Putting Names to Work

Scrap your Nameplate
Model-check your Metatheory

James Cheney
University of Edinburgh
TU Munich
February 7, 2007
Outline

- Scrap your nameplate:
  - Using Haskell-style type classes and generic programming to define substitution, FVs *once and for all*

- Metatheory modelchecking:
  - Using logic programming proof search to look for “shallow” bugs in core language/type system/operational semantics specifications.

- Will assume familiarity with nominal “stuff” (swapping-based definition of $\alpha$-equivalence, etc.)
Scrap your Nameplate
What is *nameplate*?

- I am using the term to refer to things like capture-avoiding substitution, free variables, etc.
- For clean core languages like $\lambda$, such definitions seem trivial.
- But for any realistic language, the number of cases needed is proportional to the number of language cases * number of things you can substitute for.
- So you need to write a lot of boring code before you even start to program with or reason about definitions.
- Let’s look at some examples.
let rec apply_s s t =
  let h = apply_s s in
  match t with
  | Name a -> Name a
  | Abs (a,e) -> Abs(a, h e)
  | App(c,es) -> App(c, List.map h es)
  | Susp(p,vs,x) -> (match lookup s x with
     Some tm -> apply_p p tm
     | None -> Susp(p,vs,x))
;;

let rec apply_s_g s g =
  let h1 = apply_s_g s g in
  let h2 = apply_s_p s in
match g with
  Gtrue -> Gtrue
| Gatonic(t) -> Gatonic(apply_s s t)
| Gand(g1,g2) -> Gand(h1 g1, h1 g2)
| Gor(g1,g2) -> Gor(h1 g1, h1 g2)
| Gforall(x,g) ->
    let x’ = Var.rename x in
    Gforall(x’, apply_s_g (join x (Susp(Perm.id,Univ,x’)) s) g)
| Gnew(x,g) ->
    let x’ = Var.rename x in
    Gnew(x, apply_p_g (Perm.trans x x’) g)
| Gexists(x,g) -> 
  
  let x' = Var.rename x in 
  
  Gexists(x', apply_s_g (join x (Susp(Perm.id,Univ,x'))) s) g |
| Gimplies(d,g) -> Gimplies(h2 d, h1 g) |
| Gfresh(t1,t2) -> Gfresh(apply_s s t1, apply_s s t2) |
| Gequals(t1,t2) -> Gequals(apply_s s t1, apply_s s t2) |
| Geunify(t1,t2) -> Geunify(apply_s s t1, apply_s s t2) |
| Gis(t1,t2) -> Gis(apply_s s t1, apply_s s t2) |
| Gcut -> Gcut |
| Guard (g1,g2,g3) -> Guard(h1 g1, h1 g2, h1 g3) |
| Gnot(g) -> Gnot(h1 g) |
and apply_s_p s p =
  let h1 = apply_s_g s in
  let h2 = apply_s_p s in
  match p with
  | Dtrue -> Dtrue
  | Datomic(t) -> Datomic(apply_s s t)
  | Dimplies(g,t) -> Dimplies(h1 g, h2 t)
  | Dforall (x,p) ->
    let x' = Var.rename x in
    Dforall (x', apply_s_p (join x (Susp(Perm.id,Univ,x')) s) p)
  | Dand(p1,p2) -> Dand(h2 p1,h2 p2)
  | Dnew(a,p) ->
    let a' = Var.rename a in
    Dnew(a, apply_p_p (Perm.trans a a') p)
;;
let tymap onvar c tyT =
  let rec walk c tyT = match tyT with
    TyId(b) as tyT -> tyT
  | TyVar(x,n) -> onvar c x n
  | TyArr(tyT1,tyT2) -> TyArr(walk c tyT1,walk c tyT2)
  | TyBool -> TyBool
  | TyTop -> TyTop
  | TyBot -> TyBot
  | TyRecord(fieldtys) -> TyRecord(List.map (fun (li,tyTi) -> (li, walk c tyTi)) fieldtys)
  | TyVariant(fieldtys) -> TyVariant(List.map (fun (li,tyTi) -> (li, walk c tyTi)) fieldtys)
  | TyFloat -> TyFloat
| TyString  -> TyString  |
| TyUnit    -> TyUnit    |
| TyAll(tyX,tyT1,tyT2) -> TyAll(tyX,walk c tyT1,walk (c+1) tyT2) |
| TyNat     -> TyNat     |
| TySome(tyX,tyT1,tyT2) -> TySome(tyX,walk c tyT1,walk (c+1) tyT2) |
| TyAbs(tyX,knK1,tyT2) -> TyAbs(tyX,knK1,walk (c+1) tyT2) |
| TyApp(tyT1,tyT2)    -> TyApp(walk c tyT1,walk c tyT2)  |
| TyRef(tyT1)         -> TyRef(walk c tyT1)               |
| TySource(tyT1)      -> TySource(walk c tyT1)            |
| TySink(tyT1)        -> TySink(walk c tyT1)               |

