Strongly-typed System F in GHC

- Stephanie Weirich
- WG 2.8 Meeting
- Bordeaux, France
- May 2019
Motivation
Symbolic execution of imperative programs

Crucible

SAW-script

JVM

LLVM

MIR

CVC4

Z3

...
System F types

data Ty =
  BaseTy
  \mid VarTy Nat \quad -- \text{de Bruijn index}
  \mid FnTy Ty Ty \quad -- \text{single argument function types}
  \mid PolyTy Nat Ty \quad -- \text{multiple binding polymorphic types}

data Exp :: [Ty] -> Ty -> Type where
  BaseE :: Exp g BaseTy
  AppE :: Exp g (FnTy t1 t2) \quad -- \text{function}
     -> Exp g t1 \quad -- \text{argument}
     -> Exp g t2
  ....
Strongly-typed System F in Haskell

A Type-Preserving Compiler in Haskell

Louis-Julien Guillemette  Stefan Monnier
Université de Montréal
{guillej, monnier}@iro.umontreal.ca

Abstract
There has been a lot of interest of late for programming languages that incorporate features from dependent type systems and proof assistants, in order to capture important invariants of the program in the types. This allows type-based program verification and is a promising compromise between plain old types and full blown Hoare logic proofs. The introduction of GADTs in GHC (and more recently type families) made such dependent typing available in an industry-quality implementation, making it possible to consider its use in large scale programs.

We have undertaken the construction of a complete compiler for System F, whose main property is that the GHC type checker verifies mechanically that each phase of the compiler properly preserves types. Our particular focus is on “types rather than proofs”: reasonably few annotations that do not overwhelm the actual code, similar guaranties for a functional language. Both are developed as Coq proofs from which a working compiler is obtained by means of program extraction.

With the introduction of generalized algebraic data types (GADTs) in the Glasgow Haskell Compiler (GHC), and more recently type families (Schrijvers et al. 2007), a useful (if limited) form of dependent typing is finally available in an industry-quality implementation of a general-purpose programming language. Thus arises the possibility of establishing compiler correctness through type annotations in Haskell code, without the need to encode elaborate proofs as separate artifacts. In this work, we use types to enforce type preservation: our typed intermediate representation lets GHC’s type checker manipulate and check our object types.

Other than the CPS conversion over System F of Chipala (2008) developed in parallel and presented elsewhere in these
Strongly-typed System F in Coq

Strongly Typed Term Representations in Coq

Nick Benton · Chung-Kil Hur · Andrew Kennedy · Conor McBride
Autosubst: Reasoning with de Bruijn Terms and Parallel Substitution

Steven Schäfer        Tobias Tebbi        Gert Smolka
Saarland University
June 10, 2015

To appear in Proc. of ITP 2015, Nanjing, China, Springer LNAI

Reasoning about syntax with binders plays an essential role in the formalization of the metatheory of programming languages. While the intricacies of binders can be ignored in paper proofs, formalizations involving binders tend to be heavyweight. We present a discipline for syntax with binders based on de Bruijn terms and parallel substitutions, with a decision procedure covering all assumption-free equational substitution lemmas. The approach is implemented in the Coq library AUTOSUBST, which additionally derives substitution operations and proofs of substitution lemmas for custom term types. We demonstrate the effectiveness of the approach with several case studies, including part A of the POPLmark challenge.
Singletons library  (Eisenberg, Scott, ...)

\[
\begin{align*}
\text{data Ty =} \\
\text{  BaseTy} \\
\text{  | VarTy Nat} & \quad \text{-- de Bruijn index} \\
\text{  | FnTy Ty Ty} & \quad \text{-- single argument function types} \\
\text{  | PolyTy Nat Ty} & \quad \text{-- multiple binding polymorphic types} \\
\text{  | ]]}
\end{align*}
\]

-- "forall a. a -> a"

idTy = PolyTy (S Z) (FnTy (Var Z) (Var Z))

sidTy :: Sing (PolyTy (S Z) (FnTy (Var Z) (Var Z))

sidTy = SPolyTy (SS SZ) (SFnTy (SVar SZ) (SVar SZ))
Strongly-typed Expressions

data Exp :: [Ty] -> Ty -> Type where

  BaseE :: Exp g BaseTy

  VarE :: (Idx g n ~ Just t)
          => Sing n                 -- index
          -> Exp g t

  LamE :: Sing t1              -- type of binder
          -> Exp (t1:g) t2         -- body of lambda
          -> Exp g (FnTy t1 t2)

  AppE :: Exp g (FnTy t1 t2)   -- function
          -> Exp g t1            -- argument
          -> Exp g t2

$(singletons [d]
  idx :: [Ty] -> Nat -> Maybe Ty
  idx ( ty : _ ) Z   = Just ty
  idx ( _ : tys) (S n) = idx tys n
  idx [] _   = Nothing
])

Idx :: [Ty] -> Nat -> Maybe Ty
Strongly-typed Expressions w/ Polymorphism

data Exp :: [Ty] -> Ty -> Type where

  ...

