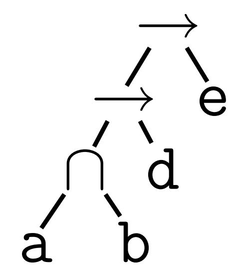
- Dual of intersection types.
- Exposes usage types throughout.

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].

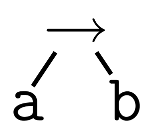
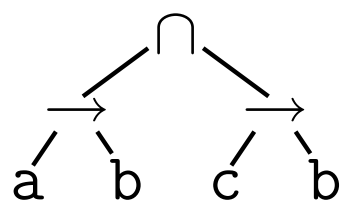
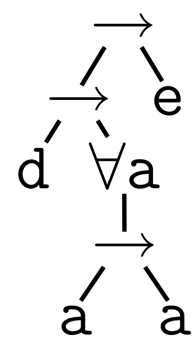
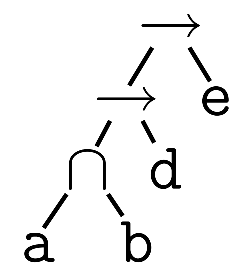
What is the rank of polymorphism?

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

Type				
Rank	0	1	2	3

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:

Type				
Rank	0	1	2	3

- 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.

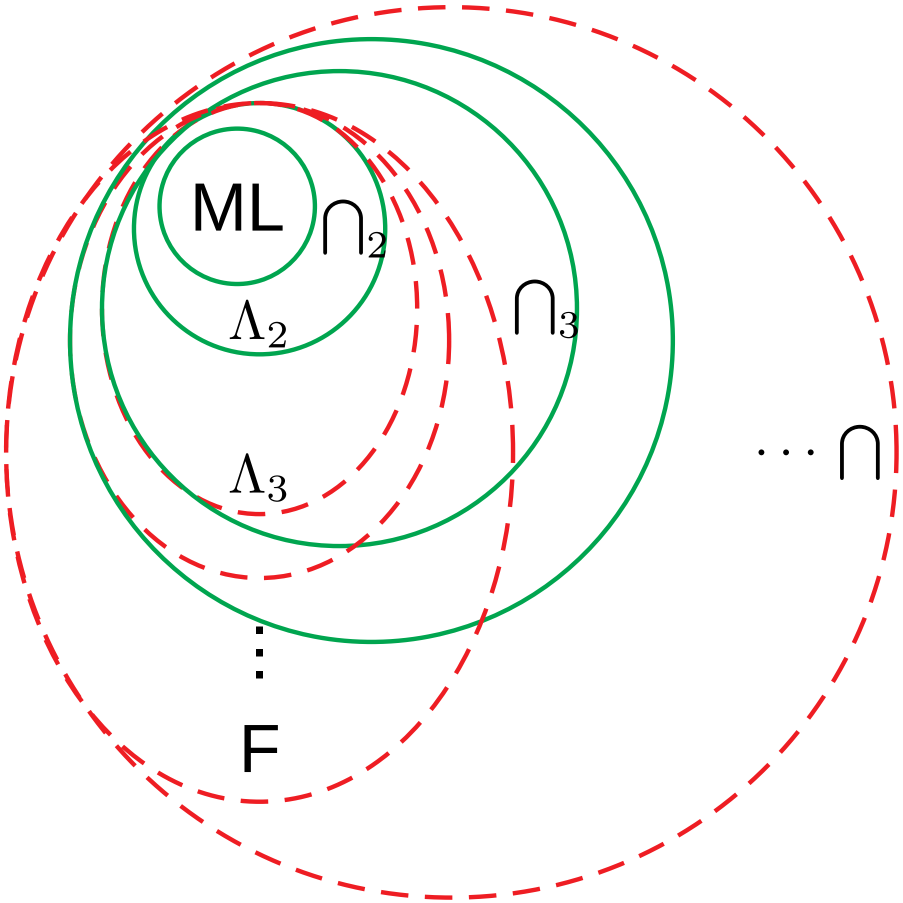
Λ_k: rank-k System F.