in walk c tyT
let tmmap onvar ontype c t =
  let rec walk c t = match t with
    TmVar(fi,x,n) -> onvar fi c x n
  | TmAbs(fi,x,tyT1,t2) -> TmAbs(fi,x,ontype c tyT1,walk (c+1) t2)
  | TmApp(fi,t1,t2) -> TmApp(fi,walk c t1,walk c t2)
  | TmTrue(fi) as t -> t
  | TmFalse(fi) as t -> t
  | TmIf(fi,t1,t2,t3) -> TmIf(fi,walk c t1,walk c t2,walk c t3)
  | TmProj(fi,t1,l) -> TmProj(fi,walk c t1,l)
  | TmRecord(fi,fields) -> TmRecord(fi,List.map (fun (li,ti) ->
                                              (li,walk c ti))
                                      fields)
TmLet(fi, x, t1, t2) -> TmLet(fi, x, walk c t1, walk (c+1) t2)
TmFloat _ as t -> t
TmTimesfloat(fi, t1, t2) -> TmTimesfloat(fi, walk c t1, walk c t2)
TmAscribe(fi, t1, tyT1) -> TmAscribe(fi, walk c t1, ontype c tyT1)
TmInert(fi, tyT) -> TmInert(fi, ontype c tyT)
TmFix(fi, t1) -> TmFix(fi, walk c t1)
TmTag(fi, l, t1, tyT) -> TmTag(fi, l, walk c t1, ontype c tyT)
TmCase(fi, t, cases) ->
  TmCase(fi, walk c t, List.map (fun (li, (xi, ti)) -> (li, (xi, walk (c+1) ti))) cases)
TmString _ as t -> t
TmUnit(fi) as t -> t
<table>
<thead>
<tr>
<th>TmLoc(fi,l) as t -&gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>TmRef(fi,t1) -&gt; TmRef(fi,walk c t1)</td>
</tr>
<tr>
<td>TmDeref(fi,t1) -&gt; TmDeref(fi,walk c t1)</td>
</tr>
<tr>
<td>TmAssign(fi,t1,t2) -&gt; TmAssign(fi,walk c t1,walk c t2)</td>
</tr>
<tr>
<td>TmError(_) as t -&gt; t</td>
</tr>
<tr>
<td>TmTry(fi,t1,t2) -&gt; TmTry(fi,walk c t1,walk c t2)</td>
</tr>
<tr>
<td>TmTAbs(fi,tyX,tyT1,t2) -&gt;</td>
</tr>
<tr>
<td>TmTAbs(fi,tyX,ontype c tyT1,walk (c+1) t2)</td>
</tr>
<tr>
<td>TmTApp(fi,t1,tyT2) -&gt; TmTApp(fi,walk c t1,ontype c tyT2)</td>
</tr>
<tr>
<td>TmZero(fi) -&gt; TmZero(fi)</td>
</tr>
<tr>
<td>TmSucc(fi,t1) -&gt; TmSucc(fi, walk c t1)</td>
</tr>
<tr>
<td>TmPred(fi,t1) -&gt; TmPred(fi, walk c t1)</td>
</tr>
<tr>
<td>TmIsZero(fi,t1) -&gt; TmIsZero(fi, walk c t1)</td>
</tr>
<tr>
<td>TmPack(fi,tyT1,t2,tyT3) -&gt; TmPack(fi,ontype c tyT1,walk c t2,ontype c tyT3)</td>
</tr>
<tr>
<td>TmUnpack(fi,tyX,x,t1,t2) -&gt; TmUnpack(fi,tyX,x,walk c t1,walk (c+2) t2)</td>
</tr>
<tr>
<td>in walk c t</td>
</tr>
</tbody>
</table>

