Generative type abstraction and type-level computation

(Wrestling with System FC)

Stephanie Weirich, Steve Zdancewic

University of Pennsylvania

Dimitrios Vytiniotis, Simon Peyton Jones

Microsoft Research, Cambridge

POPL 2011, Austin TX, January 2011

Type generativity is useful

- Module implementor:

```
module MImpl ( Tel, … )
…
newtype Tel = MkTel String
…
```

Inside MImpl:

```
      Tel ~ String
```

We can also lift this equality:

```
   List Tel ~ List String
Tel -> Int ~ String -> Int
```

etc.

- Module consumer:

```
module MCons
import MImpl
…
f :: Tel -> Tel
f x = “0030” ++ x
```

Inside MCons:

```
      Tel ≁ String
```

- Well-explored ideas found in various forms in modern languages [e.g. see papers on ML modules by Harper, Dreyer, Rossberg, Russo, …]

Type-level computation is useful

In the Glasgow Haskell Compiler, type-level computation involves type classes and families:

```
module MImpl (Tel)
…
class LowLevel a where
  type R a
  toLowLevel :: a -> R a

instance LowLevel String where
  type R String = ByteArray
  toLowLevel x = strToByteArray x

instance LowLevel Tel where
  type R Tel = Int64
  toLowLevel x = …


…
```

R is a "type function"

`R String ~ ByteArray`

`R Tel ~ Int64`

But there’s a problem!

```
module MImpl (Tel, …)

newtype Tel = MkTel String

class LowLevel a where
  type R a
  …

instance LowLevel String where
  type R String = ByteArray
  …

instance LowLevel Tel where
  type R Tel = Int64
  …


…
```

In the rest of the module:

`Tel ~ String`

Hence by lifting

`R Tel ~ R String`

Hence …

`ByteArray ~ Int64`

This paper

- Type generativity and type functions are both and simultaneously useful!
- But it's easy to lose soundness [e.g. see GHC bug trac #1496]
- So, what's some good solution that combines these features?

This talk. The rest is in the paper

System FC2

A novel, sound, strongly-typed language with type-level equalities

1. Stages the use of the available equalities, to ensure soundness
2. Distinguishes between "codes" and "types" as in formulations of Type Theory [e.g. see papers by Dybjer] and intensional type analysis [e.g. see papers by Weirich, Crary]
3. Improves GHC's core language [System FC, Sulzmann et al.]
4. Soundness proof w/o requiring strong normalization of types

Recap

```
newtype Tel = MkTel String         -- Tel ~ String

type instance R String = ByteArray -- R String ~ ByteArray
type instance R Tel = Int64        -- R Tel ~ Int64
```

`R String` MUST NOT BE EQUATED TO `R Tel`

`(List String)` OK TO BE EQUATED TO `(List Tel)`

A non-solution

- So lifting is(?) the source of all evil:

$$\frac{\Gamma \;\vdash\; \tau \sim \sigma}{\Gamma \;\vdash\; \mathrm{T}\,\tau \sim \mathrm{T}\,\sigma}$$

- Possible solution: disallow lifting if T is a type function
- Seems arbitrary, and restrictive, **and** does not quite work

```
data TR a = MkTR (R a)

to :: ByteArray -> TR String
to x = MkTR x

from :: TR Tel -> Int64
from (MkTR x) = x
```

```
TR Tel ~ TR String
```

JUST AS BAD, BECAUSE THEN:

```
from.to :: ByteArray -> Int64
```

Type Theory to the Rescue: Roles

- As is common in Type Theory, distinguish between a code (a “name”) and a type (a “set

```
newtype Tel
```

YOUR TAKEAWAY #1

- Newtype definitions introd
  - A code (such as Tel) can imp .g.
    (λx: Tel. `
- Importantly codes and types have **different notions of equality**: code-equality and type-equality

```
Γ ⊦ Tel ~ String : */TYPE
Γ ⊦ Tel ≁ String : */CODE
```

Code vs Type Equality

- If τ and σ are equal as codes then they are equal as types:

$$\frac{\Gamma \ \vdash \ \tau \sim \sigma : */\mathrm{CODE}}{\Gamma \ \vdash \tau \sim \sigma : */\mathrm{TYPE}}$$

- But two different codes may or may not be equal as types

```
newtype Tel = MkTel String
newtype Address = MkAddr String
```

```
Γ ⊦ Tel ~ Address : */TYPE
Γ ⊦ Tel ≁ Address : */CODE
```

Using the FC2 kind system to track roles

- Key idea:

  Type-level computations dispatch on codes, not types

- Use the kind system of FC2 to track codes

Fω:

```
κ ::= * | κ → κ
```

FC2:

```
η ::= * | κ → η
κ ::= <η/TYPE> | <η/CODE>
```

```
type family R a
type instance R String = ByteArray
type instance R Tel = Int64
```

Source

```
R : (<*/CODE> → *)/CODE
R String ~ ByteArray : */CODE
R Tel ~ Int64 : */CODE
```

FC2 axioms

Look ma, no special lifting!