$\bigcap$: intersection types.

$\bigcap_k$: rank-k of $\bigcap$.

Decidable.

Undecidable.



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

Flexibility of intersection types.

$$M = \left(\begin{array}{l} \texttt{fun self\_apply2 z} \Rightarrow (\texttt{z z})\ \texttt{z}; \\ \texttt{fun apply f x} \Rightarrow \texttt{f x}; \\ \texttt{fun reverse\_apply y g} \Rightarrow \texttt{g y}; \\ \texttt{fun id w} \Rightarrow \texttt{w}; \\ (\texttt{self\_apply2 apply not true}, \\ \ \texttt{self\_apply2 reverse\_apply id false not}); \end{array} \right)$$

- Program fragment M *safely* computes $(\texttt{false}, \texttt{true})$.
- Urzyczyn [1997] proved that M is not typable in F_ω, and F_ω 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 λ-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 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.

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$\sharp$ 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$\sharp$ 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:

$$M : \overbrace{\langle A \vdash T \rangle}^{\textit{typing}}$$

untyped term *type environment* *result type*

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:

$$M : \overbrace{\langle A \vdash T \rangle}^{\textit{typing}}$$

untyped term *type environment* *result type*

A type system can thus be seen as a set of pairs of the form $(M : \Theta)$ where Θ 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 $\Theta_1 \leq_S \Theta_2$ ("Θ_1 is *at least as strong as* Θ_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 $\Theta_1 \leq_S \Theta_2$ ("Θ_1 is *at least as strong as* Θ_2 in system S") means $M : \Theta_1$ implies $M : \Theta_2$ for every M.
- A typing Θ for term M is *principal* exactly when Θ 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 $\Theta_1 \leq_S \Theta_2$ ("Θ_1 is *at least as strong as* Θ_2 in system S") means $M : \Theta_1$ implies $M : \Theta_2$ for every M.
- A typing Θ for term M is *principal* exactly when Θ 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 $\Theta_1 \leq_S \Theta_2$ ("Θ_1 is *at least as strong as* Θ_2 in system S") means $M : \Theta_1$ implies $M : \Theta_2$ for every M.
- A typing Θ for term M is *principal* exactly when Θ 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 $\Theta_1 \leq_S \Theta_2$ ("Θ_1 is *at least as strong as* Θ_2 in system S") means $M : \Theta_1$ implies $M : \Theta_2$ for every M.
- A typing Θ for term M is *principal* exactly when Θ 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.
- Until Wells [2002], each system with PTs had its own definition via syntactic operations like *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 ∀-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 $\forall$-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.
- 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 $(\mathtt{let}\ x = M\ \mathtt{in}\ N)$ 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 $(\mathtt{let}\ x = M\ \mathtt{in}\ N)$ it first analyzes M and then uses the result in analyzing N.
- Not use HM typings for intermediate results. E.g., the typing of $(\mathtt{x}\mathtt{x})$ in the Chap. 1 system of Damas [1985]:

  $$\langle(\mathtt{x} : \mathtt{a}, \mathtt{x} : \mathtt{a} \to \mathtt{b}) \vdash \mathtt{b}\rangle$$

  This is essentially intersection types, i.e.:

  $$\langle(\mathtt{x} : \mathtt{a} \cap (\mathtt{a} \to \mathtt{b})) \vdash \mathtt{b}\rangle$$

  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.

```
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
```

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

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
```

Another 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

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.

```
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
```

Type error slice.

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

```
(.. 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 ..)
```

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 ..)
```

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 λ-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 λ-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 λ-term:

$$M = \underbrace{(\lambda x.x\,(\lambda y.y\;z))}_{N}\,\underbrace{(\lambda f.\lambda x.f\;(f\;x))}_{P}$$

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

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

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

A problematic type inference example.

