Type checking and normalisation

James Chapman - University of Nottingham

My thesis

- Type checking in Haskell
  - Epigram language - McBride/McKinna

- Big-step normalisation for simple types, formalised in Agda
- Big-step normalisation for dependent types, partially formalised in Agda

Goal of my current work

- To write a correct-by-construction type checker for type theory in type theory
- Epigram in Epigram/Agda in Agda

A brief history of type theory

Brouwer's intuitionism

- Start again with a new mathematics
- Rejects some classical laws:
  - Excluded middle: $A \vee \neg A$
  - Proof by contradiction: $A \Leftrightarrow \neg\neg A$

BHK Interpretation

- A proof of A → B is a function which converts proofs of A into proofs of B
- A proof of A ∧ B is a pair of a proof of A and a proof of B
- A proof of A ∨ B is either a proof of A or a proof of B
- A proof of ∃x . P x means a pair of a witness x and a proof that x satisfies P

Heyting - Intuitionistic logic

Intuitionistic Logic

Proof	=	Program
Proposition	=	Specification
Proposition	≠	Type

Is intuitionistic logic enough?

We can prove theorems by writing programs:

```
d : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C)
d (a , left  b) = left  (a , b)
d (a , right c) = right (a , c)
```

But we cannot reason about proofs:

```
e  :  ∀ p , q : A ∧ (B ∨ C) →
            (d p == d q) → (p == q)
e = ?
```

A full-scale intuitionistic system

- Can quantify over proofs of a proposition (Sigma and Pi types)
- Satisfies BHK interpretation of logic
- Started by Howard
  - "The formulae-as-types notion of construction"
- Finished by Martin-Löf
  - "an intuitionistic theory of types"

A full-scale intuitionistic system (2)

Proof	=	Program
Proposition	=	Type
Cut elimination	=	Normalisation
Proof checking	=	Type checking

Present day

- Type theory is used (and still being refined) for theorem proving, certified software and dependently typed programming
- Coq - developed at INRIA France
  - Essential for proving Four Color Theorem
  - Common Criteria certification of JavaCard
- Agda - developed at Chalmers Sweden
  - Prototype dependently typed programming language (used in this talk)

Why certify a type checker?

- Certified certification
- Introspective language design
- Intuitionistic metatheory

Certified certification

- It’s the central component of systems for developing certified software. A faulty checker might accept faulty certificates
- Coq is used for industrial strength certification
- HOL-Light’s 100 line type checker had a bug in it for 15 years. Uncovered by Flyspeck project
- No fixed idea about what can be implemented in type theory. E.g. Verified tactics

Introspective language design

- Sound engineering approach:
  - E.g. write a C compiler in C
  - Often first big test for a language
- Synergy between language design and implementation
  - E.g. GHC leading the development of Haskell

Intuitionistic metatheory

- A type checker includes and executable semantics for type theory
- Martin-Löf worked in an informal intuitionistic metalanguage
- Working formally gives:
  - Computer assistance
  - High assurance
- Another synergy here: E.g. induction recursion

Type checking and normalisation

Lists

```
data List (A : Set) : Set where
  nil  : List A
  cons : A → List A → List A

app : List A → List A → List A
app nil         ws = ws
app (cons v vs) ws = cons v (app vs ws)
```

Typing constraints from definition of app:

```
List A = List A
List A = List A
```

Vectors

```
data Vec (A : Set) : Nat → Set where
  nil  : Vec A zero
  cons : A → Vec A n → Vec A (suc n)

app : Vec A m → Vec A n → Vec A (m + n)
app nil         ws = ws
app (cons v vs) ws = cons v (app vs ws)
```

Typing constraints from definition of app:

```
Vec A (zero  + n) = Vec A n
Vec A (suc m + n) = Vec A (suc (m + n))
```

Well typed syntax (1)

Example: Simply typed lambda calculus

Types are base ι or σ → τ, contexts are lists of types

```
data Tm : Con → Ty → Set where
   var : Var Γ σ → Tm Γ σ
   app : Tm Γ (σ → τ) → Tm Γ σ
         Tm Γ τ
   λ   : Tm (Γ , σ) τ → Tm Γ (σ → τ)
```

We express the syntax and the type system in one

Well typed syntax (2)

Well scoped nameless variables (de Bruijn indices)

```
data Var : Con → Ty → Set where
  top : Var (Γ , σ) σ
  pop : Var Γ σ → (τ : Ty) →
        Var (Γ , τ) σ
```

Never have to deal with dangling variables

Big-step normalisation

- Define a partial function `nf : Tm → Nf` such that it satisfies the following properties:
  - termination: `∀t.∃n.nf t ⇓ n`
  - completeness: `emb (nf t) ≅ t`
  - soundness: `t ≅ u → nf t = nf u`

BSN compared

- Traditional small-step normalisation
  - Not how you'd implement it
  - Doesn't work well with βη-equality
- Normalisation-by-evaluation (NBE)
  - Practical approach to βη-normalisation
  - Everything at once, higher order
- Big-step normalisation (BSN)
  - A variation on NBE
  - Separates computation and termination, first order

