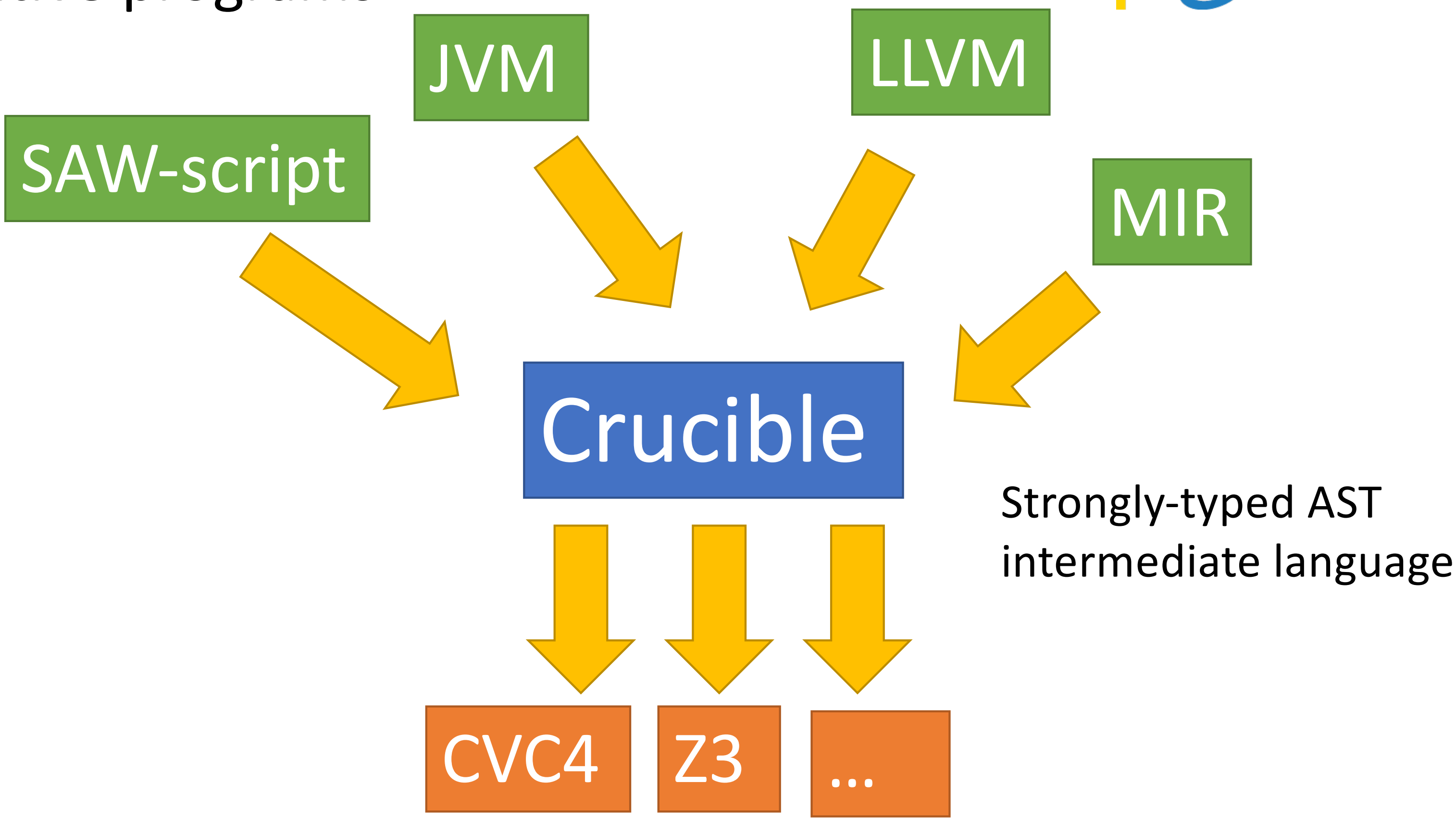
Strongly-typed System F in GHC

- Stephanie Weirich
- WG 2.8 Meeting
- Bordeaux, France
- May 2019

Motivation

Symbolic execution of imperative programs



System F types

```
data Ty =
      BaseTy
    | VarTy Nat          -- de Bruijn index
    | FnTy Ty Ty         -- single argument function types
    | PolyTy Nat Ty      -- multiple binding polymorphic types

data Exp :: [Ty] -> Ty -> Type where
   BaseE  :: Exp g BaseTy
   AppE   :: Exp g (FnTy t1 t2)      -- function
          -> Exp g t1                -- argument
          -> Exp g t2
   ....
```

Strongly-typed System F in Haskell

A Type-Preserving Compiler in Haskell

Louis-Julien Guillemette Stefan Monnier
Université de Montréal
{guillelj,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 Chlipala (2008) developed in parallel and presented elsewhere in these

