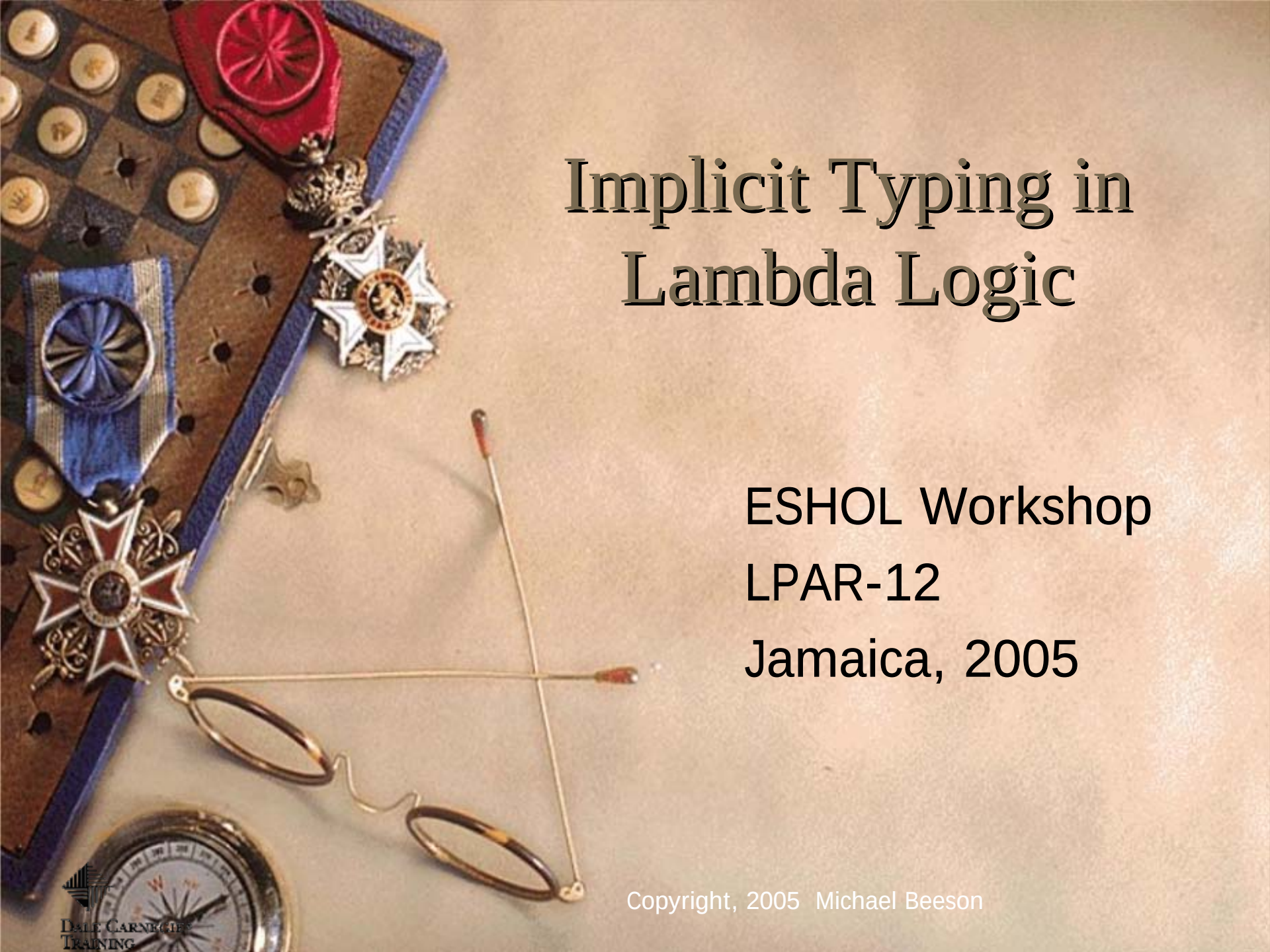

Implicit Typing in Lambda Logic

ESHOL Workshop

LPAR-12

Jamaica, 2005

Lambda Logic

- Combines lambda calculus and first-order logic (FOL). Has syntax of both.
- Ap(f,x) and lambda(x,t)
- Ap(lambda(x,t),q) = q[x:=t].
- An untyped system, unlike type theory, but like lambda calculus and FOL.
- Not first order because lambda terms can define functions and predicates.

Lambda Unification

- If substitution S makes tS and qS provably equivalent in lambda logic then S is called a lambda unifier of t and q.
- Lambda unification is a new algorithm that finds lambda unifiers.
- Not the same as ``higher-order unification''.
- Not the same as first-order unification.

To unify *Ap(X,w)* with *t*

- Pick a "masking subterm" *q* of *t*. It must contain all occurrences of *X* in *t*. If there are none, it can be any subterm of *t*.
- Unify *q* with *w*, producing substitution *A*.
- If *qA* occurs more than once in *tA*, pick a subset *S* of these occurrences. If *x* occurs in *q* then *S* must be *all* occurrences.
- Let *z* be a fresh variable. Substitute *z* in *tA* for each occurrence of *qA* in the set S. Call the result *r*.
- Return the substitution *A*, *X:=lambda(z,r)*.

Otter-lambda

- Theorem prover based on Otter 3.2
- Partially implements lambda unification
- Lambda unification used in resolution, paramodulation, demodulation, and factoring.
- Some advantages of higher order systems
- All the advantages of a modern first-order prover.

Lambda Unification

- Clauses like Robinson unification
- Clauses for alpha-equivalence
- Heart of the matter: to unify

  Ap(X,w) with *t*

  getting a lambda-term for the value of the variable *X*. Here *w* and *t* are terms.