- Lifting equalities must simply be kind respecting:

$$\frac{(\mathrm{T} : <*/\rho> \Rightarrow *) \in \Gamma \qquad \Gamma \vdash \tau \sim \sigma : */\rho}{\Gamma \vdash \mathrm{T}\,\tau \sim \mathrm{T}\,\sigma : */\mathrm{TYPE}}$$

- Actual rule is more general but the above simplification conveys the intentions!

Why does that fix the problem?

YOUR TAKEAWAY #2

$$\frac{\begin{array}{c}(*/\rho > \Rightarrow *) \in \Gamma \\ \vdash\ \tau \sim \sigma : */\rho\end{array}}{\tau \sim T\,\sigma : */\mathrm{TYPE}}$$

Impossible to derive
`R String ~ R Tel : */TYPE`
… because R expects a CODE equality!

```
Tel ~ String : */TYPE
Tel ≁ String : */CODE

R : (<*/CODE> → *) ∈ Γ
```

Lifting over type constructors

$$\frac{\begin{array}{c}(\mathrm{T} : <*/\rho> \Rightarrow *) \in \Gamma \\ \Gamma \ \vdash \ \tau \sim \sigma : */\rho\end{array}}{\Gamma \ \vdash \ \mathrm{T}\,\tau \sim \mathrm{T}\,\sigma : */\mathrm{TYPE}}$$

```
Tel ~ String : */TYPE
Tel ≁ String : */CODE
R : (<*/CODE> → *) ∈ Γ

data TR a    = MkTR (R a)
data List a = Nil | Cons a (List a)
```

Similarly:

```
      TR : (<*/CODE> → *)
```

Hence:

```
      TR Tel ≁ TR String : */TYPE
```

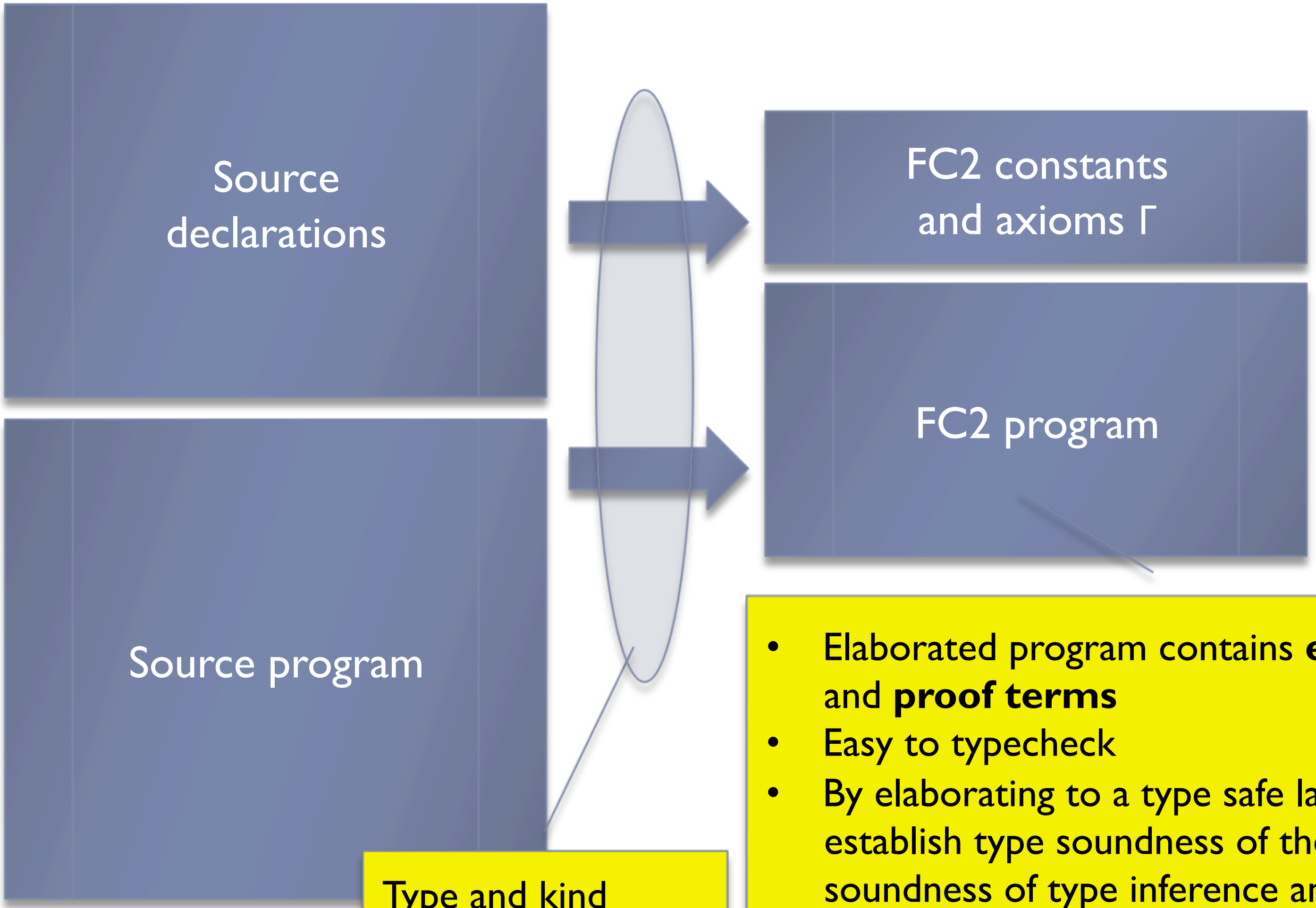
BUT:

```
      List : (<*/TYPE> → *)
```