  TyLam :: Sing n          -- num of tyvars to bind
           -> Exp ???? t    -- need to shift the context
           -> Exp g (PolyTy n t)

  TyApp :: (k ~ Length ts) -- length requirement
           => Exp g (PolyTy k t) -- polymorphic term
           -> Sing ts          -- type arguments
           -> Exp g ????      -- need to substitute \{ ts / 0 .. k-1 \} t
de Bruijn indices with Parallel Substitutions

• Substitution $\sigma$  
  type Sub = Nat $\to$ Ty

• Operation  
  subst :: Sub $\to$ Ty $\to$ Ty

• Substitution algebra
  
  - identity  
    id x = VarTy x
  
  - composition  
    $(\sigma_1 \circ \sigma_2) x = \text{subst } \sigma_2 (\sigma_1 x)$
  
  - increment  
    inc x = VarTy (x+1)

• View as infinite list of types $(t_0, t_1, t_2, \ldots)$
  
  - cons  
    $t \cdot (t_0, t_1, \ldots) = (t, t_0, t_1, \ldots)$

• Simultaneous substitution
  
  \{ t_0, t_1, \ldots t_k/0, 1, \ldots k \}  
  fromList [t_0, t_1, \ldots, t_k] = t_0 \cdot t_1 \cdot \ldots \cdot t_k \cdot \text{id}
data Sub =
    Inc Nat -- increment by n, n == 0 is id
  | Ty :: Sub -- cons
  | Sub :∘ Sub -- compose substitutions

applyS :: Sub -> Nat -> Ty
applyS (Inc n) x = VarTy (n + x)
applyS (ty :: s) x = case x of
    Z     -> ty
    (S m) -> applyS s m
applyS (s1 :∘ s2) x = subst s2 (applyS s1 x)
de Bruijn indices with Parallel Substitutions

\[ \text{subst} :: \text{Sub} \to \text{Ty} \to \text{Ty} \]
\[ \text{subst} \circ \sigma \circ \text{BaseTy} = \text{BaseTy} \]
\[ \text{subst} \circ \sigma \circ (\text{VarTy} \ x) = \text{applyS} \circ \sigma \circ x \]
\[ \text{subst} \circ \sigma \circ (\text{FnTy} \ a \ r) = \text{FnTy} \ (\text{subst} \circ \sigma \circ a) \ (\text{subst} \circ \sigma \circ r) \]
\[ \text{subst} \circ \sigma \circ (\text{PolyTy} \ 1 \ a) = \text{PolyTy} \ 1 \ (\text{subst} \circ \text{lift1} \circ \sigma \circ a) \]

\[ \text{lift1} :: \text{Sub} \to \text{Sub} \]
\[ \text{lift1} \circ \sigma = (\text{VarTy} \ 0) \circ (\sigma \circ \text{inc}) \quad \text{-- leave variable 0 alone, shift domain of } \sigma \]
\[ \quad \text{-- increment all free vars in range of } \sigma \text{ by 1} \]
de Bruijn indices with Parallel Substitutions

\[ \text{subst} :: \text{Sub} \to \text{Ty} \to \text{Ty} \]

\[ \text{subst} \; \sigma \; \text{BaseTy} \quad = \quad \text{BaseTy} \]

\[ \text{subst} \; \sigma \; (\text{VarTy} \; x) \quad = \quad \text{supply} \; \sigma \; x \]

\[ \text{subst} \; \sigma \; (\text{FnTy} \; a \; r) \quad = \quad \text{FnTy} \; (\text{subst} \; \sigma \; a) \; (\text{subst} \; \sigma \; r) \]

\[ \text{subst} \; \sigma \; (\text{PolyTy} \; n \; a) \quad = \quad \text{PolyTy} \; n \; (\text{subst} \; \text{(lift} \; n \; \sigma) \; a) \]

\[ \text{lift} :: \text{Nat} \to \text{Sub} \to \text{Sub} \]

\[ \text{lift} \; n \; \sigma = \ldots \quad -- \text{leave variables 0 .. n-1 alone, shift domain of } \sigma \]

\[ -- \text{increment all free vars in range of } \sigma \text{ by } n \]
Strongly-typed Expressions w/ Polymorphism

data Exp :: [Ty] -> Ty -> Type where

  ...  
  TyLam :: Sing n         -- num of tyvars to bind
          -> Exp (IncList n g) t  -- body of type abstraction
          -> Exp g (PolyTy n t)