let typeShiftAbove d c tyT =
  tymap
  (fun c x n -> if x>=c then TyVar(x+d,n+d) else TyVar(x,n+d))
  c tyT
let termShiftAbove d c t = 
  tmmap
  (fun fi c x n -> if x>=c then TmVar(fi,x+d,n+d)
    else TmVar(fi,x,n+d))
  (typeShiftAbove d)
  c t

let termShift d t = termShiftAbove d 0 t

let typeShift d tyT = typeShiftAbove d 0 tyT
let bindingshift d bind =
  match bind with
  NameBind -> NameBind
  | TyVarBind(tyS) -> TyVarBind(typeShift d tyS)
  | VarBind(tyT) -> VarBind(typeShift d tyT)
  | TyAbbBind(tyT, opt) -> TyAbbBind(typeShift d tyT, opt)
  | TmAbbBind(t, tyT_opt) ->
    let tyT_opt' = match tyT_opt with
      None -> None
      | Some(tyT) -> Some(typeShift d tyT)
    in
    TmAbbBind(termShift d t, tyT_opt')
(* Substitution *)

let termSubst j s t =
  tmmap
    (fun fi j x n -> if x=j then termShift j s else TmVar(fi,x,n))
  (fun j tyT -> tyT)
  j t

let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
let typeSubst tyS j tyT =
  tymap
  (fun j x n -> if x=j then (typeShift j tyS) else (TyVar(x,n)))
  j tyT

let typeSubstTop tyS tyT =
  typeShift (-1) (typeSubst (typeShift 1 tyS) 0 tyT)

let rec tytermSubst tyS j t =
  tmmap (fun fi c x n -> TmVar(fi,x,n))
  (fun j tyT -> typeSubst tyS j tyT) j t

let tytermSubstTop tyS t =
  termShift (-1) (tytermSubst (typeShift 1 tyS) 0 t)
What is *nameplate*?

*Nameplate* (n.) — boilerplate having to do with $\alpha$-renaming, capture-avoiding substitution, free variables, and other “mostly generic” traversals of datatypes with names.
What is nameplate?

- **Nameplate** (n.) — boilerplate having to do with $\alpha$-renaming, capture-avoiding substitution, free variables, and other “mostly generic” traversals of datatypes with names.
- Nominal techniques nicely handle programming (recursion) and reasoning (induction) over syntax modulo $\equiv_\alpha$.
- But (in contrast to HOAS) they do *not* provide built-in capture-avoiding substitution.
What is *nameplate*?

- *Nameplate* (n.) — boilerplate having to do with $\alpha$-renaming, capture-avoiding substitution, free variables, and other “mostly generic” traversals of datatypes with names.
- Nominal techniques nicely handle programming (recursion) and reasoning (induction) over syntax modulo $\equiv_\alpha$.
- But (in contrast to HOAS) they do *not* provide built-in capture-avoiding substitution.
- Can we have both?
Substitution without binding is generic

For syntax trees without binding, subst and FVs are essentially “fold”, most of whose cases are boring.

data Exp = Var Name
  | Plus Exp Exp
  | ...

subst a t (Var b) | a == b = t
subst a t (Var b) | otherwise = Var b
subst a t (Plus e1 e2) = Plus (subst a t e1) (subst a t e2)

These functions are prime examples of scrap your boilerplate-style generic traversals [Peyton Jones and Lämmel 2003, 2004, 2005]

Thus, prime candidates for boilerplate-scrapping
What goes wrong?