ICFP 2008

Strongly-typed System F in Coq

J Autom Reasoning manuscript No.
(will be inserted by the editor)

Strongly Typed Term Representations in Coq

**Nick Benton · Chung-Kil Hur ·
Andrew Kennedy · Conor McBride**

JAR August 2012, Volume 49, Issue 2, pp 141-159

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, ...)

```
$(singletons [d|
   data Ty =
       BaseTy
     |  VarTy Nat           -- de Bruijn index
     |  FnTy Ty Ty          -- single argument function types
     |  PolyTy Nat Ty      -- multiple binding polymorphic types
   |])
-- "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 σ type Sub = Nat -> Ty
- *Operation*   subst :: Sub -> Ty -> Ty
- Substitution algebra

identity	**id** x	= VarTy x
composition	$(\sigma 1 \circ \sigma 2)$ x	= subst $\sigma 2$ ($\sigma 1$ x)
increment	**inc** x	= VarTy (x+1)

- View as infinite list of types (t0, t1, t2, ...)

cons   t • (t0, t1, ...) = (t, t0, t1, ...)

- Simultaneous substitution

{ t0,t1,...tk/0,1,...k }   fromList [t0, t1, ..., tk] = t0 · t1 · ... · tk · id

Defunctionalize for GHC

```
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

subst :: Sub -> Ty -> Ty
subst σ BaseTy = BaseTy
subst σ (VarTy x) = applyS σ x
subst σ (FnTy a r) = FnTy (subst σ a) (subst σ r)
subst σ (PolyTy 1 a) = PolyTy 1 (subst (lift1 σ) a)

If you are playing along in a proof assistant, this is NOT structurally recursive. The definition of subst refers to lift, which refers to compose, which refers to subst.
Solution: define renaming first

lift1 :: Sub -> Sub
lift1 σ = (VarTy 0) :· (σ :∘ inc) -- leave variable 0 alone, shift domain of σ
-- increment all free vars in range of σ by 1

de Bruijn indices with Parallel Substitutions

```
subst :: Sub -> Ty -> Ty
subst σ BaseTy        = BaseTy
subst σ (VarTy x)     = spplyS σ x
subst σ (FnTy a r)    = FnTy (subst σ a) (subst σ r)
subst σ (PolyTy n a)  = PolyTy n (subst (lift n σ) a)

lift :: Nat -> Sub -> Sub
lift n σ =  ...        -- leave variables 0 .. n-1   alone, shift domain of σ
                       -- increment all free vars in range of σ 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

```
substTy :: forall s g ty.
     Sing s
  -> Exp g ty
  -> Exp (SubstList s g) (Subst s ty)
substTy s (VarE n)         = VarE n
substTy s BaseE            = BaseE
substTy s (LamE ty e)      = LamE (sSubst s ty) (substTy s e)
substTy s (AppE e1 e2)  = AppE (substTy s e1) (substTy s  e2)
```

SubstList s g == Map (Subst s) g

sSubst :: Sing s -> Sing t -> Sing (Subst s t) debruijn — ghc -B/Users/sweirich/local/stow/ghc-8.6.4/lib/ghc-8.6.4 --interactive sysf.lhs

```
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.
*Nat>
```

Type substitution in terms

```
substTy :: forall s g ty.
     Sing s
  -> Exp g ty
  -> Exp (SubstList s g) (Subst s ty)
substTy s (VarE n)
      | Refl <- axiom_SubstIdx (undefined :: Sing g) n s
      = VarE n
substTy s BaseE          = BaseE
substTy s (LamE ty e)    = LamE (sSubst s ty) (substTy s e)
substTy s (AppE e1 e2)   = AppE (substTy s e1) (substTy s  e2)
```

Idx g n ~ Just t implies
Idx (SubstList s g) n ~ Just (Subst s t)

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 Haskell

```
lemma_SubstIdx :: (Idx g n ~ Just t) =>
    Sing g -> Sing n -> Sing s -> Idx (SubstList s g) n :~: Just (Subst s t)
lemma_SubstIdx (SCons _ _)  SZ     s   = Refl
lemma_SubstIdx (SCons _ xs)  (SS n) s | Refl <- lemma_SubstIdx xs n s = 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

```
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
```

- "Testable" in Haskell

```
prop_SubstIdx :: [Ty] -> Nat -> Sub -> Bool
prop_SubstIdx g n s =
   idx (substList s g) n == (subst s <$> idx g n)
```

- Singletons means we already have non-refined implementation
- Small modification necessary to make testing effective

```
(idx g n == Just t) ==> (idx (substList s g) n == Just (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)
```

Easy to test w/ QuickCheck
Much, **much** more difficult to prove in Haskell and Coq

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

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