  TyApp :: (k ~ Length ts) -- length requirement
          => Exp g (PolyTy k t) -- polymorphic term
          -> Sing ts          -- type arguments
          -> Exp g (Subst (FromList ts) t)

  IncList n g == Map (Subst (Inc n)) g
Type substitution in well-typed terms

\[
\text{substTy} :: \forall s \ g \ ty.
\]

\[
\begin{align*}
\text{Sing } s & \rightarrow \text{Exp } g \ ty \\
\text{Sing } s & \rightarrow \text{Exp} (\text{SubstList } s \ g) (\text{Subst } s \ ty)
\end{align*}
\]

\[
\begin{align*}
\text{substTy } s \ (\text{VarE } n) & = \text{VarE } n \\
\text{substTy } s \ \text{BaseE} & = \text{BaseE} \\
\text{substTy } s \ (\text{LamE } ty \ e) & = \text{LamE} (\text{sSubst } s \ ty) (\text{substTy } s \ e) \\
\text{substTy } s \ (\text{AppE } e1 \ e2) & = \text{AppE} (\text{substTy } s \ e1) (\text{substTy } s \ e2)
\end{align*}
\]
sysf.lhs:898:6: error:
  • Could not deduce: Idx (Map (SubstSym1 s) g) n
       ~ 'Just (Subst s ty)
   arising from a use of ‘VarE’
   from the context: Idx g n ~ 'Just ty
   bound by a pattern with constructor:
     VarE :: forall (g :: [Ty]) (n :: Nat) (t :: Ty).
     (Idx g n ~ 'Just t) =>
     Sing n -> Exp g t,
   in an equation for ‘substTy’
   at sysf.lhs:896:12-17
  • In the expression: VarE n
   In an equation for ‘substTy’: substTy s (VarE n) = VarE n
  • Relevant bindings include
    n :: Sing n (bound at sysf.lhs:896:17)
    s :: Sing s (bound at sysf.lhs:896:9)
    substTy :: Sing s -> Exp g ty -> Exp (SubstList s g) (Subst s ty)
              (bound at sysf.lhs:896:1)

          898| =  VarE n
             ^^^^^
Failed, one module loaded.
Type substitution in terms

\( \text{substTy} :: \forall s \ g \ t y. \)

- \( \text{Sing} \ s \)
- \( \to \ \text{Exp} \ g \ t y \)
- \( \to \ \text{Exp} \ (\text{SubstList} \ s \ g) \ (\text{Subst} \ s \ t y) \)

\( \text{substTy} \ s \ (\text{VarE} \ n) \)
- \( \mid \ \text{Refl} \leftarrow \text{axiom}_\text{SubstIdx} \ (\text{undefined} :: \text{Sing} \ g) \ n \ s \)
  \= \text{VarE} \ n \)

\( \text{substTy} \ s \ \text{BaseE} \)
- \( \= \text{BaseE} \)

\( \text{substTy} \ s \ (\text{LamE} \ t y \ e) \)
- \( \= \text{LamE} \ (\text{sSubst} \ s \ t y) \ (\text{substTy} \ s \ e) \)

\( \text{substTy} \ s \ (\text{AppE} \ e1 \ e2) \)
- \( \= \text{AppE} \ (\text{substTy} \ s \ e1) \ (\text{substTy} \ s \ e2) \)

\( \text{Idx} \ g \ n \sim \text{Just} \ t \implies \text{Idx} \ (\text{SubstList} \ s \ g) \ n \sim \text{Just} \ (\text{Subst} \ s \ t) \)
Why should we believe this axiom?