As soon as we add binding syntax, this nice structure disappears!

```haskell
data Exp = ... | Lam Name Exp
instance Monad M where ...
fresh :: M Name
rename :: Name -> Name -> Exp -> M Exp
subst :: Name -> Exp -> Exp -> M Exp
subst a t (Var b) | a == b = return t
subst a t (Var b) = return (Var b)
subst a t (Lam b e) =
    do b' <- fresh
       e' <- rename b b' e
       e'' <- subst a t e'
       return (Lam b' e'')
```
The real problem

- As soon as we add binding syntax, this nice structure disappears!
- Because
  - We need to know how to safely rename bound names to fresh ones
  - That means we need to generate fresh names
  - and need to know which names are bound
- This makes CAS much trickier to implement generically.
- And things get even worse when there are multiple datatypes involved, each with variables (e.g., types, terms, kinds).
Is there another way?

- Using the Gabbay-Pitts/FreshML approach (which I refer to as *nominal abstract syntax*), substitution and FVs are much better behaved.

- Starting point: much of the functionality of FreshML can be provided within Haskell using a class library (folklore)

- Use Lämmel-Peyton Jones “scrap your boilerplate” style of generic programming to provide instances *automatically* (including substitution, FVs)

- Claim: Users can use it without having to understand how it works.
Our approach

- First, observe that we can factor the code as follows:

  ```haskell
  data Abs a t = Abs a t
  data Exp    = ... | Lam (Abs Name Exp)
  subst_abs subst a t (Abs b e)
        = do b' <- fresh
           e'  <- rename b b' e
           e'' <- subst a t e'
           return (Abs b' e'')
  subst a t (Lam e) = do e' <- subst_abs subst a t e
                         return (Lam e')
  ```

- Note: we do the *same work* as the naive version, but the cases involving name-binding are handled by an “abstraction” type constructor and written *once and for all*. 
Our approach (2)

- Next, let’s use a pure function \texttt{swap} instead of \texttt{rename}.

\begin{verbatim}
data Abs a t = Abs a t
data Exp = ... | Lam (Abs Name Exp)

swap :: Name -> Name -> Exp -> Exp
subst_abs subst a t (Abs b e)
    = do b' <- fresh
         e' <- subst a t
             (swap b b' e)
         return (Abs b' e')

subst a t (Lam e) = do e' <- subst_abs subst a t e
                      return (Lam e')
\end{verbatim}

- We’ll see why this is important later.

- (Basically, it’s because \texttt{swap} is pure, easy to define and “naturally” capture avoiding.)
Our approach (3)

Next, note that we can parameterize the substitution functions by a monad $m$ that provides a fresh name generator:

```haskell
class Monad m => FreshM m where
  fresh :: m Name

subst_abs :: FreshM m =>
  Name -> Exp -> Abs Name Exp -> m (Abs Name Exp)

subst :: FreshM m => Name -> Exp -> Exp -> m Exp
```
Our approach (4)

- Finally, observe that we can make both substitution functions instances of a type class:

```haskell
class Subst t u where
    subst :: FreshM m => Name -> t -> u -> m u

instance Subst Exp (Abs Name Exp) where
    subst a t (Abs b e) = do b' <- fresh
                            e' <- subst a t
                             (swap b b' e)
                            return (Abs b' e')

instance Subst Exp Exp
    subst a t (Lam e) = do e' <- subst a t e
                            return (Lam e')
```

...
Story so far

- So far, I’ve suggested how nameplate can be reorganized, but not yet scrapped.
- E.g., using a type class for \texttt{Subst} and a monad for name-generation.
- Next step: provide a library with appropriate type classes and instances for common situations
- Key issue: defining renaming at all types.
- We use a FreshML-like approach based on swapping as the primitive renaming operation.
- I’ll describe FreshLib: a library that provides much of the functionality of FreshML as a Haskell class library.
Getting started

- To use FreshLib, you just write data declarations, empty Nom instances, and HasVar declarations.

```haskell
data Lam = Var Name
        | App Lam Lam
        | Lam (Abs Name Lam)
        | Const Int
        | ...

instance Nom Lam where
    -- empty
instance HasVar Lam where
    is_var (Var x) = Just x
    is_var _ = Nothing
```