The no-nilpotents example

- Integral domain: ring in which $xy=0$ implies $x = 0$ or $y = 0$.
- No nilpotents: $x^n = 0$ implies $x = 0$.
- n is a natural number, x is in the ring.
- Do we need unary predicates $N(x)$ and $R(x)$ to formalize this problem?
- No!

Induction in clausal form

- $-Ap(X,0) \mid Ap(X,g(X)) \mid Ap(X,w)$.
- $-Ap(X,0) \mid -Ap(X,s(g(X)) \mid Ap(X,w)$.

To prove $P(z)$ by induction on z, we unify $Ap(X,w)$ with $P(z)$, getting

$X := lambda(z, P(z))$.

- We then prove the base case $P(0)$ and resolution leaves us with the induction hypothesis $Ap(X,g(X))$ and the negated induction step $-Ap(X,s(g(X))$.

Induction and Resolution

- Otter can already solve the no-nilpotents problem if we *give it the right instance of induction*.
- Otter-lambda can *find the right instance of induction* using lambda unification, being given only the Peano axioms.

Implicit Typing

- Do not type variables
- Predicates and function symbols get their parameter types and value types specified.
- type(R,pow(R,N)) says that pow takes a ring argument and a natural number argument and returns a ring argument.
- type(o,N); type(0,R); type(1,R); type(R,*(R,R)); type(R,+(R,R)); type(N,s(N)).

Theorem

- If the axioms are correctly typeable then the conclusions are correctly typeable.
- Applies to resolution, factoring, paramodulation (not from or into variables), and demodulation.
- For FOL, perhaps due to Wick and McCune, or to folklore.
- We want a version of this theorem that applies to some version of lambda logic and some version of Otter-lambda.

Fixed points

- Unify *Ap*(*x*,*w*) with *f*(*Ap*(*x*,*w*))
- Masking subterm *q* is just *x*. Unifying *q* with *w* we get *w*:=*x* as the substitution *A*. There are two occurrences of *qA* in the right hand side; we get

  X := lambda(*z*,*f*(*Ap*(*z*,*z*)))
- This is Church's fixed-point construction, an automatic consequence of lambda unification.

Untyped deduction in lambda logic:

- The fixed point deduction cannot be correctly typed, unless we have a type T such that $T = i(T,T)$.
- In particular not if types are given by constant terms, such as in finite type theory.

Fixed points in Otter-lambda

- To get Otter-lambda to deduce the existence of fixed points, we must enter a negated goal that is not correctly typeable.
- Thus an implicit typing theorem is not ruled out by the fact that lambda logic can make untypeable inferences.

Another example

- Write down the axioms of group theory in lambda logic using $i(x)$ for inverse and * for the group operation.
- Fix c, and let $H(f,x) = c^* Ap(f,x)$.
- Choose a fixed point f , so $Ap(f,x) = H(f,x)$.
- Then $Ap(f,x) = c^* Ap(f,x)$.
- Hence c is the group identity. Since c was arbitrary, any two objects are equal.
- That is a contradiction in lambda logic, which postulates the existence of two distinct objects.

What does this mean?

- Semantically: there is no way to turn a lambda model into a group.
- The axioms are well typed.
- We can’t get a contradiction unless we add an untypeable formula in the input file.
- In view of our implicit typing theorem, it’s OK to go ahead and reason about groups and subgroups without explicit typing.

Hypothesis of the implicit typing theorem

- Every formula in the input file is typeable according to some *coherent list of type specifications*. That means, each *f* or *P* gets a unique type specification (for each fixed arity), except *Ap* is allowed one with value type *Prop* and one with another value type, and there are some conditions on the types of *Ap* and *lambda*:

- Although *Ap* can have two type specifications, they must have the form

  type(*V*, *Ap*(*i*(*U*,*V*),*U*)

where the "ground type" *U* is the same.

- type(*i*(*X*,*Y*),lambda(*X*,*Y*)) is in the list if and only if
- type(*Y*,*Ap*(*i*(*X*,*Y*),*X*)) is in the list

A consequence of coherent typing

- If a clause is correctly typed by a coherent list of type specifications then each variable in the clause gets a unique type (at all occurrences).

Type-safe lambda unification

- A particular lambda unification is called type-safe, with respect to a particular typing, if when unifying $Ap(X,w)$ with t, the masking term q always has the same type as w.
- Type-safe lambda unification preserves correct typing.

Implicit Typing in Lambda Logic

- If the axioms are correctly typed by a coherent list of type specifications, and
- If only type-safe lambda unification is performed, then
- resolution, factoring, demodulation, and paramodulation (but not from or into variables) lead to correctly typed conclusions.

Implementing type-safe lambda unification

- We must restrict the choice of the masking subterm *q*. If the (unique) type of the second arguments of *Ap* is called the "ground type", then we must ensure that the masking subterm has ground type.

Choosing a masking subterm

Choose either

- A term that occurs as a second argument of Ap (as a subterm of *t*) or
- A constant, and require all constants to be of the same type in the given implicit typing.
- This last is not a serious restriction as we can always replace other constants by terms *h(c)* using a fresh function symbol *h*.

Explicit typing

- Of course, it is a simple matter to allow an input flag *set(types)* and a part of the input file that contains

list(types)

type(N,s(N)).

*type(R, *(R,R)).*

end_of_list.

Then the type of any candidate masking term can just be looked up. But we have not found it necessary to do this. Many interesting examples have been proved with the scheme on the previous slide.

Implicit Typing in Otter-lambda

- When Otter-lambda uses only type-safe lambda unification, and the input file is correctly typed using some coherent list of type specifications, then
- Any proof produced is guaranteed *a priori* to also be correctly typeable.
- Thus we need not build a type-checker to check proofs *ex post facto*, or rely on hand verification that proofs are correctly typed.