Consider typing this example λ-term:

$$M = \underbrace{(\lambda x.x\,(\lambda y.y\;z))}_{N}\,\underbrace{(\lambda f.\lambda x.f\;(f\;x))}_{P}$$

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

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

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

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

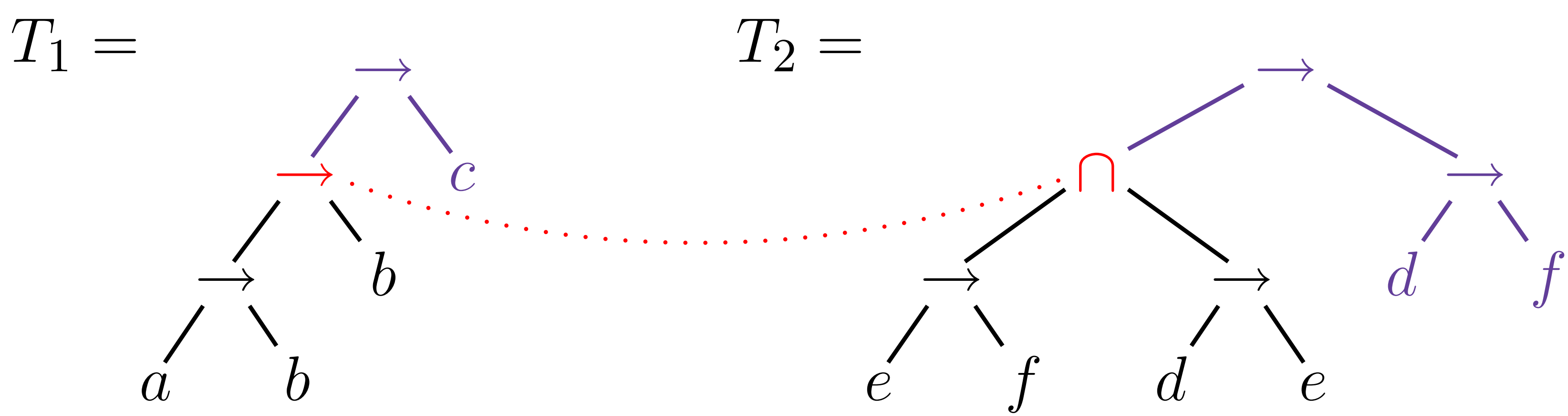
$$N : \langle (z : T'') \vdash T \to T' \rangle \qquad P : \langle () \vdash T \rangle$$

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?

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

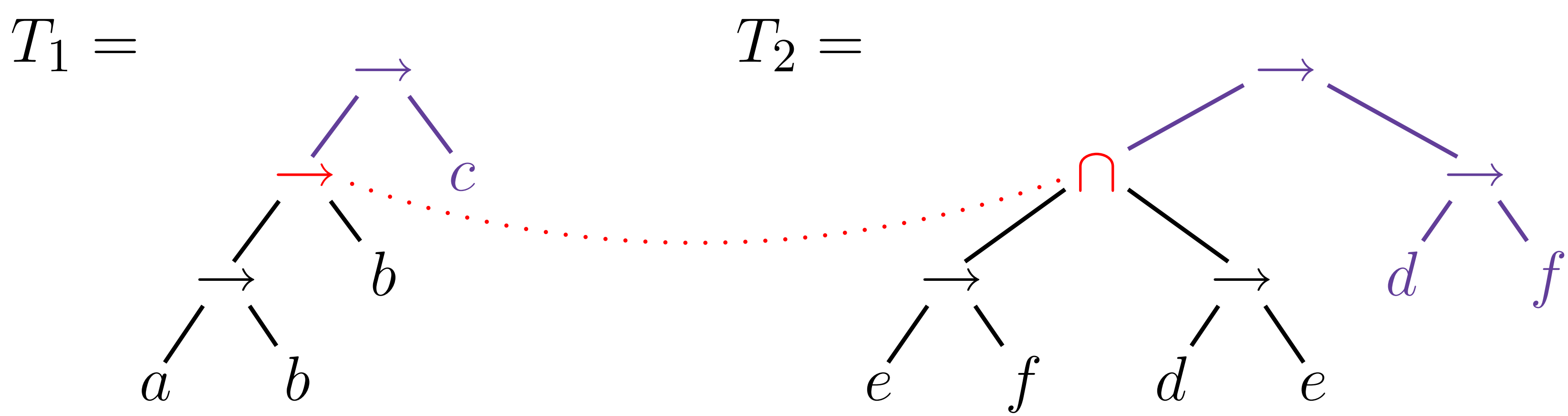


Problem: clash between $\to$ and $\cap$.