- swap, subst are derived automatically.
Nominal types

- Type class `Nom`

  ```haskell
  class Nom a where
    swap :: Name -> Name -> a -> a
    fresh :: Name -> a -> Bool
    aeq :: a -> a -> Bool
  ```

- `swap a b x`: exchanges (all occurrences of) two names `a`, `b` in `x`
- `fresh a x`: tests whether `a` is “fresh for” (not free in) `x`
- `aeq x y`: tests alpha-equivalence of `x` and `y`

Note: Already have (essentially) this in Isabelle/HOL+Nominal.
Class Subst

- For ordinary types, substitutions ignore structure.
  
  \[
  \text{instance } (\text{Subst } t \ a, \, \text{Subst } t \ b) \Rightarrow \\
  \, \text{Subst } t \ (a,b) \\
  \text{where} \\
  \text{subst } a \ t \ (x,y) = \begin{aligned} \\
  &\text{do } x' \leftarrow \text{subst } a \ t \ x \\
  &y' \leftarrow \text{subst } a \ t \ y \\
  \end{aligned} \\
  \text{return } (x',y')
  \]

- For abstractions, substitutions rename bound names, then proceed.
  
  \[
  \text{instance Subst } t \ a \Rightarrow \text{Subst } t \ (\text{Abs Name } a) \text{ where} \\
  \text{subst } a \ t \ (\text{Abs } b \ x) = \begin{aligned} \\
  &\text{do } b' \leftarrow \text{fresh} \\
  &x' \leftarrow \text{subst } a \ t \ (\text{swap } b \ b' \ x) \\
  \end{aligned} \\
  \text{return } (\text{Abs } b' \ x')
  \]
Class FreeVars

- For ordinary types, \( fvs \) is union of \( fvs \) of components.
  
  \[
  \text{instance} \ (\text{FreeVars} \ t \ a, \text{FreeVars} \ t \ b) \Rightarrow \\
  \text{FreeVars} \ t \ (a,b) \\
  \text{where} \\
  \text{FreeVars} \ t \ (x,y) = \text{union} \ (fvs \ t \ x) \ (fvs \ t \ y)
  \]

- For abstractions, remove bound name from set
  
  \[
  \text{instance} \ \text{FreeVars} \ t \ a \Rightarrow \\
  \text{FreeVars} \ t \ (\text{Abs} \ Name \ a) \\
  \text{where} \\
  fvs \ t \ (\text{Abs} \ b \ x) = fvs \ t \ x \ \setminus \ [b]
  \]
Generic substitution

- In FreshLib, instances of Subst and FreeVars are auto-derived given instantiation of HasVar class.
- That is, once you know how to tell whether an Exp is a variable, all of the other cases of substitution are filled in “for free”.
- However, making this work relies on fairly involved generic programming techniques (cutting edge 1.5 years ago; hopefully better understood now)
- But probably not available for Isabelle/HOL anytime soon.
Problems with Haskell version

- The Haskell version of FreshLib has several limitations I haven’t figured out how to lift.
- Hard to **generalize to multiple name-types** (efficiently)
- Hard to **mask name-generation “effects”**—which don’t “really” affect the result, up to permutation of generated names, but Haskell doesn’t know this
- Not clear that swapping is a good way to **implement** name-binding/substitution—even in a lazy setting. Can we implement FreshLib efficiently using de Bruijn for bound names?
Perhaps the same idea can be adapted for nominal datatypes.

Given: Given a datatype `exp` with a "designated" variable

```
case var_exp : name -> exp
```

Construct: A "substitution function"

```
subst_exp : Nom a => name -> exp -> a -> a
```

for substituting for `exp` inside an arbitrary other nominal datatype.
Another question

- In addition to auto-deriving substitution, “standard lemmas” could be provided:

\[ x \not\in M \Rightarrow M[N/x] = M \]

\[ x \not\in N' \Rightarrow M[N/x][N'/y] = M[N'/y][N[N'/y]/x] \]

- Can we generate other useful traversals?

