Rewriting in the Design of Type Systems

Joe Wells

ULTRA Group

School of Mathematical and Computer Sciences

Heriot-Watt University

`http://www.macs.hw.ac.uk/~jbw/`

Overview

- **Goals for this talk**
- Type inference as rewriting via an example with simple types
- Intersection types and why you might want them
- Type inference as rewriting via an example with intersection types
- Various concluding remarks

Some Goals for this Talk

- Possibly enlarge some audience members' conception of what types can be.

Some Goals for this Talk

- Possibly enlarge some audience members' conception of what types can be.
- Show examples where it is reasonable to use similar syntax for types and terms.

Some Goals for this Talk

- Possibly enlarge some audience members' conception of what types can be.
- Show examples where it is reasonable to use similar syntax for types and terms.
- Show how the definition of a type system might be based on rewriting on the terms of the system.

Some Goals for this Talk

- Possibly enlarge some audience members' conception of what types can be.
- Show examples where it is reasonable to use similar syntax for types and terms.
- Show how the definition of a type system might be based on rewriting on the terms of the system.
- Give what may be a clearer explanation of type inference in System I [Kfoury and Wells, 1999].

Some Important Points

- Non-definitional aspects of types:

Some Important Points

- Non-definitional aspects of types:
  - Types may be used for *description* or *prescription*.

Some Important Points

- Non-definitional aspects of types:
  - Types may be used for *description* or *prescription*.
  - Types may be intended for reading by *humans* or *computers*.

Some Important Points

- Non-definitional aspects of types:
  - Types may be used for *description* or *prescription*.
  - Types may be intended for reading by *humans* or *computers*.
  - Types may be *easy* or *hard* to determine.

Some Important Points

- Non-definitional aspects of types:
  - Types may be used for *description* or *prescription*.
  - Types may be intended for reading by *humans* or *computers*.
  - Types may be *easy* or *hard* to determine.
- It is not immoral/wrong if types are not formulas of well known independently interesting logics.

Some Important Points

- Non-definitional aspects of types:
  - Types may be used for *description* or *prescription*.
  - Types may be intended for reading by *humans* or *computers*.
  - Types may be *easy* or *hard* to determine.
- It is not immoral/wrong if types are not formulas of well known independently interesting logics.
- Reasoning about a software system is *compositional* if the pieces are reasoned about independently and the results are composed without reinspecting the pieces.

Overview

- Goals for this talk
- **Type inference as rewriting via an example with simple types**
- Intersection types and why you might want them
- Type inference as rewriting via an example with intersection types
- Various concluding remarks

An Example Program

This ML program:

$$\texttt{fun twice f x} = \texttt{f}\ (\texttt{f x});$$
$$\texttt{fun id z} = \texttt{z};$$
$$\texttt{twice}\ (\texttt{twice id});$$

An Example Program

This ML program:

$\texttt{fun twice f x} = \texttt{f}\ (\texttt{f x});$

$\texttt{fun id z} = \texttt{z};$

$\texttt{twice}\ (\texttt{twice id});$

can be seen as this λ-term:

$$(\lambda t.(\lambda i.t(ti))(\lambda z.z))(\lambda f.\lambda x.f(fx))$$

An Example Program

This ML program:

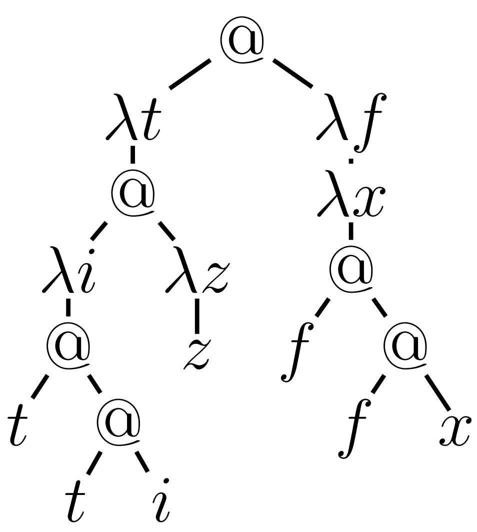
$$\texttt{fun twice f x} = \texttt{f}\ (\texttt{f x});$$

$$\texttt{fun id z} = \texttt{z};$$

$$\texttt{twice}\ (\texttt{twice id});$$

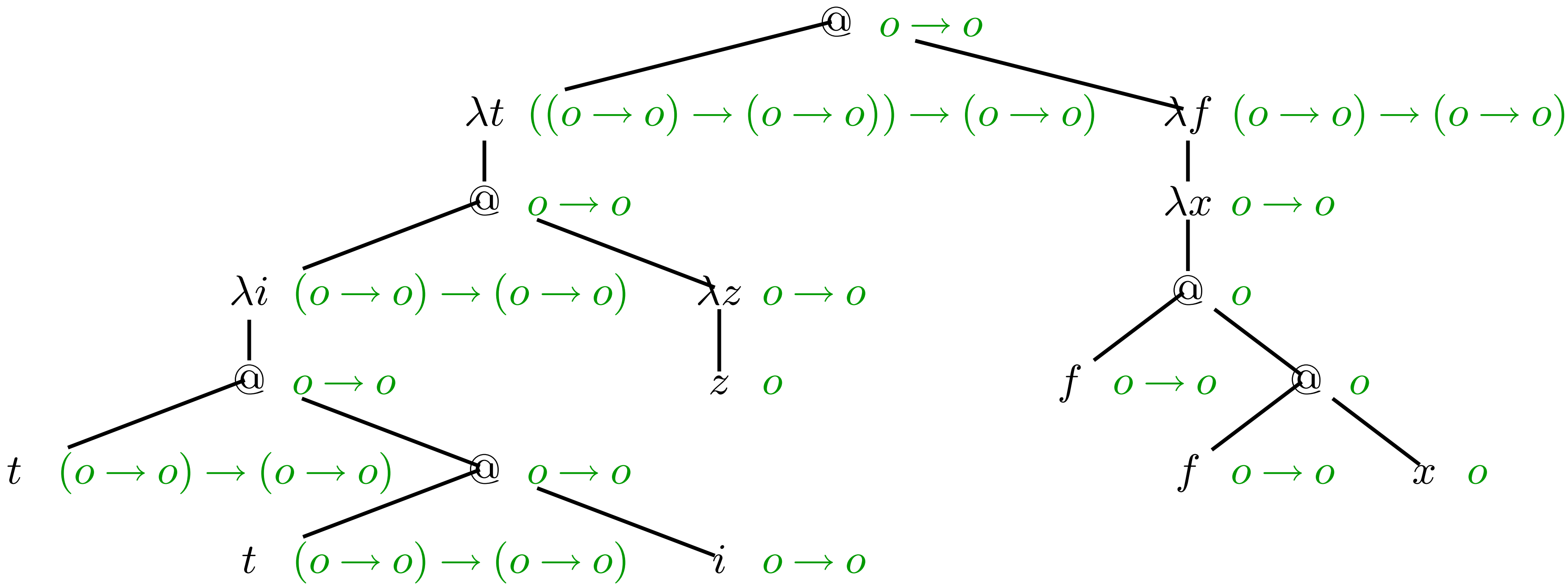
can be seen as this λ-term:
$(\lambda t.(\lambda i.t(ti))(\lambda z.z))(\lambda f.\lambda x.f(fx))$

which can be drawn as this tree:



Simple Types for the Example

Our example analyzed using the simply typed λ-calculus:



Type Inference for Simple Types

- There is a well known analysis algorithm using Robinson's unification algorithm as a subprocedure (see Hindley [1997] for details).

Type Inference for Simple Types

- There is a well known analysis algorithm using Robinson's unification algorithm as a subprocedure (see Hindley [1997] for details).
- Pretty much everything is known about type inference for simple types, including the complexity ($O(n)$ under standard assumptions).