Can we unify the example types? (1)

Can we unify T_1 and T_2 merely by substitution?

$$
\begin{aligned}
T_1 &= ((a \rightarrow b) \rightarrow \quad b \quad) \rightarrow \quad c \\
T_2 &= ((e \rightarrow f) \cap (d \rightarrow e)) \rightarrow (d \rightarrow f)
\end{aligned}
$$

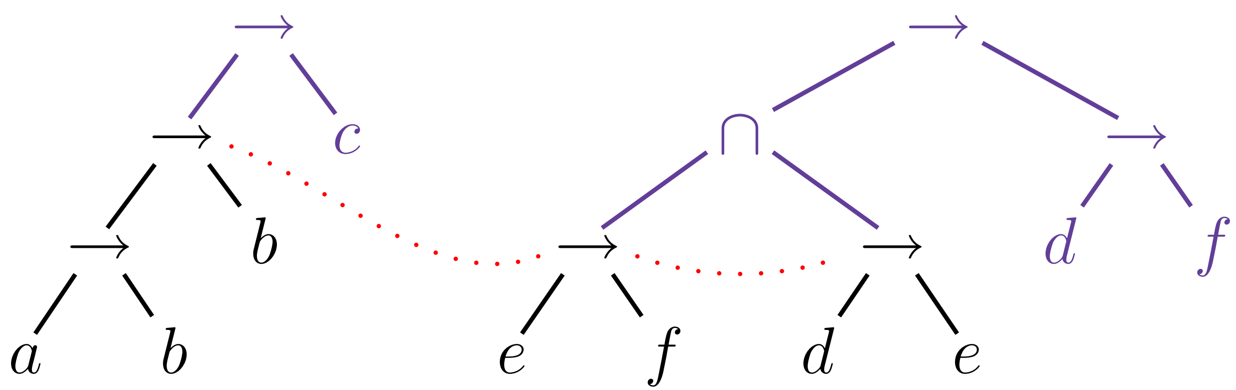


Problem: clash between $\rightarrow$ and $\cap$.

Could we use $T \cap T = T$ to make the intersection go away?

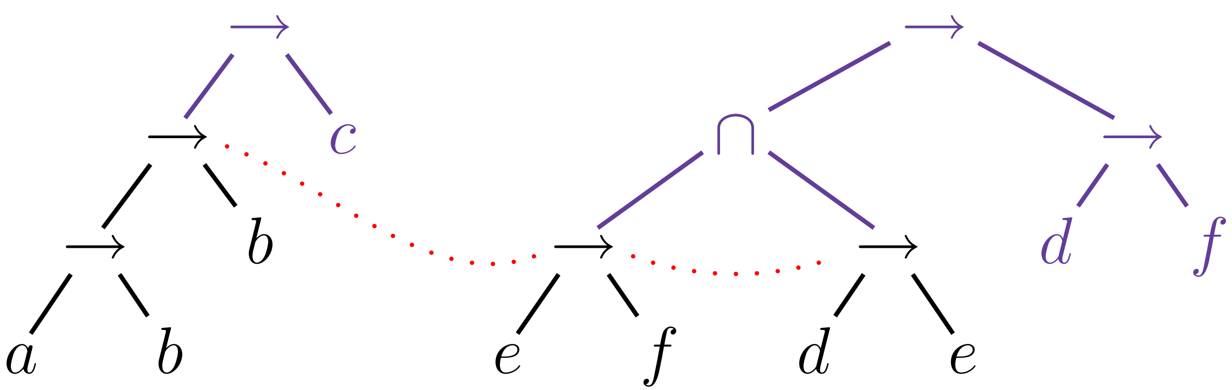
Can we unify the example types? (2)