• Vigorous assertion
  \[
  \text{axiom\_SubstIdx} :: (\text{Idx } g \text{ n } \sim \text{ Just } t) \Rightarrow \\
  \quad \text{Sing } g \rightarrow \text{Sing } n \rightarrow \text{Sing } s \rightarrow \text{Idx } (\text{SubstList } s \text{ g}) \text{ n } :\sim: \text{Just } (\text{Subst } s \text{ t}) \\
  \text{axiom\_SubstIdx } g \_n \_s = \text{unsafeCoerce } \text{Refl}
  \]

• "Provable" in Haskell
  \[
  \text{lemma\_SubstIdx} :: (\text{Idx } g \text{ n } \sim \text{ Just } t) \Rightarrow \\
  \quad \text{Sing } g \rightarrow \text{Sing } n \rightarrow \text{Sing } s \rightarrow \text{Idx } (\text{SubstList } s \text{ g}) \text{ n } :\sim: \text{Just } (\text{Subst } s \text{ t}) \\
  \text{lemma\_SubstIdx } (\text{SCons } \_\_ ) \text{ \_SZ \_s } = \text{Refl} \\
  \text{lemma\_SubstIdx } (\text{SCons } \_\_s) \text{ (SS n) s | Refl } \underline{\text{<- lemma\_SubstIdx } xs \_n \_s = \text{Refl}}
  \]

• It's an easy lemma, even in Haskell
• BUT, runtime cost and need to have "Sing g" available
Why should we believe this axiom?

• Vigorous assertion
  axiom_SubstIdx :: (Idx g n ~ Just t) =>
    Sing g -> Sing n -> Sing s -> Idx (SubstList s g) n :~: Just (Subst s t)
axiom_SubstIdx _g _n _s = unsafeCoerce Refl

• "Provable" in Coq
  Lemma fIdx : forall {a}{b} (f:a -> b) n y ts,
    idx n ts = Some y ->
    idx n (map f ts) = Some (f y).
Proof.
  induction n; destruct ts; simpl; try done.
  - intros h; inversion h; done.
  - intros h; eauto.
Qed.

• It's an easy lemma
• Coq and Haskell are close but aren't the same
Why should we believe this axiom?

• Vigorous assertion
  \[\text{axiom}_{\text{SubstIdx}} :: (\text{Idx} \ g \ n \sim \text{Just} \ t) \implies\]
  \[\text{Sing} \ g \implies \text{Sing} \ n \implies \text{Sing} \ s \implies \text{Idx} \ (\text{SubstList} \ s \ g) \ n \sim \text{Just} \ (\text{Subst} \ s \ t)\]
  \[\text{axiom}_{\text{SubstIdx}} \ g \ n \ s = \text{unsafeCoerce} \ \text{Refl}\]

• "Testable" in Haskell
  \[\text{prop}_{\text{SubstIdx}} :: [\text{Ty}] \implies \text{Nat} \implies \text{Sub} \implies \text{Bool}\]
  \[\text{prop}_{\text{SubstIdx}} \ g \ n \ s =\]
  \[\text{idx} \ (\text{substList} \ s \ g) \ n \equiv (\text{subst} \ s \ <\!\!\!\!\!\!\!\!\!\!\!<) \ \text{idx} \ g \ n\]

• Singletons means we already have non-refined implementation
• Small modification necessary to make testing effective
  \[(\text{idx} \ g \ n \equiv \text{Just} \ t) \implies (\text{idx} \ (\text{substList} \ s \ g) \ n \equiv \text{Just} \ (\text{subst} \ s \ t))\]
Other axioms needed for substTy

axiom_LiftIncList :: forall s k g.
  LiftList k s (IncList k g) :~: IncList k (SubstList s g)

axiom_SubstFromList :: forall t s tys.
  Subst s (Subst (FromList tys) t)
  :~: Subst (FromList (SubstList s tys))
  (Subst (Lift (Length tys) s) t)

axiom_LengthSubstList :: forall s tys.
  Length (SubstList s tys) :~: Length tys

Easy to test w/ QuickCheck
Much, much more difficult to prove in Haskell and Coq
What can you do with this in GHC?

- **System F**
  - Type checker
  - Type-safe evaluation
  - Parallel reduction
  - CPS conversion (cf. Pottier, "Revisiting the CPS Transformation and its Implementation")

- **Crucible**
  - wip-poly branch
  - mir-verifier project
Conclusions

• I'm ok with unsafeCoerce, backed by QuickCheck
  • Benefits of strong typing, yet assumptions clearly marked in code
  • Hard to test typed ASTs, easy to test (type) substitution
  • Mistakes are fatal though
  • What else can we do, really?

• Code available:

https://github.com/sweirich/challenge/debruijn/sysf.lhs