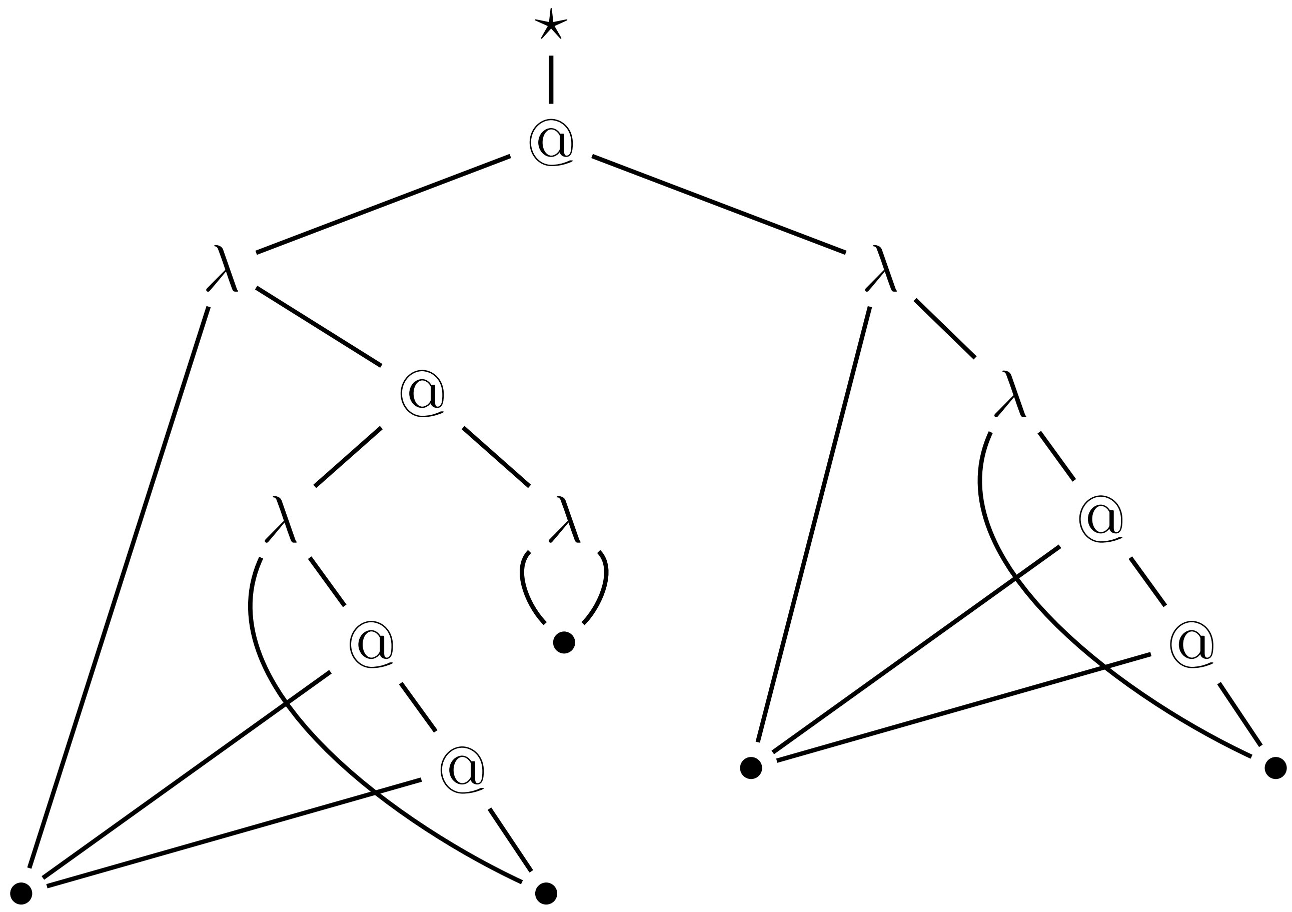
Type Inference for Simple Types

- There is a well known analysis algorithm using Robinson's unification algorithm as a subprocedure (see Hindley [1997] for details).
- Pretty much everything is known about type inference for simple types, including the complexity ($O(n)$ under standard assumptions).
- However, I think it will be helpful to view the process from a different angle. Probably someone else has done something like this before, perhaps not quite the way I will do it.

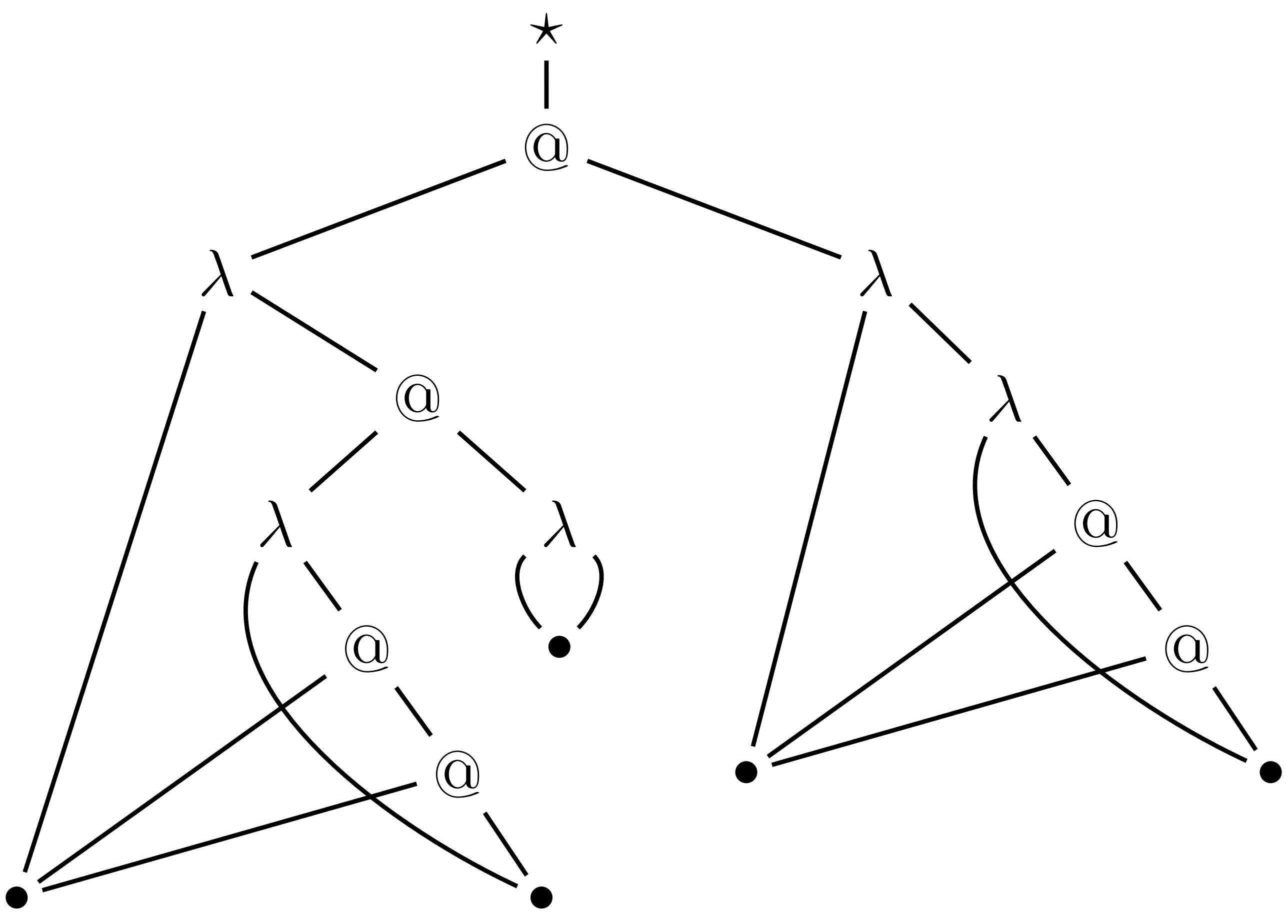
Type Inference for Simple Types

- There is a well known analysis algorithm using Robinson's unification algorithm as a subprocedure (see Hindley [1997] for details).
- Pretty much everything is known about type inference for simple types, including the complexity ($O(n)$ under standard assumptions).
- However, I think it will be helpful to view the process from a different angle. Probably someone else has done something like this before, perhaps not quite the way I will do it.
- It is not essential that the following diagrams are DAGs, but any implementation would do so and it makes the examples fit.