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

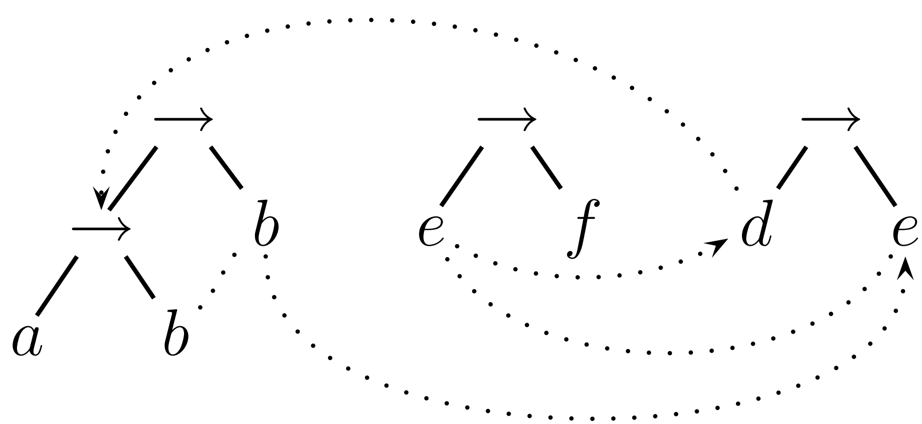


Can we unify the example types? (2)

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



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



Solving the example with expansion.

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

$$N : \langle \quad (z : a) \quad \vdash (\qquad ((a \to b) \to b) \qquad \to \quad c \quad) \to c \rangle$$

$$\downarrow$$

$$N : \langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to \quad c \quad) \to c \rangle$$

Solving the example with expansion.

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

$$N : \langle (z : a) \vdash (\underbrace{((a \to b) \to b)} \to c) \to c \rangle$$

$$\downarrow$$

$$N : \langle (z : a_1 \cap a_2) \vdash (\overbrace{((a_1 \to b_1) \to b_1)} \cap \overbrace{((a_2 \to b_2) \to b_2)} \to c) \to c \rangle$$

Solving the example with expansion.

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

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

$$\downarrow$$

$$N : \langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c) \to c \rangle$$

$$P : \langle () \vdash (e \to f) \cap (d \to e) \to (d \to f) \rangle$$

Solving the example with expansion.

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

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

$$\downarrow$$

$$N : \langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c) \to c \rangle$$

$$P : \langle () \vdash (e \to f) \cap (d \to e) \to (d \to f) \rangle$$

Then we apply this substitution (dotted lines above):

$$S_{\mathsf{f}} = (e := a_1 \to b_1,\ f := b_1,\ d := a_2 \to a_1 \to b_1,\ b_2 := a_1 \to b_1,\ c := (a_2 \to a_1 \to b_1) \to b_1\ )$$

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?