Hence:

```
      List Tel ~ List String : */
TYPE
```

FC2: The formal setup



Type and kind (and role) inference

- Elaborated program contains **explicit types** and **proof terms**
- Easy to typecheck
- By elaborating to a type safe language we establish type soundness of the source and soundness of type inference and soundness of compiler transformations

FC2 typing judgements

- All equalities have explicit proof witnesses. Three judgements:

$$\Gamma \vdash e : \tau$$

Role ρ ::= TYPE | CODE

$$\Gamma \vdash \tau : \eta/\rho$$

$$\tau \;::=\; a \mid T\,\overline{\tau} \mid \forall a{:}\kappa.\tau \mid \tau \sim \sigma \Rightarrow \varphi$$

Coercion abstractions

$$\Gamma \vdash \gamma : \tau \sim \sigma : \eta/\rho$$

$$\gamma \;::=\; id_{\tau} \mid sym\,\gamma \mid c \mid C \mid \gamma_1;\gamma_2 \mid \mathrm{T}\,\gamma \mid \mathrm{nth}\; i\;\gamma$$

Coercions γ: Equality proof witnesses

- Typing rule that connects typing and coercions in FC2:

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \gamma : \tau \sim \sigma : * \,/\, \mathrm{TYPE}}{\Gamma \vdash (e \rhd \boldsymbol{\gamma}) : \sigma}$$

Type-soundness via consistency

- Based on progress and subject reduction, using a semantics that "pushes" coercions:

$$\frac{\gamma_1 = nth\ 1\ \gamma \qquad \gamma_0 = nth\ 0\ \gamma}{\big((\lambda x{:}\tau.e_1)\rhd\gamma\big)\ e_2 \quad\longrightarrow\quad (\lambda x{:}\tau.e_1\rhd\gamma_1)\ (e_2\rhd sym\ \gamma_0)}$$

We know that:

γ : (τ → σ) ~ (τ' → σ')

Hence:

γ1 : σ ~ σ'

Hence:

γ0 : τ ~ τ'

Hence:

sym γ0 : τ' ~ τ

- Progress is proven with the assumption of **consistency**:

*A context Γ is consistent iff whenever $\Gamma \vdash \gamma : \tau \sim \sigma : \eta/\mathrm{TYPE}$ is derived and τ, σ are value types, and τ is a datatype application (T φ) then σ is also **the same** datatype application (T φ')*

Establishing consistency

- Step 1
  - Define a role-sensitive type rewrite relation
  - [Novel idea: don't require strong normalization of axioms, but require instead more determinism]
- Step 2
  - Prove soundness and completeness of the type rewrite relation wrt the coercibility relation
- Step 3:
  - Show that rewriting preserves head value constructors

See paper and extended version for the gory details

More interesting details in the paper

-

- I've talked about coercion lifting, but when is coercion decomposition safe? And under which roles?

$$\frac{\Gamma \ \vdash \mathrm{T}\,\varphi \sim \mathrm{T}\,\psi : * \ /\ \mathrm{TYPE}}{\Gamma \ \vdash \varphi \sim \psi : ????}$$

- FC2 typing rules are not formulated with only two universes (TYPE / CODE) but allow a semi-lattice of universes – perhaps a nice way to incorporate safely many notions of equality?

Is this all Haskell specific?

No, though no other existing language demonstrates the same problem today so Haskell is a good motivation

But:

- Type generativity via some mechanism is useful
- Type-level computation is independently useful
- GHC happened to arrive at this situation early

Sooner or later, as soon as both these features are in your type system you have to look for a solution

Lots of exciting future directions

- Present a semantics that justifies the proof theory of FC2
- Shed more light into coercion decomposition:
  - Injectivity of constructors admissible in Fω but not derivable (conj.)
  - Hence in need of semantic justification for the decomposition rules
  - Direction: Extend the kinds of Fω with roles and type functions, and encode equalities as Leibniz equalities. Can this shed any more light? What are the parametric properties of that language?
- Enrich the universe of codes with term constructors
- Investigate other interesting equalities (e.g. syntactic, β)
  - Can roles help in security and information flow type systems where different equalities may arise from different confidentiality levels?
- Develop source language technology to give programmers control over the kinds of their declarations

Thank you for your attention

Questions?