Another Look at the Lambda Term

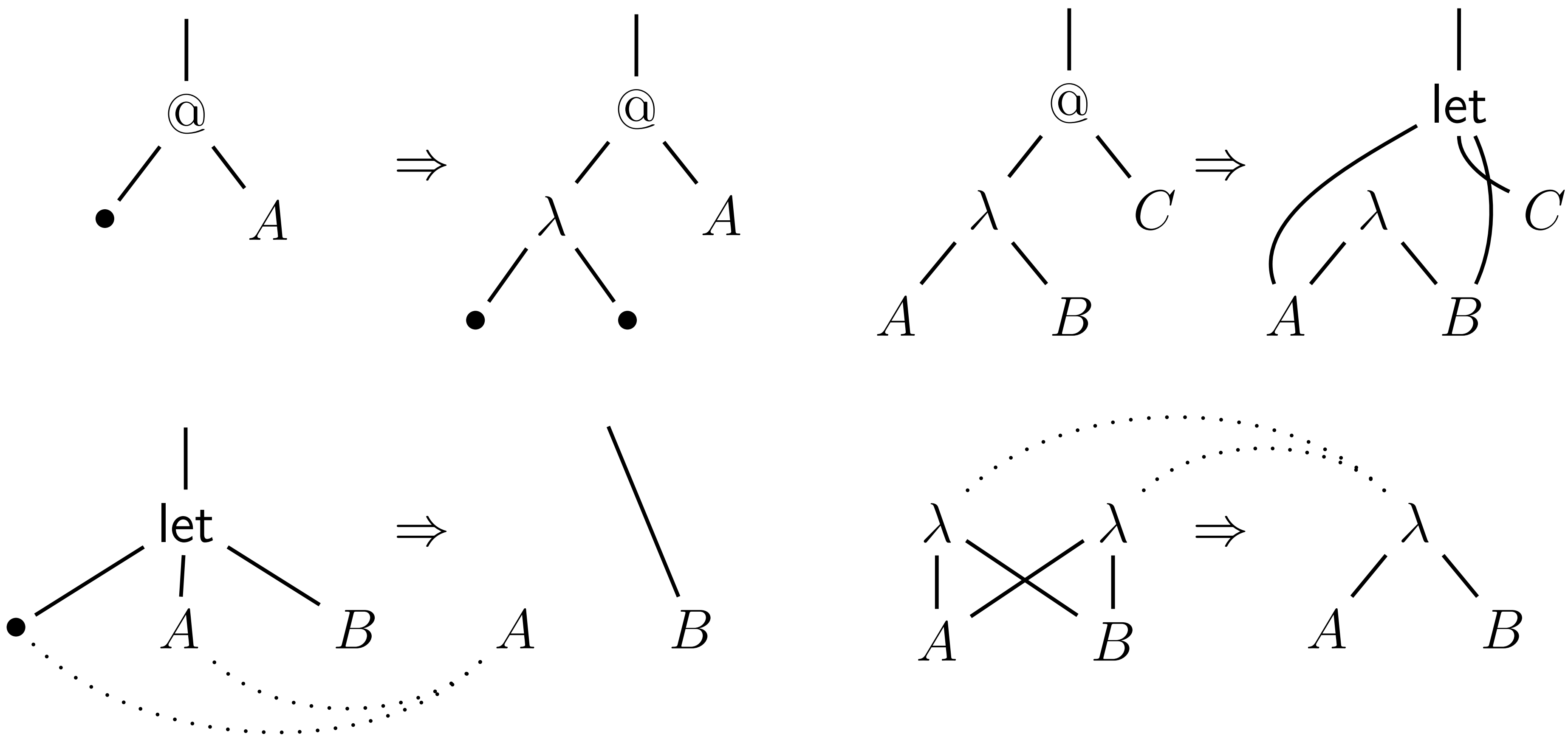


Another Look at the Lambda Term

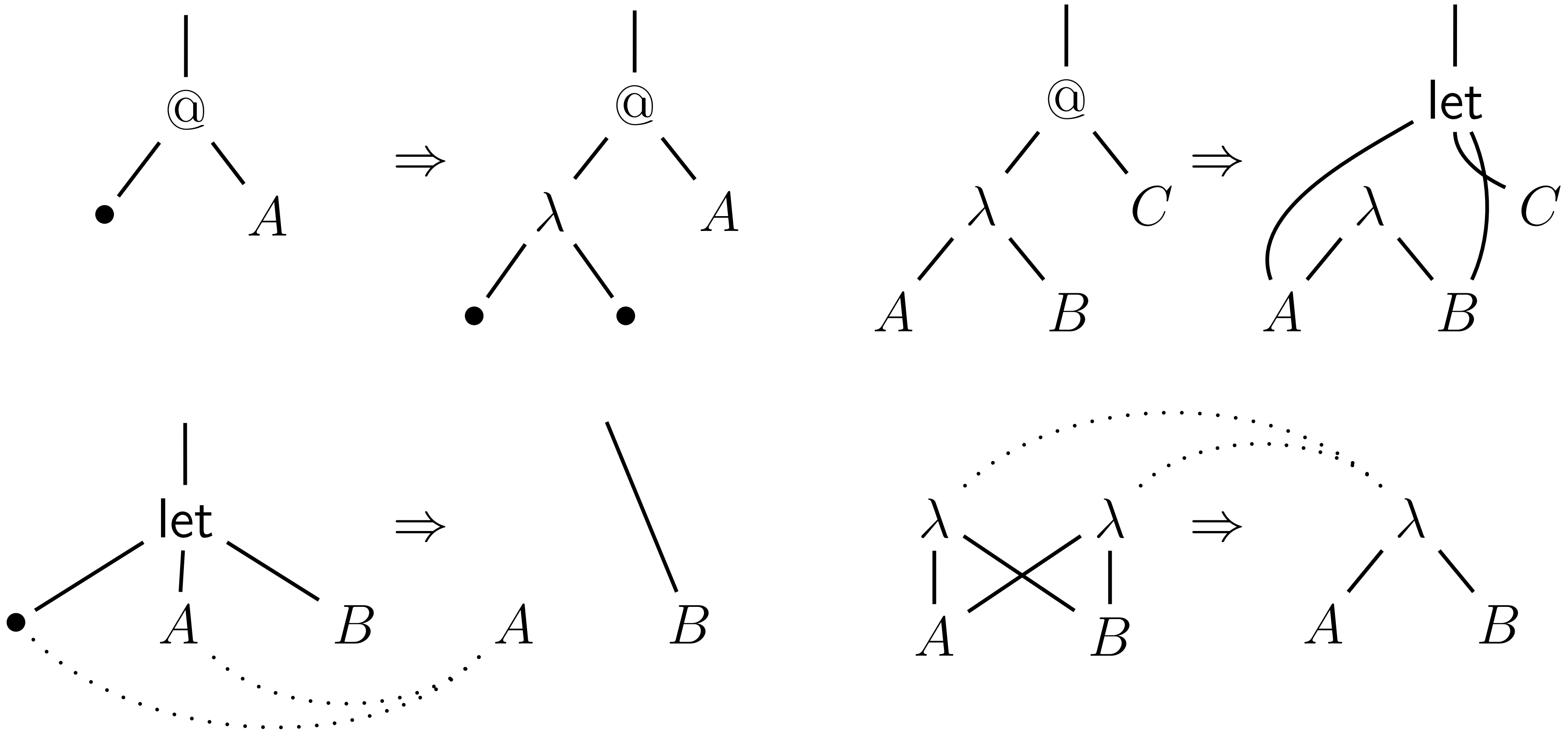


This is just another view of the same term. However, I will also say it is the term's *type*, just not yet normalized.

Some of the Type Rewriting Rules

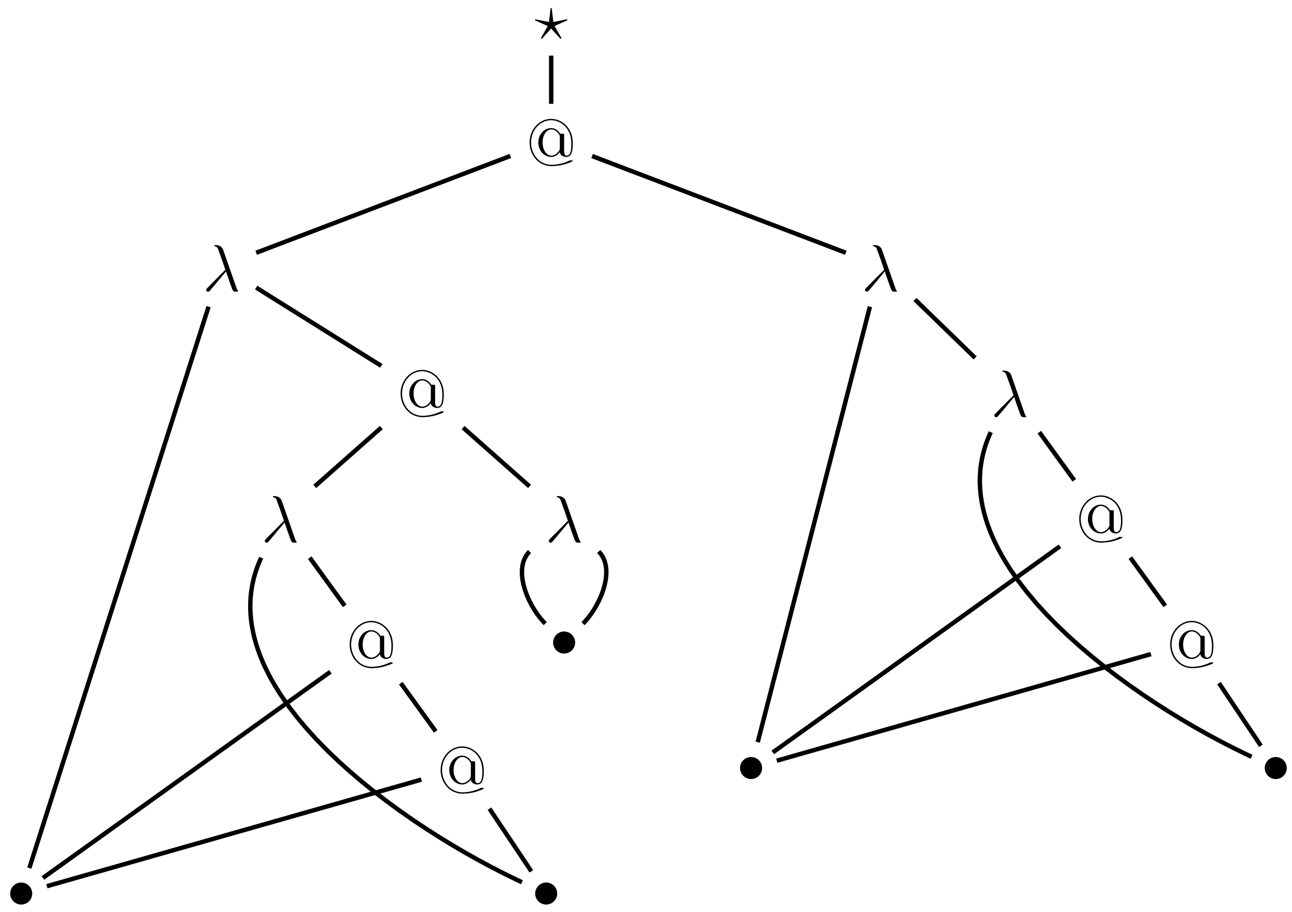


Some of the Type Rewriting Rules

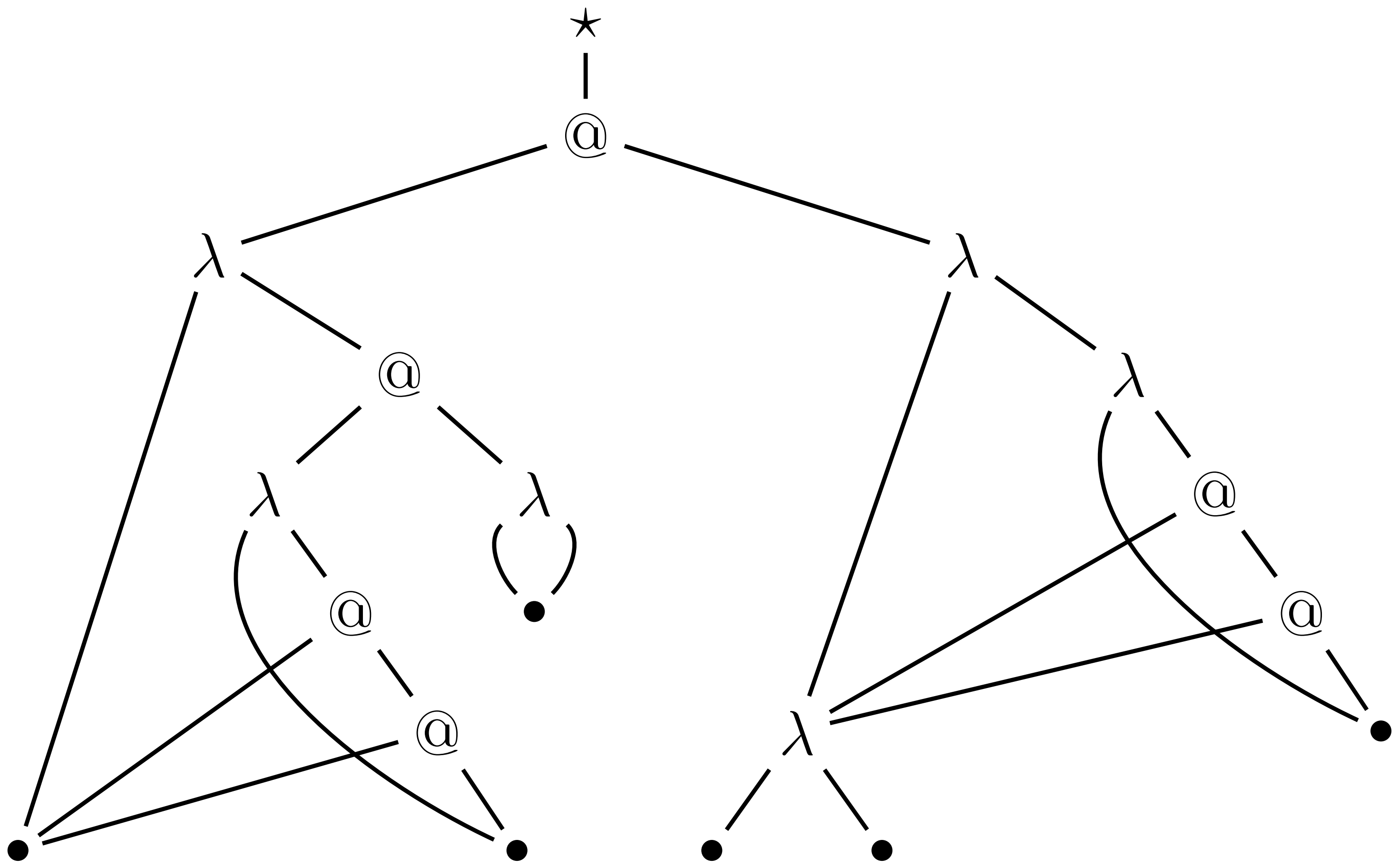


Some rules omitted. There are rules for garbage collection.
Formalism unverified, for discussion only.