Big-step normalisation for Abadi, Cardelli and Curien's λ^{σ}-calculus

Joint work with Thorsten Altenkirch

λ^σ syntax

We have a well typed syntax of terms and substitions

```
-- conventional lambda and application
λ     : Tm (Γ , σ) τ → Tm Γ (σ → τ)
app   : Tm Γ (σ → τ) → Tm Γ σ → Tm Γ τ

-- variables and expl. substitution
top   : Tm (Γ , σ) σ
_[_]  : Tm Δ σ → Sub Γ Δ → Tm Γ σ

-- explicit weakening
↑σ : Sub (Γ , σ) Γ
```

Equational theory

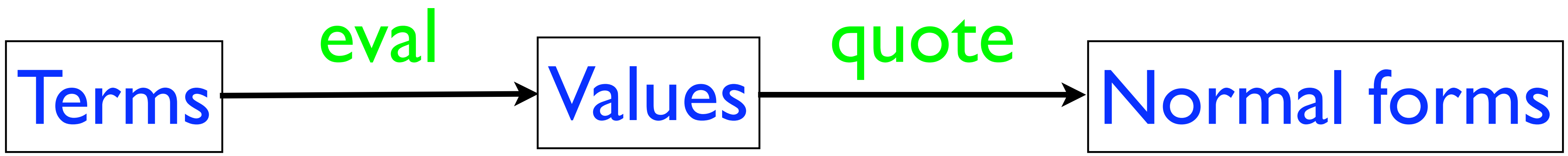
We can write down the laws of the equational theory as an inductively defined relation on terms and substitutions

```
subid  : t [ id ] ≅ t
β      : app (λ t) u ≅ t [ id , u ]
proj   : top [ ts , t ] ≅ t

compid : ts • id ≅ ts
```

Implementing the normaliser

We do this in two stages:



Values

are either lambda closures or neutral (stuck) terms

```
data Val : Con → Ty → Set where

  λv : Tm (Δ , σ) τ → Env Γ Δ →
       Val Γ (σ → τ)

  ne : Ne Γ τ → Val Γ τ
```

Evalution (1)

We define the following operations mutually:

```
eval  : Tm  Δ σ → Env Γ Δ → Val Γ σ
seval : Sub Δ E → Env Γ Δ → Env Γ E
_@@_  : Val Γ (σ → τ) →
        Val Γ σ → Val Γ τ
```

Afterwards we prove that they terminate

Evaluation (2)

```
eval : Tm Δ σ → Env Γ Δ → Val Γ σ
eval (λ t)        vs         = λv t vs
eval (app t u)    vs         =
  (eval t vs) @@ (eval u vs)
eval top          (vs , v) = v
eval (t [ ts ]) vs           =
  eval t (seval ts vs)

_@@_ : Val Γ (σ → τ) → Val Γ σ → Val Γ τ
λv t vs @@ v = eval t (vs , v)
ne n    @@ v = ne (app n a)
```

β-normal η-long forms

are either lambda-abstraction or neutral embedded at base type

```
data Nf : Con → Ty → Set where

  λn : Nf (Δ , σ) τ → Nf Γ (σ → τ)

  ne : Ne Γ ι → Nf Γ ι
```

Quote

is defined by recursion on types.

```
quote : Val Γ σ → Nf Γ σ

-- quote the components
quoteι          (ne n) = ne (nquote n)

-- perform eta expansion
quote(σ → τ) f          =
  λn (quoteτ (wk f @@ top)))
```

The normaliser

Having defined `eval` and `quote` we can define:

$$\texttt{nf} : \texttt{Tm}\ \Gamma\ \sigma \rightarrow \texttt{Nf}\ \Gamma\ \sigma$$

$$\texttt{nf}\ t = \texttt{quote}_{\sigma}\ (\texttt{eval}\ t\ \texttt{id}_{\Gamma})$$

which is:

- terminating (proof using strong computability)
- sound (proof using logical relations)
- and complete (simple induction on big-step relation)

BSN Summary

- Write a partial normaliser function
- Prove termination over graph (big-step semantics) of partial function
- Combine function and termination proof using Bove-Capretta technique to get a total function
- Note: Termination proof doesn't add computational content, computational behaviour remains the same as the partial function

Dependent types

- Can extend this approach to dependent types but equational theory is now mutually defined with the syntax:

$$\frac{\mathtt{t} : \mathtt{Tm}\ \Gamma\ \sigma \qquad \mathtt{p} : \sigma \cong \sigma'}{\mathtt{coe}\ \mathtt{t}\ \mathtt{p} : \mathtt{Tm}\ \Gamma'\ \sigma'}\ \text{conversion rule}$$

- Also contexts, types, terms and substitutions must be mutually defined

Related work

- Simple types
  - NBE for λ^{σ}-calculus - C. Coquand
- Dependent types
  - Coq-in-Coq - Barras
  - Internal Type Theory - Dybjer
  - NBE for Martin-Löf's logical framework - Danielsson

Conclusion

- Dependent typed languages are suited to implementing languages and much more
- I want to verify a type checker for higher assurance, and to explore both dependently typed programming and intuitionistic metatheory
- Big-step normalisation is a practical, scalable approach to normalisation