- e.g. nonstandard substitution operations like continuation substitution in \( \lambda \mu \)-calculus, “hereditary substitution” in new presentations of LF...

- No idea if this is possible/interesting/worth the trouble...
Related work

- [Pottier ML Workshop 2005]: Cαml, a source-to-source frontend that generates OCaml datatypes & “visitor” traversals from types decorated with binding structure
- [Sewell et al.] OTT tool: aimed at typesetting inference rules, binding, etc.
- [Mathijssen & Gabbay 2006]: capture-avoiding substitution via “nominal algebra”; essentially same idea as SYN
Mechanized Metatheory Model-Checking
Type systems are a powerful techniques for verifying properties of programs (e.g. memory safety)
- Provides guarantee that all programs in language have property
- To stay decidable, some “safe” programs must be disallowed

Much research in PL of the form “design a type system/program analysis to enforce $P$” (recently, $P$ often a security property)

Problem: How to specify and verify type systems?
Example

- $\lambda \rightarrow \times$ typing

\[
\begin{align*}
\Gamma \vdash () : \text{unit} & \quad \text{if } \forall x : \tau \in \Gamma \quad \text{then } \Gamma \vdash x : \tau \\
\Gamma \vdash e_1 : \tau \rightarrow \tau' & \quad \Gamma \vdash e_2 : \tau' \quad \Gamma \vdash e : \tau \\
\Gamma \vdash e_1 \, e_2 : \tau & \\
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 & \quad \Gamma \vdash e : \tau_1 \times \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 & \\
\Gamma \vdash \pi_1(e) : \tau_1 & \\
\Gamma \vdash \pi_2(e) : \tau_1 \\
(\lambda x. e) \, e' & \rightarrow e[e'/x] \\
\pi_i(e_1, e_2) & \rightarrow e_i
\end{align*}
\]

- Claim: This version is full of bugs.
Metatheory verification

- Current state of practice:
  1. write down typing rules, operational semantics
  2. try to prove syntactic properties, culminating in soundness
  3. if proof fails, goto 1.

- Step 2 tedious & sensitive to changes, so tempting to “handwave”
  - Especially hours before paper deadline (I’m certainly guilty of this)

- But this is dangerous (ML ∀ + ref bug, Java array subtyping bug, Cyclone ∃ + ref bug)
Mechanized metatheory verification

- Computers should be doing most of the work of verification.
- Recent interest in making metatheory verification tools “ready for prime-time” (POPLMark Challenge)
- Long-term research program on metatheory verification at CMU using higher order abstract syntax & LF
  - “Realistic” core languages can be formalized (e.g. POPL 2007 formalization of core ML)
  - Probably the most practical approach
  - But still a lot of work to learn
- Several other syntax encodings (de Bruijn, nominal) and theorem provers also considered (Coq, HOL, Isabelle/HOL)
Find the bug

- \( \lambda \rightarrow \times \) typing

\[
\begin{align*}
\Gamma \vdash () : \text{unit} & \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\Gamma \vdash e_1 : \tau \rightarrow \tau' & \quad \Gamma \vdash e_2 : \tau' \\
\Gamma \vdash e_1 \; e_2 : \tau & \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'} \\
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 & \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \\
\Gamma \vdash \pi_2(e) : \tau_1
\end{align*}
\]
Find the bugs

- $\lambda \rightarrow \times$ typing

\[
\begin{align*}
\Gamma \vdash () : \text{unit} & \quad x : \tau \in \Gamma \\
\Gamma \vdash x : \tau & \\
\Gamma \vdash e_1 : \tau \rightarrow \tau' & \quad \Gamma \vdash e_2 : \tau' & \quad \Gamma \vdash e : \tau \\
\Gamma \vdash e_1 \ e_2 : \tau & \quad\Gamma \vdash \lambda x. e : \tau \rightarrow \tau' \\
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 & \quad \Gamma \vdash e : \tau \times \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 & \quad \Gamma \vdash \pi_1(e) : \tau_1 & \quad \Gamma \vdash \pi_2(e) : \tau_1
\end{align*}
\]