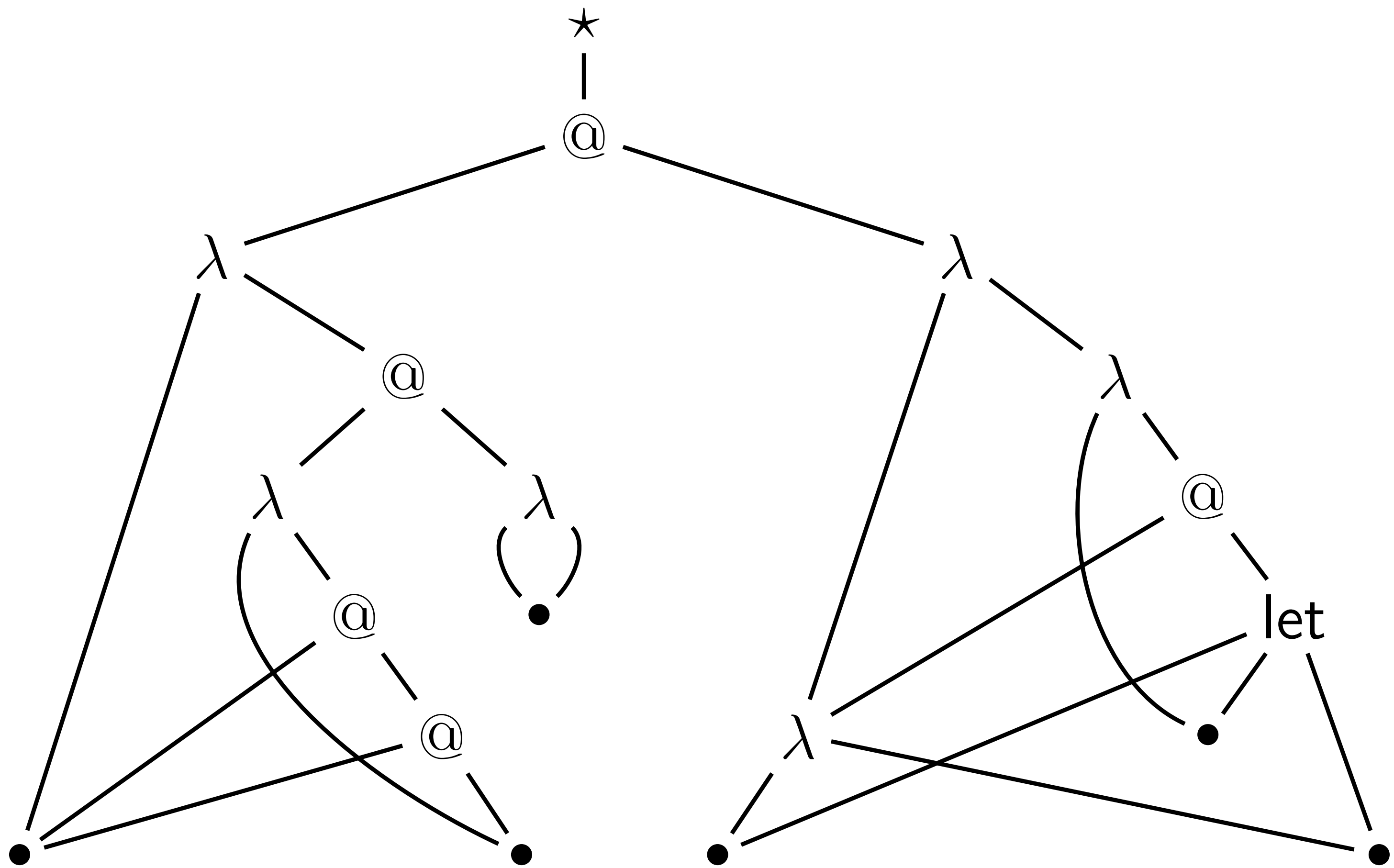
Normalizing the Term’s Type (1)



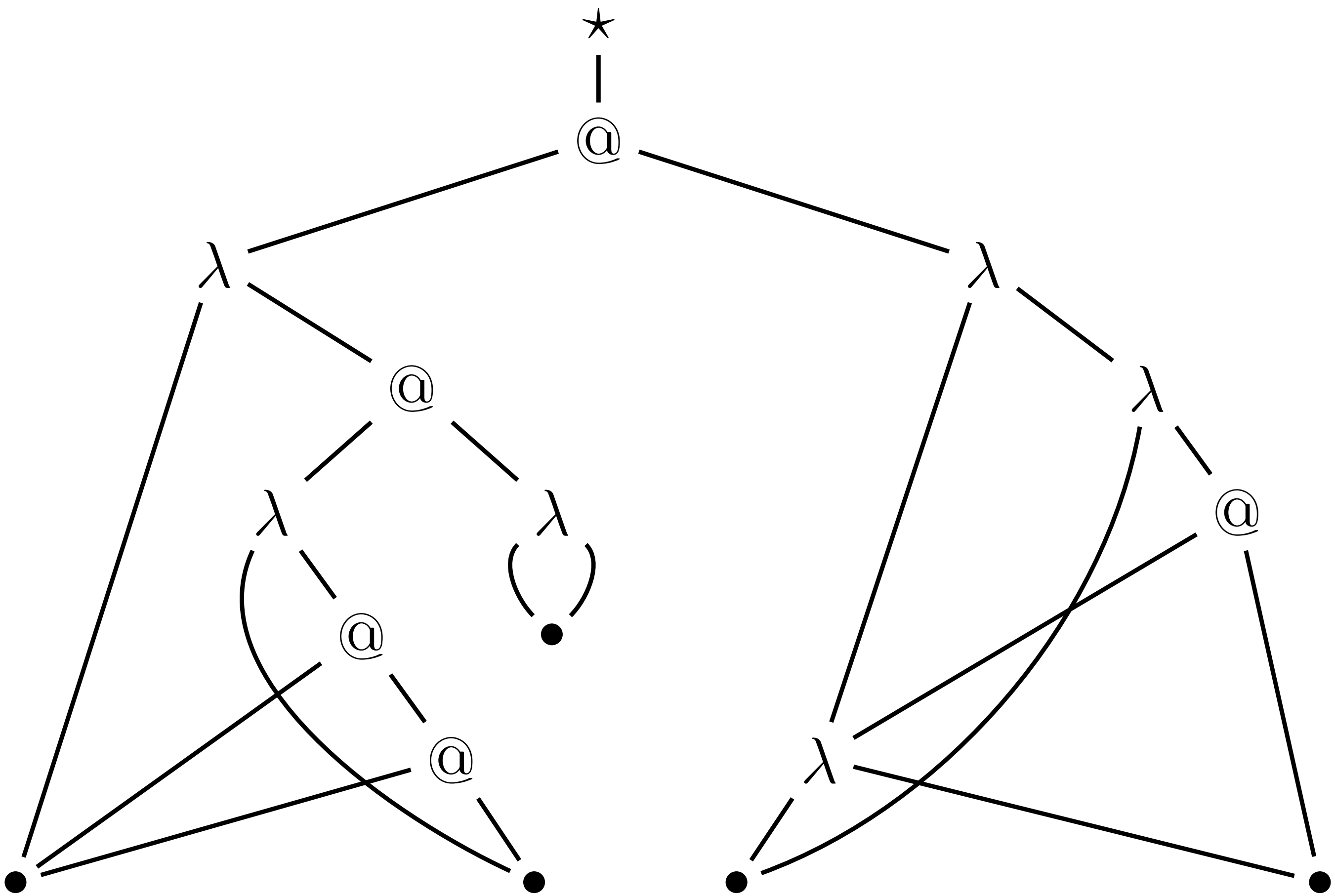
Normalizing the Term's Type (2)



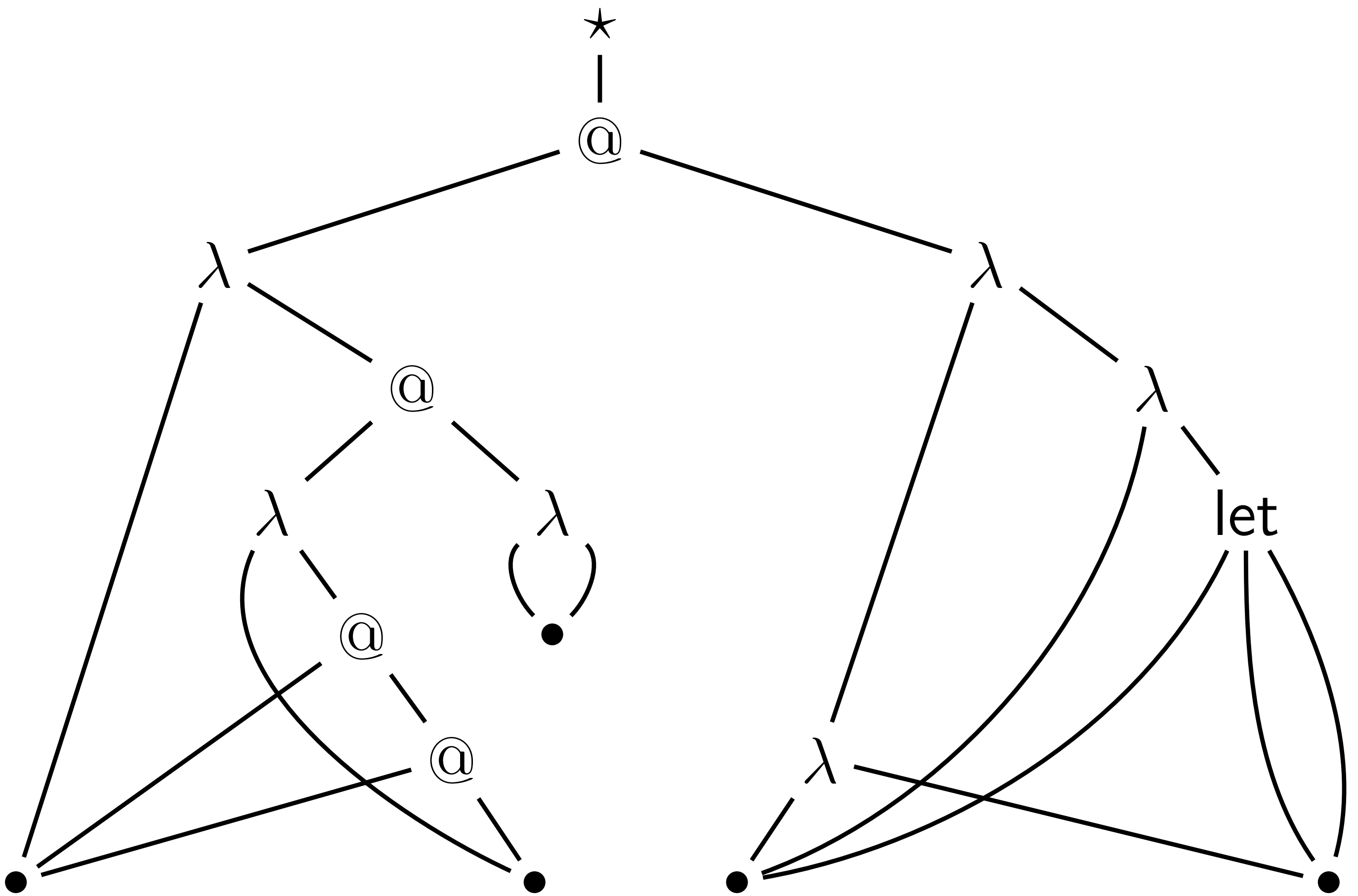
Normalizing the Term's Type (3)