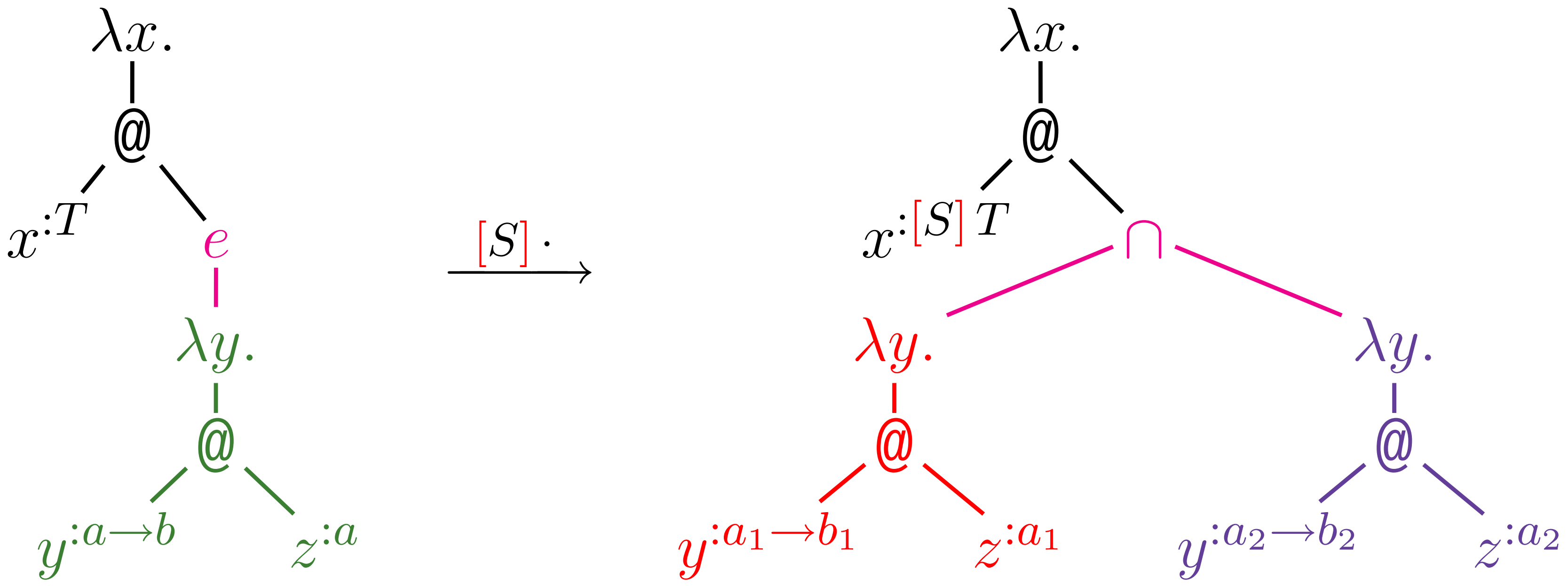
But how precisely did expansion go from the 1st to the 2nd typing for N?

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.

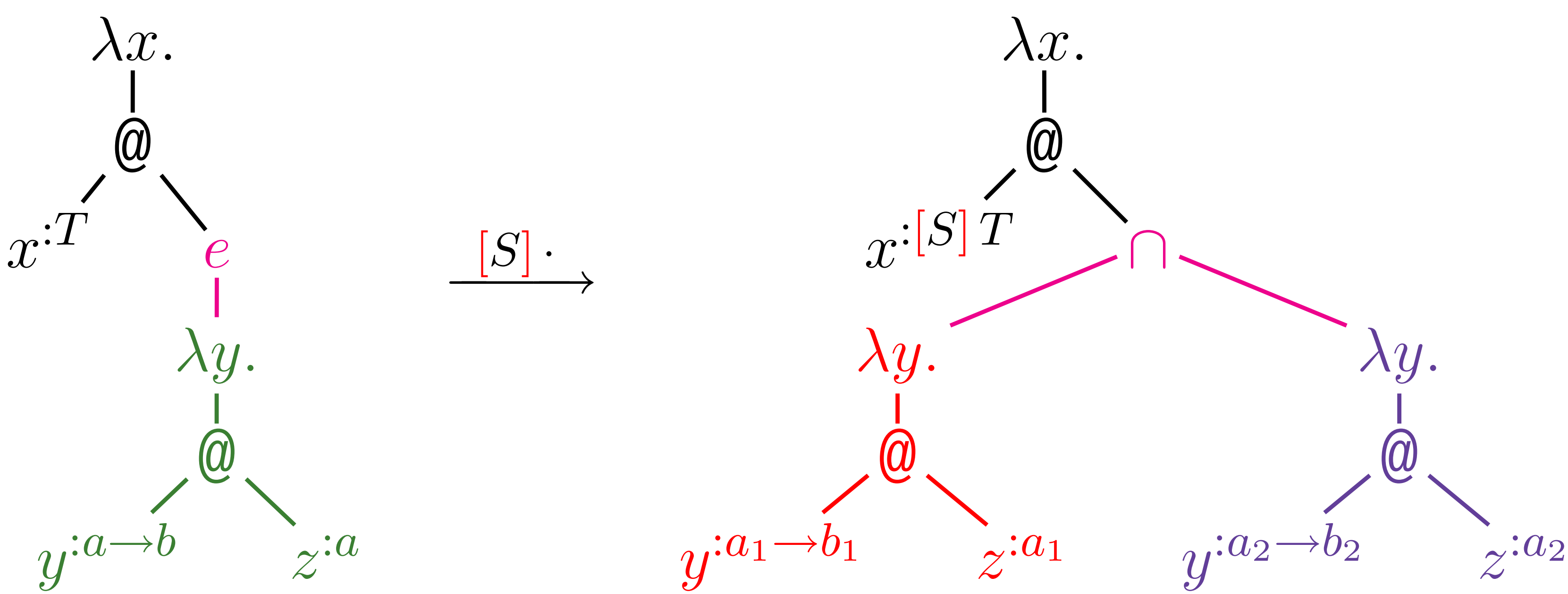
How to do expansion with E-variables.