- Claim: Trying to verify correctness is not the fastest way to find such bugs.
Find the bugs, reloaded

- $\lambda \rightarrow \times$ typing

\[
\begin{align*}
\Gamma \vdash () : \text{unit} & \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\Gamma \vdash e_1 : \tau \rightarrow \tau' & \quad \Gamma \vdash e_2 : \tau' \quad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'} \\
\Gamma \vdash e_1 \ e_2 : \tau & \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \\
\Gamma \vdash e : \tau_1 \times \tau_2 & \quad \frac{\Gamma \vdash \pi_1(e) : \tau_1}{\Gamma \vdash \pi_2(e) : \tau_1}
\end{align*}
\]

- Claim: Trying to verify correctness is not the fastest way to find such bugs.

- Also, it is dangerous to intentionally add errors to an example; it keeps you from looking for the unintentional ones.
Example

- Consider reduction step $\pi_2(1,()) \rightarrow ()$
- Then we have

\[
\begin{align*}
\Gamma & \vdash 1 : \text{int} & \Gamma & \vdash () : \text{unit} \\
\Gamma & \vdash (1,()) : \text{int} \times \text{unit} \\
\Gamma & \vdash \pi_2(1,()) : \text{int} \quad (\ast)
\end{align*}
\]

But no derivation of

\[
\begin{align*}
\Gamma & \vdash () : \text{int}
\end{align*}
\]

- If only we had a way of systematically searching for such counterexamples...
Experimental metatheory?!

- Any current verification approach introduces a “gap” between formally verified language and implemented version.
- Type systems are theories of programming language behavior.
- Testing theories against reality by attempting falsification and independent confirmation is a basic scientific principle.
- Though weaker than formal verification of “real” system, rigorous testing complements informal verification (or verification of abstract system).
Metatheory model-checking?

- Goal: Catch “shallow” bugs in type systems, operational semantics, etc.
- Model checking: attempt to verify finite system by searching exhaustively for counterexamples
  - Highly successful for validating hardware designs
  - More helpful in (common) case that system has bug
- Partial model checking: search for counterexamples over some finite subset of infinite search space
  - Produces a counterexample if one exists, but cannot verify system correct
Pros

- Finds shallow counterexamples quickly
- Separates concerns (researchers focus on efficiency, engineers focus on real work)
- Lifts user’s brain out of inner loop
- Easy to use; theorem prover expertise/Kool-Aid™ not required
- Easy to implement naive solution
- (Buzzword-compatible? Guilty as charged)
Cons

- Failure to find counterexample does not guarantee property holds
- Hard to tell what kinds of counterexamples might be missed
- “Nontrivial” bugs (e.g. $\forall /\text{ref}$, $\leq /\text{ref}$) currently beyond scope
Idea

- Represent object system in a suitable meta-system.
- Specify property it should have.
- System searches exhaustively for counterexamples.
- Meanwhile, you try to prove properties (or get coffee, sleep, whatever).
Realization

- Represent object system in a suitable meta-system.
  - I will use pure $\alpha$Prolog programs (but many other possibilities)
- Specify property it should have.
  - Universal Horn ($\Pi_1$) formulas can specify type preservation, progress, soundness, weakening, substitution lemmas, etc.
- System searches exhaustively for counterexamples.
  - Bounded DFS, negation as failure
- Meanwhile, you try to prove properties (or get coffee, sleep, whatever).
The “code” slide

- $\alpha$Prolog: a simple extension of Prolog with nominal abstract syntax.