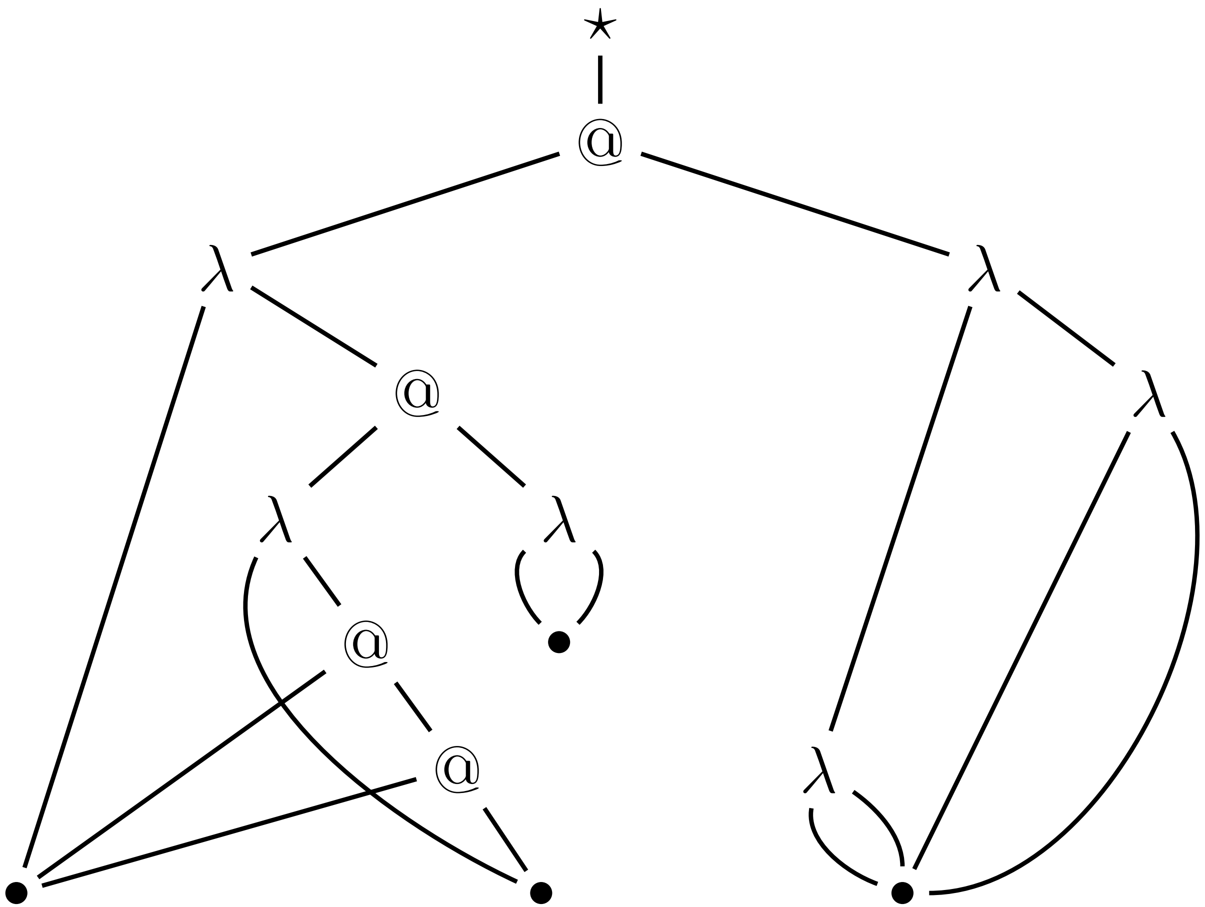
Normalizing the Term’s Type (4)



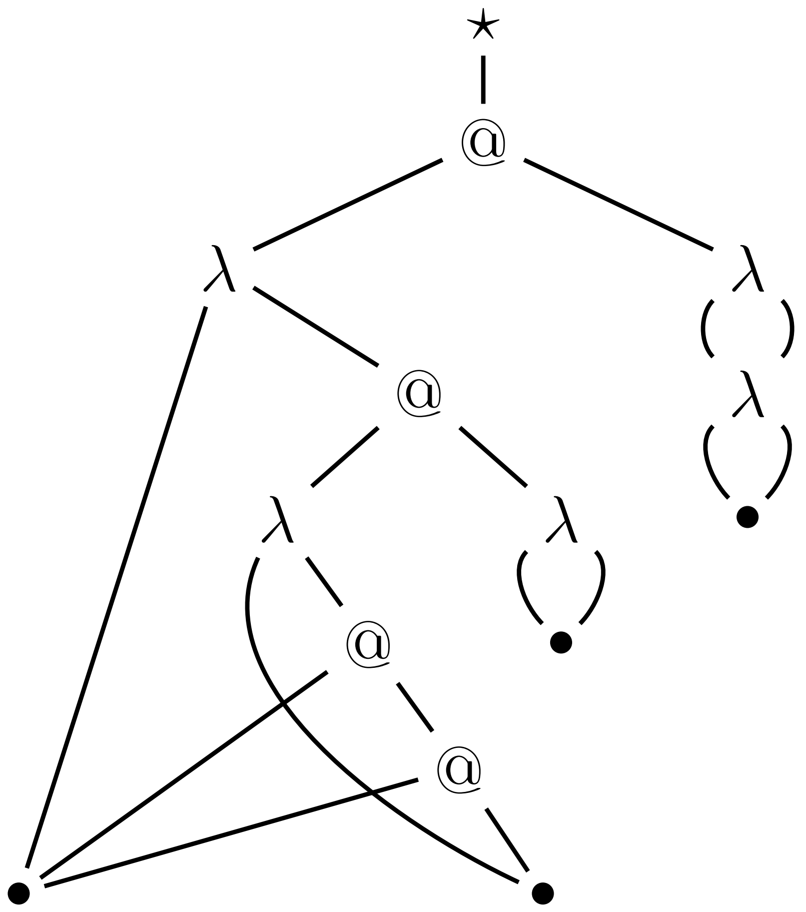
Normalizing the Term's Type (5)



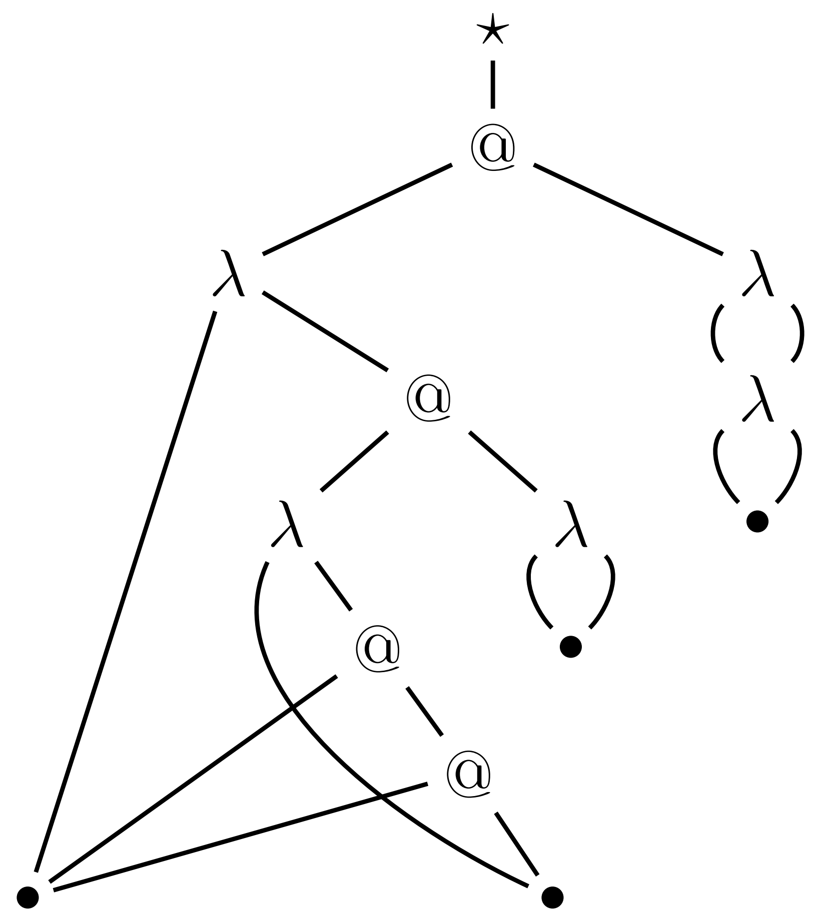
Normalizing the Term's Type (6)



Normalizing the Term's Type (7)

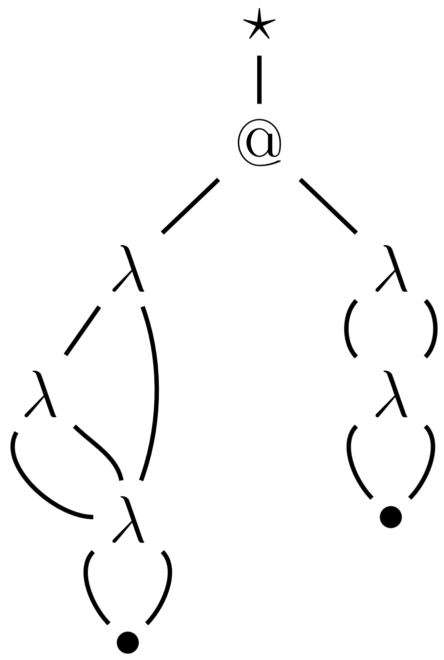


Normalizing the Term's Type (7)



Read the "λ" as "$\rightarrow$" to see a traditional principal typing for the right subterm.