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



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:

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

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

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 λ-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*.

References

Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*, pages 1–10. ACM Press, 1997. ISBN 0-89791-918-1. URL .

Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York, Stony Brook, November 1995.

Sébastien Carlier, Jeff Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, volume 2986 of *LNCS*, pages 294–309. Springer-Verlag, 2004. ISBN 3-540-21313-9.

M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard type inference. *Theoret. Comput. Sci.*, 272(1-2):69–111, February 2002.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ-calculus semantics. In J. R[oger] Hindley and J[onathan] P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560. Academic Press, 1980. ISBN 0-12-349050-2.

L. Damas and Robin Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 207–212, 1982.

Luis Manuel Martins Damas. *Type assignment in Programming Languages*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, April 1985.

F. Damiani and P. Giannini. Automatic useless-code detection and elimination for HOT functional programs. *J. Funct. Programming*, pages 509–559, 2000.

Ferruccio Damiani. A conjunctive type system for useless-code elimination. *Math. Structures Comput. Sci.*, 13:157–197, 2003.

Paola Giannini, Furio Honsell, and Simona Ronchi Della Rocca. Type inference: Some results, some problems. *Fund. Inform.*, 19(1/2):87–125, September/October 1993.

J[ean]-Y[ves] Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse d'Etat, Université de Paris VII, 1972.

Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Programming Languages & Systems, 12th European Symp. Programming*, volume 2618 of *LNCS*, pages 284–301. Springer-Verlag, 2003. ISBN 3-540-00886-1. Superseded by Haack and Wells [2004].

Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Programming*, 50:189–224, 2004. doi: doi:10.1016/j.scico.2004.01.004. Supersedes Haack and Wells [2003].

Thomas Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998. ISBN 0-89791-979-3.

Trevor Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

Assaf J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pages 161–174, 1999. ISBN 1-58113-095-3. Superseded by Kfoury and Wells [2004].

Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes Kfoury and Wells [1999], August 2003.

Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004. doi: doi:10.1016/j.tcs.2003.10.032. Supersedes Kfoury and Wells [1999]. For omitted proofs, see the longer report Kfoury and Wells [2003].

Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pages 90–101. ACM Press, 1999. ISBN 1-58113-111-9.

Daniel Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 88–98, 1983. ISBN 0-89791-090-7.

I. Margaria and M. Zacchi. Principal typing in a $\forall\cap$-discipline. *J. Logic Comput.*, 5(3):367–381, 1995.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Langs. & Systs.*, 10(3): 470–502, July 1988.

J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

Simona Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2):181–209, March 1988.

Zhong Shao and Andrew Appel. Smartest recompilation. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, 1993.

Olin Shivers. *Control Flow Analysis of Higher Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In *Proc. 1st Int'l Static Analysis Symp.*, pages 408–422, 1994.

Paweł Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Structures Comput. Sci.*, 7(4):329–358, 1997.

Steffen J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002. ISBN 3-540-43864-5.