\[
\begin{align*}
\text{var} : \text{name} & \rightarrow \text{exp}. & \text{app} : (\text{exp}, \text{exp}) & \rightarrow \text{exp}. & \text{lam} : \langle \text{name}\rangle\text{exp} & \rightarrow \text{exp}.
\end{align*}
\]

\[
\begin{align*}
tc(G, \text{var}(X), T) & : - \text{List.mem}((X, T), G). \\
tc(G, \text{app}(M, N), U) & : - \exists T.\text{tc}(G, M, \text{arr}(T, U)), \text{tc}(G, N, T). \\
tc(G, \text{lam}(\langle x \rangle M), \text{arr}(T, U)) & : - x \# G, \text{tc}([(x, T)|G], M, U).
\end{align*}
\]

\[
\begin{align*}
\text{sub}(&\text{var}(X), X, N) = N. \\
\text{sub}(&\text{var}(X), Y, N) = \text{var}(X) : - X \# Y. \\
\text{sub}(&\text{app}(M_1, M_2), Y, N) = \text{app}(\text{sub}(M_1, Y, N), \text{sub}(M_2, Y, N)). \\
\text{sub}(&\text{lam}(\langle x \rangle M), Y, N) = \text{lam}(\langle x \rangle \text{sub}(M, Y, N)) : - x \# (Y, N).
\end{align*}
\]

- Equality coincides with $\equiv_\alpha$, $\#$ means “not free in”, $\langle x \rangle M$ is an $M$ with $x$ bound.
Problem definition

- Define model $\mathcal{M}$ using a (pure) logic program $P$.
- Consider specifications of the form

$$\forall \vec{X}. B_1 \land \cdots \land B_n \supset A$$

(note: disjunctive, existential $A, B_i$ possible by adding clauses)
- A *counterexample* is a ground substitution $\theta$ such that

$$\mathcal{M} \models \theta(G_1) \land \cdots \land \mathcal{M} \models \theta(G_n) \land \mathcal{M} \not\models \theta(A)$$

- The *partial model checking problem*: Does a counterexample exist? If so, construct one.
- Obviously r.e., undecidable
Implementation

- Naive idea: generate substitutions and test; iterative deepening.
- Write “generator” predicates for all base types.
- For all combinations, see if hypotheses succeed while conclusion fails.

\[ \text{?- } \text{gen}(X_1) \land \cdots \land \text{gen}(X_n) \land G_1 \land \cdots \land G_n \land \text{not}(A) \]

- Problem: High branching factor
  - even if we abstract away infinite base types
- Can only check up to max depth 1-3 before boredom sets in.
Implementation (II)

- Fact: Searching for instantiations of variables first is wasteful.
- Want to delay this expensive step as long as possible.
- Less naive idea: generate derivations and test.
- Search for complete proof trees of all hypotheses
- Instantiate all remaining variables
- Then, see if conclusion fails.

\[
?\leftarrow G_1 \land \cdots \land G_n \land \text{gen}(X_1) \land \cdots \land \text{gen}(X_n) \land \text{not}(A)
\]

- Raises boredom horizon to depths 5-10 or so.
Demo

- Debugging simply-typed lambda calculus spec.
Experience

- Implemented within $\alpha$Prolog; more or less a hack...
- Checked $\lambda \to \times$ example, up to type soundness
- Checked some syntactic properties of an LF typechecking algorithm
- Since then, have implemented and checked Ch. 8, 9, some of Ch. 11 of TAPL too
- NB: Published, high-quality type systems are probably not the most interesting test cases...
Experience (II)

- Writing $\Pi_1$ specifications is **dirt simple**
  - They make great **regression tests**
  - I now write them as a matter of course

- Order of goals makes a big difference to efficiency; optimization principles not clear yet.

- Not enough to just check “main” theorems
  - System could be “trivially” sound
  - Checking intermediate lemmas helps catch bugs earlier

- Bounded DFS also useful for exploration, “yes, $\neg \phi$ can happen”
Is this trivial?

- Tried a few “realistic” examples recently
- Naive Mini-ML with references: boredom horizon 9; smallest counterexample I can think of needs depth 18.
  - Back of envelope estimate: would need somewhere between 191 and 4.4 million years to find
  - I guess I need a faster laptop.
  - Bright side: blind search massively parallelizable...
- At this point, won’t catch any “real” bugs in finished products.
- But perhaps useful during development of type system
Conclusions

- Simplistic model checking/counterexample search techniques are useful for catching shallow bugs
- Improvement needed to increase coverage
- Many refinements possible
- Checker implemented in $\alpha$Prolog; will be in next release