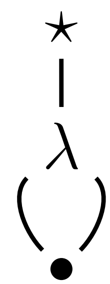
Normalizing the Term's Type (8)



In a number of additional steps, the left subterm's principal typing is found.

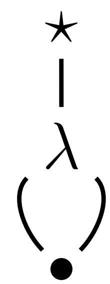
Normalizing the Term's Type (9)

- In a number of additional steps, the principal typing of the entire term is reached:



Normalizing the Term's Type (9)

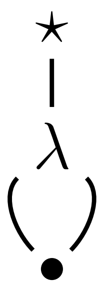
- In a number of additional steps, the principal typing of the entire term is reached:



- So type inference for simple types can be viewed as simply applying an unusual set of rewrite rules to the λ term.

Normalizing the Term's Type (9)

- In a number of additional steps, the principal typing of the entire term is reached:



- So type inference for simple types can be viewed as simply applying an unusual set of rewrite rules to the λ term.
- What about for more complex type systems? Now I will consider intersection types.

Overview

- Goals for this talk
- Type inference as rewriting via an example with simple types
- **Intersection types and why you might want them**
- Type inference as rewriting via an example with intersection types
- Various concluding remarks

Intersection Types

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

Intersection Types

- Type polymorphism by *listing* usage types [Coppo, Dezani-Ciancaglini, and Venneri, 1980].
- Why "intersection"? If semantic denotations $[\![\sigma]\!]$ and $[\![\tau]\!]$ are program fragment sets, then $[\![\sigma \cap \tau]\!] = [\![\sigma]\!] \cap [\![\tau]\!]$.

Intersection Types

- Type polymorphism by *listing* usage types [Coppo, Dezani-Ciancaglini, and Venneri, 1980].
- Why "intersection"? If semantic denotations $[\![\sigma]\!]$ and $[\![\tau]\!]$ are program fragment sets, then $[\![\sigma \cap \tau]\!] = [\![\sigma]\!] \cap [\![\tau]\!]$.
- Example comparing intersection and $\forall$-quantified types:

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

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

  Example is semantically like $\forall \alpha \in \{\mathtt{int}, \mathtt{bool}\} . \alpha \to \alpha$, but has significant practical differences.

Typability for Various Systems

F: System F.

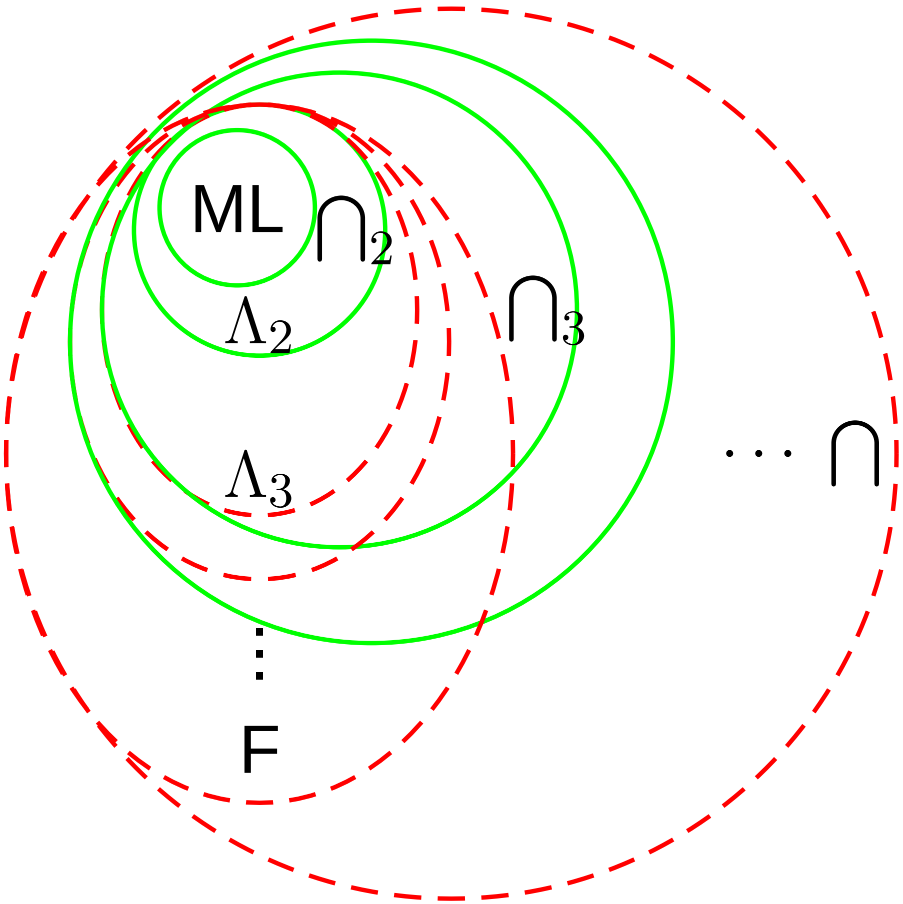
Λ_k: rank-k System F.

$\bigcap$: intersection types.

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

Decidable.

Undecidable.



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

Flexibility of Intersection Types

```
fun self_apply2 z  ⇒ (z z) z;
fun apply f x ⇒ f x;
fun reverse_apply y g ⇒ g y;
fun id w ⇒ w;
(self_apply2 apply not true,
 self_apply2 reverse_apply id false not);
```

- The example *safely* computes (false, true).
- Urzyczyn [1997] proved this example is not typable in F_ω, considered the most powerful type system with universal quantifiers.
- The example is typable in the rank-3 restriction of intersection types.

Overview

- Goals for this talk
- Type inference as rewriting via an example with simple types
- Intersection types and why you might want them
- **Type inference as rewriting via an example with intersection types**
- Various concluding remarks

Another Example

Consider this λ-term:

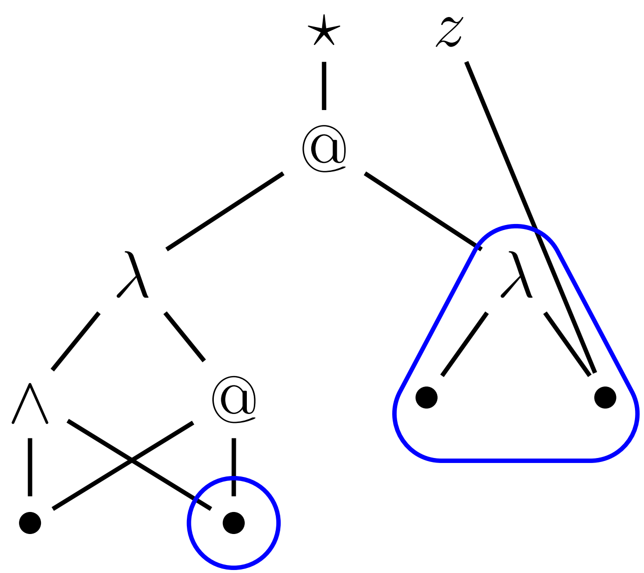
$$(\lambda x.xx)(\lambda y.z)$$

Another Example

Consider this λ-term:

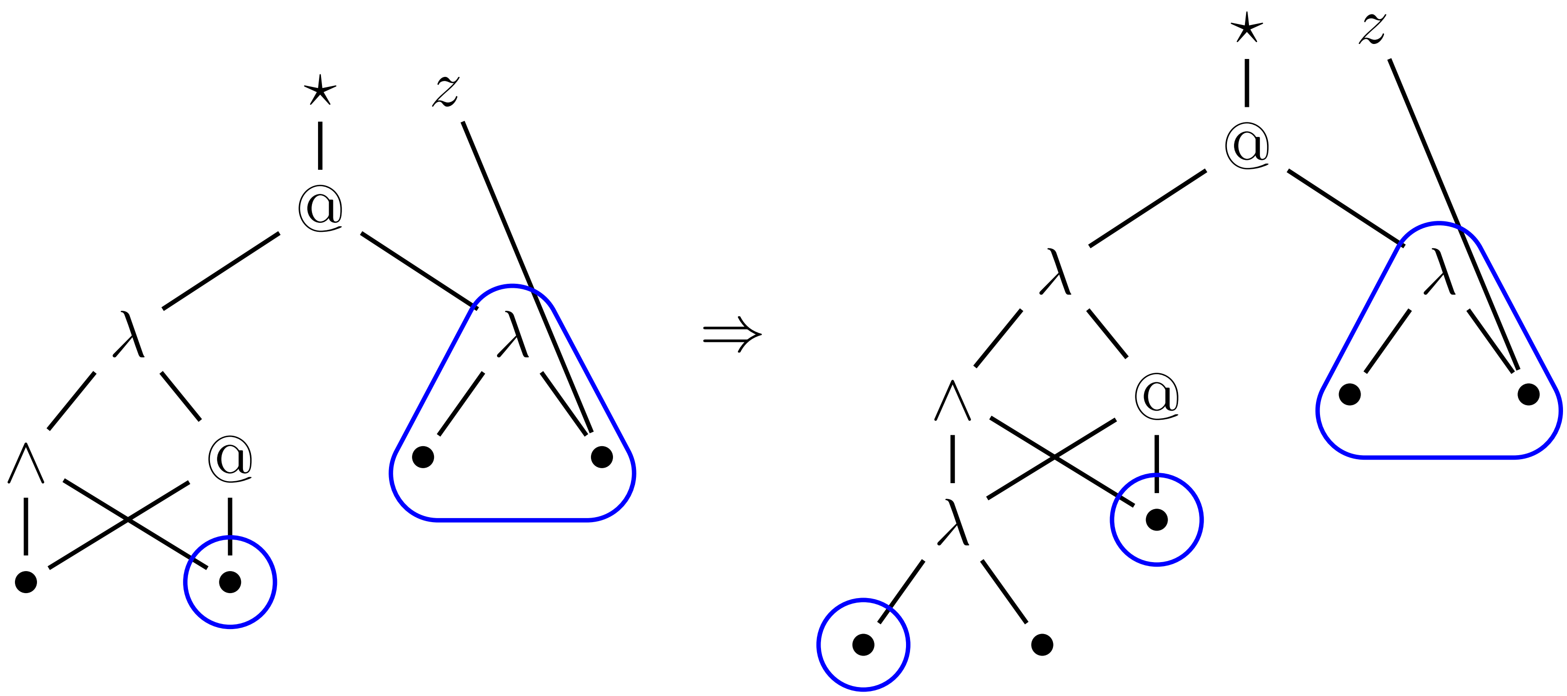
$$(\lambda x.xx)(\lambda y.z)$$

The DAG is formed a bit differently from before because intersection types are more flexible:

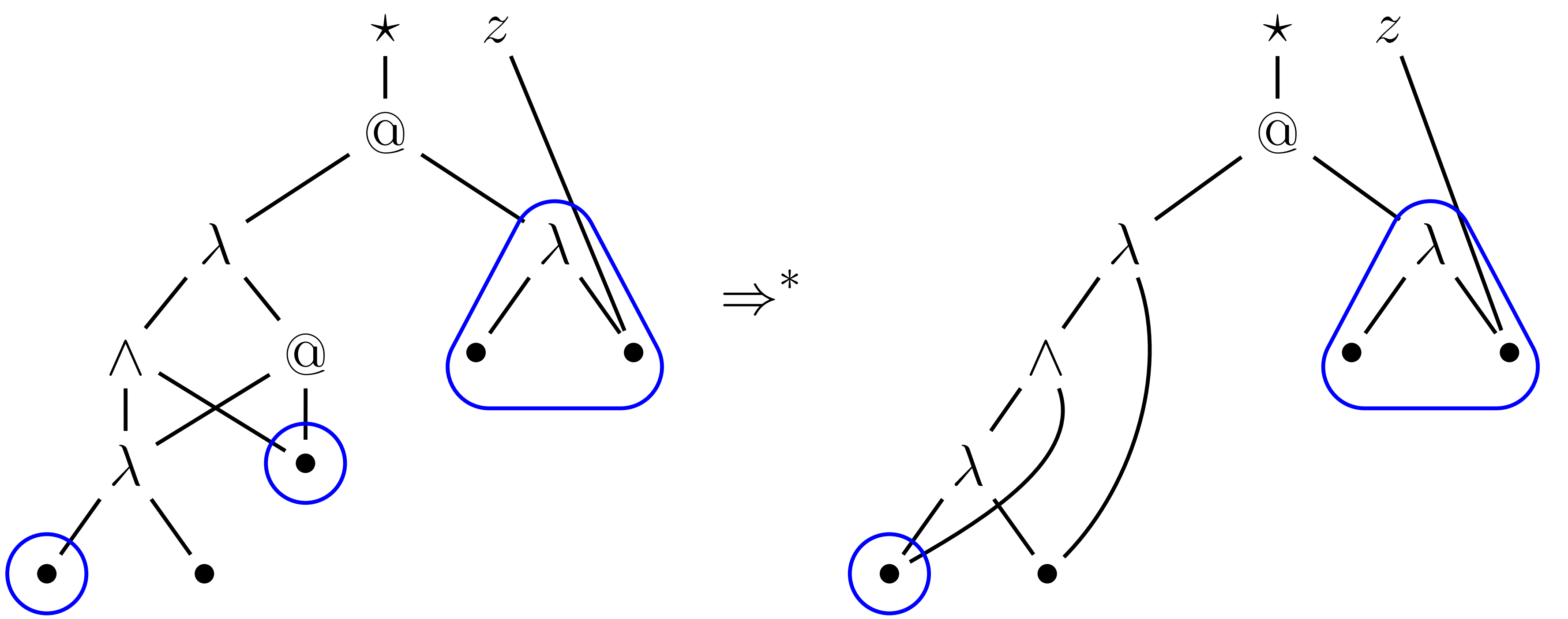


The colored boundaries correspond to *expansion variables* in System I.

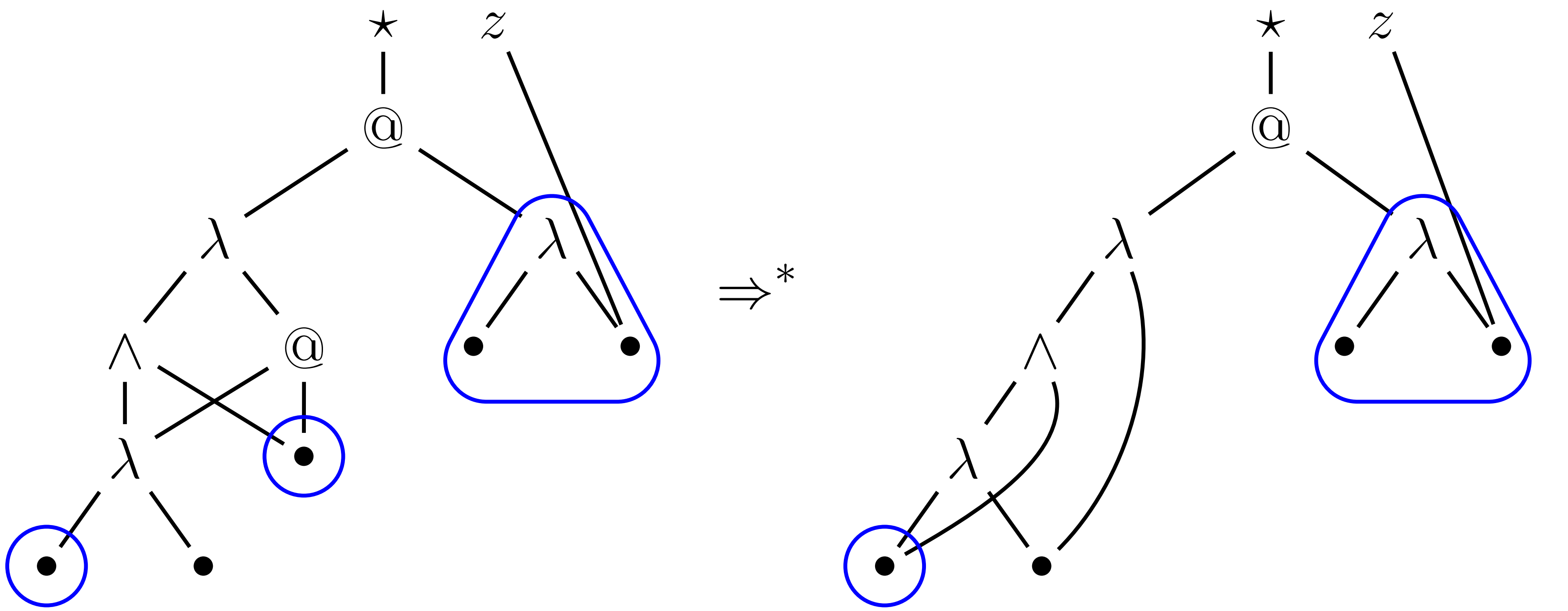
Normalizing the Typing (1)



Normalizing the Typing (2)



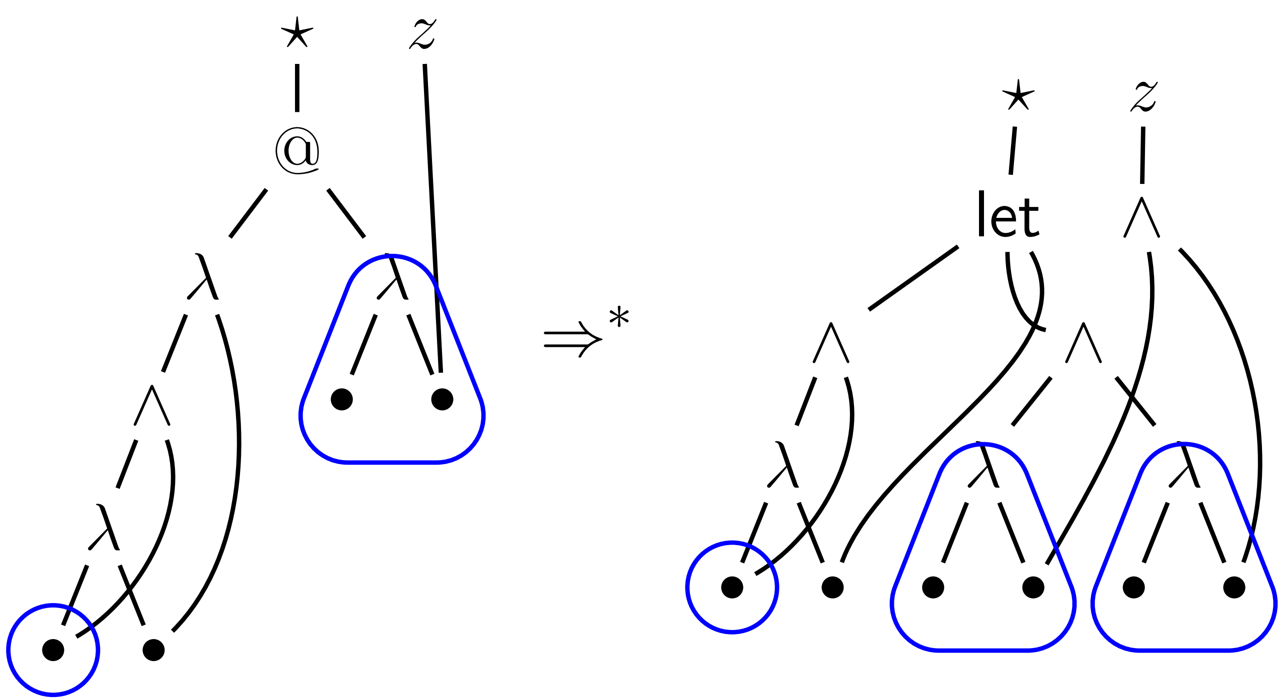
Normalizing the Typing (2)



The principal typing of $(\lambda x.xx)$ now appears on the left. The typing on the right is already the principal typing of $(\lambda y.z)$.

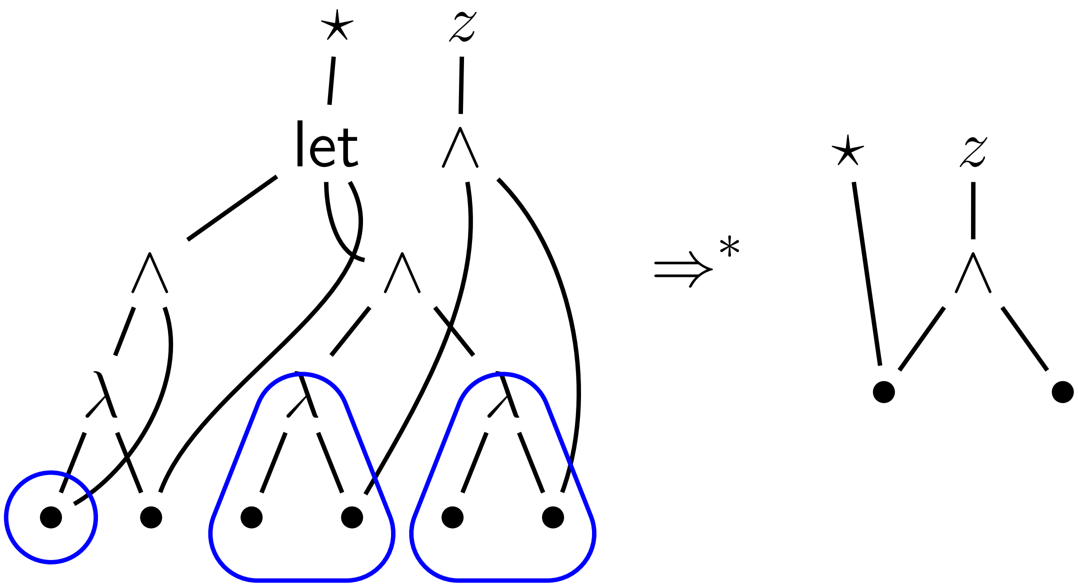
Normalizing the Typing (3)

In 2 more steps, the boundaries (expansion variables) play a role:

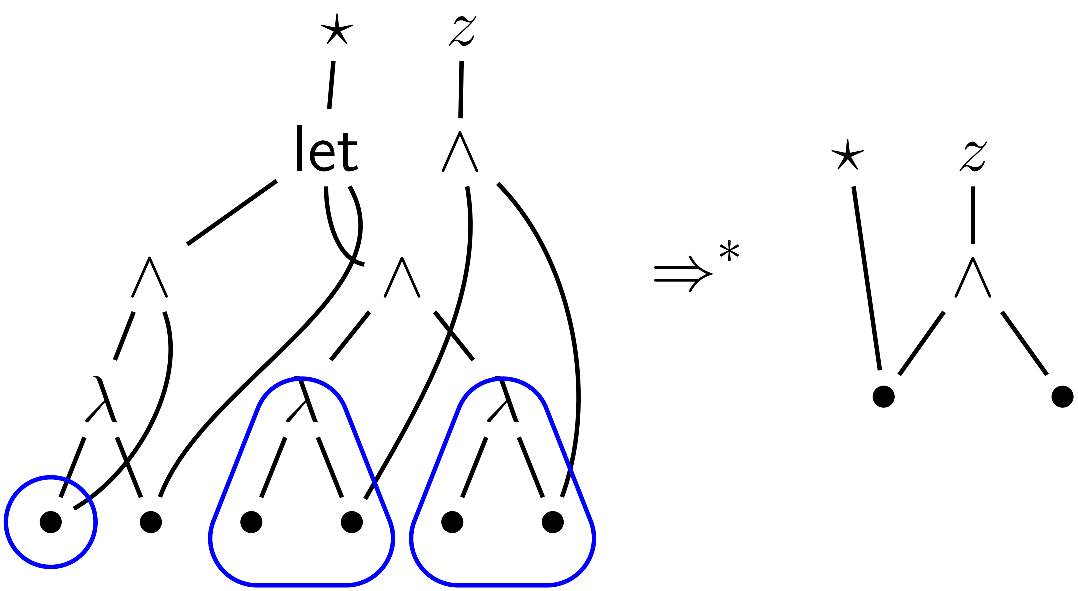


Only the contents of a boundary may be duplicated and then all incoming edges must be split with a $\wedge$ node connected to the corresponding nodes in the split copies.

Normalizing the Typing (4)

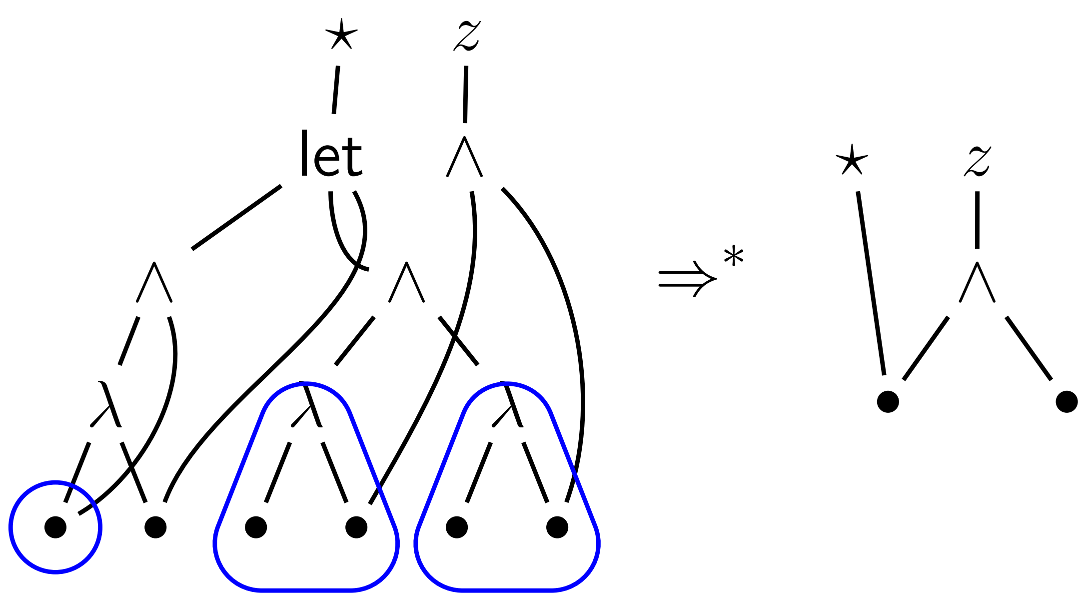


Normalizing the Typing (4)



- The typing's normal form indicates that the result is obtained from the variable z and that a copy of z is discarded in the process.

Normalizing the Typing (4)



- The typing's normal form indicates that the result is obtained from the variable z and that a copy of z is discarded in the process.
- Thus, a complex type inference problem is just applying a set of rewrite rules to the λ term.

Overview

- Goals for this talk
- Type inference as rewriting via an example with simple types
- Intersection types and why you might want them
- Type inference as rewriting via an example with intersection types
- **Various concluding remarks**

Future Work

- Formalize these ideas.

Future Work

- Formalize these ideas.
- Extend with many other real language features.

Future Work

- Formalize these ideas.
- Extend with many other real language features.
- Extend with conditional types.

Future Work

- Formalize these ideas.
- Extend with many other real language features.
- Extend with conditional types.
- Implement in a compiler.

Related Work in the Church Project

- Kfoury [1996, 2000]: “A linearization of the lambda calculus”.
- Kfoury [1999]: “Beta-reduction as unification”.
- Kfoury and Wells [1999]: “Principality and Decidable Type Inference for Finite-Rank Intersection Types”, the paper which introduced System I and its principal typing algorithm.
- Updated work on System I: Carlier [2002] and Kfoury, Washburn, and Wells [2002].
- Ongoing work extending expansion variables to work for more programming language features: tagged variants (usually handled with sum types), mutually recursive definitions, etc.

Conclusions

- For some type systems, typings can be viewed as just the result of normalizing the term using a particular set of rewrite rules.

Conclusions

- For some type systems, typings can be viewed as just the result of normalizing the term using a particular set of rewrite rules.
- These rules may yield results more abstract than those yielded by the usual evaluation rules.

Conclusions

- For some type systems, typings can be viewed as just the result of normalizing the term using a particular set of rewrite rules.
- These rules may yield results more abstract than those yielded by the usual evaluation rules.
- This may be able to give a clearer explanation of how some type systems work.

Conclusions

- For some type systems, typings can be viewed as just the result of normalizing the term using a particular set of rewrite rules.
- These rules may yield results more abstract than those yielded by the usual evaluation rules.
- This may be able to give a clearer explanation of how some type systems work.
- The close connection between rewriting and types is made more apparent.

References

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inform. & Comput.*, 119:202–230, 1995.

Sébastien Carlier. Polar type inference with intersection types and ω. In ITRS '02 ITRS '02. To appear as a volume of ENTCS.

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.

J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

ITRS '02. *Proceedings of the 2nd Workshop on Intersection Types and Related Systems*, 2002. To appear as a volume of ENTCS.

Assaf J. Kfoury. A linearization of the lambda-calculus. A refereed version is Kfoury [2000]. This version was presented at the Glasgow Int'l School on Type Theory & Term Rewriting, September 1996.

Assaf J. Kfoury. Beta-reduction as unification. In D. Niwinski, editor, *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996), Banach Center Publication, Volume 46*, pages 137–158. Springer-Verlag, 1999.

Assaf J. Kfoury. A linearization of the lambda-calculus. *J. Logic Comput.*, 10(3), 2000. Special issue on Type Theory and Term Rewriting. Kamareddine and Klop (editors).

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.

Assaf J. Kfoury, Geoff Washburn, and J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In ITRS '02 ITRS '02. To appear as a volume of ENTCS.

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.

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