Structural Recursion over Contextual Objects

Sherry Shanshan Ruan[1] Brigitte Pientka[1] Andreas Abel[2]

McGill University[1] Chalmers University of Technology[2]

September 2, 2014

Logical Framework

Logical framework (LF) is an elegant system for specifying formal systems via axioms and inference rules [Harper et al., 1993]

- Model variables in the object language by variables in LF
- Model binders in the object language by binders in LF

Logical Framework

Logical framework (LF) is an elegant system for specifying formal systems via axioms and inference rules [Harper et al., 1993]

- Model variables in the object language by variables in LF
- Model binders in the object language by binders in LF

Object Language	$\mathit{term}\ M, N := x \mid \mathit{lam}\ x.M \mid M\ N$
Representation in LF	$\textbf{datatype}\ \texttt{term} : \textbf{type} =$ $\mid\ \ \mathtt{app} : \mathtt{term} \to \mathtt{term} \to \mathtt{term}$ $\mid\ \ \mathtt{lam} : (\mathtt{term} \to \mathtt{term}) \to \mathtt{term}$

Logical Framework

Logical framework (LF) is an elegant system for specifying formal systems via axioms and inference rules [Harper et al., 1993]

- Model variables in the object language by variables in LF
- Model binders in the object language by binders in LF

Object Language	$term\ M, N := x \mid lam\ x.M \mid M\ N$
Representation in LF	$\textbf{datatype}\ \texttt{term} : \textbf{type} =$ $\mid\ \ \texttt{app} : \texttt{term} \to \texttt{term} \to \texttt{term}$ $\mid\ \ \texttt{lam} : (\texttt{term} \to \texttt{term}) \to \texttt{term}$

$$lam\ x.lam\ y.x\ y \leftrightsquigarrow \texttt{lam}\ (\lambda \texttt{x}.\texttt{lam}\ (\lambda \texttt{y}.\texttt{app x y}))$$

Logical Framework

Logical framework (LF) is an elegant system for specifying formal systems via axioms and inference rules [Harper et al., 1993]

- Model variables in the object language by variables in LF
- Model binders in the object language by binders in LF

Object Language $\quad term\ M, N := x \mid lam\ x.M \mid M\ N$

Representation in LF

$$\begin{array}{l} \textbf{datatype}\ \texttt{term} : \textbf{type} = \\ \mid\ \ \texttt{app} : \texttt{term} \to \texttt{term} \to \texttt{term} \\ \mid\ \ \texttt{lam} : (\texttt{term} \to \texttt{term}) \to \texttt{term} \end{array}$$

$$lam\ x.lam\ y.x\ y \leftrightsquigarrow \texttt{lam}\ (\lambda\texttt{x}.\texttt{lam}\ (\lambda\texttt{y}.\boxed{\text{I'm dependent on x and y}}))$$

Contextual Modal Type Theory

- As we recursively traverse higher-order abstract syntax trees, we extend the context of assumptions and LF object does not remain closed

Contextual Modal Type Theory

- As we recursively traverse higher-order abstract syntax trees, we extend the context of assumptions and LF object does not remain closed
- Contextual object $[\Gamma \vdash M]$ pairs an open term M together with the context Γ in which it is meaningful [Nanevski et al., 2008]

Contextual Modal Type Theory

- As we recursively traverse higher-order abstract syntax trees, we extend the context of assumptions and LF object does not remain closed
- Contextual object $[\Gamma \vdash M]$ pairs an open term M together with the context Γ in which it is meaningful [Nanevski et al., 2008]
- Beluga is a programming environment for reasoning about formal systems in contextual LF [Cave and Pientka, 2012]

Contextual Modal Type Theory

- As we recursively traverse higher-order abstract syntax trees, we extend the context of assumptions and LF object does not remain closed
- Contextual object $[\Gamma \vdash M]$ pairs an open term M together with the context Γ in which it is meaningful [Nanevski et al., 2008]
- Beluga is a programming environment for reasoning about formal systems in contextual LF [Cave and Pientka, 2012]
- However, Beluga lacks guarantees that a given program is total

Contextual Modal Type Theory

- As we recursively traverse higher-order abstract syntax trees, we extend the context of assumptions and LF object does not remain closed
- Contextual object $[\Gamma \vdash M]$ pairs an open term M together with the context Γ in which it is meaningful [Nanevski et al., 2008]
- Beluga is a programming environment for reasoning about formal systems in contextual LF [Cave and Pientka, 2012]
- However, Beluga lacks guarantees that a given program is total

My work:

Develop a well-founded recursion principle and provide a consistency proof

Example: type uniqueness

$$\textit{Type}\ T, S := \textit{bool} \mid T \to S \qquad \textit{Term}\ M, N := x \mid \textit{lam}\ x{:}T.M \mid M\ N$$

Example: type uniqueness

$$Type\ T, S := bool \mid T \to S \qquad Term\ M, N := x \mid lam\ x{:}T.M \mid M\ N$$

datatype `tp` : **type** =
| `bool` : `tp`
| `arr` : `tp` → `tp` → `tp` ;

datatype `tm` : **type** =
| `lam` : `tp` → (`tm` → `tm`) → `tm`
| `app` : `tm` → `tm` → `tm` ;

Example: type uniqueness

$$Type\ T, S := bool \mid T \to S \qquad Term\ M, N := x \mid lam\ x{:}T.M \mid M\ N$$

datatype `tp` : **type** =
| `bool` : `tp`
| `arr` : `tp` → `tp` → `tp` ;

datatype `tm` : **type** =
| `lam` : `tp` → (`tm` → `tm`) → `tm`
| `app` : `tm` → `tm` → `tm` ;

$$\frac{\Gamma \vdash M : T \to S \qquad \Gamma \vdash N : T}{\Gamma \vdash M\ N : S}\ t\_app \qquad \frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash lam\ x{:}T.M : T \to S}\ t\_lam$$

Example: type uniqueness

$$Type\ T, S := bool \mid T \to S \qquad Term\ M, N := x \mid lam\ x{:}T.M \mid M\ N$$

datatype `tp` : **type** =
| `bool` : `tp`
| `arr` : `tp` → `tp` → `tp` ;

datatype `tm` : **type** =
| `lam` : `tp` → (`tm` → `tm`) → `tm`
| `app` : `tm` → `tm` → `tm` ;

$$\frac{\Gamma \vdash M : T \to S \qquad \Gamma \vdash N : T}{\Gamma \vdash M\ N : S}\ t\_app \qquad \frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash lam\ x{:}T.M : T \to S}\ t\_lam$$

datatype `oft` : `tm` → `tp` → **type** =
| `t_app` : `oft M (arr T S)` → `oft N T` → `oft (app M N) S`
| `t_lam` : (Π `x:tm. oft x T` → `oft (M x) S`) → `oft (lam T M) (arr T S)` ;

Example: type uniqueness

$$\textit{Type } T, S := \textit{bool} \mid T \to S \qquad \textit{Term } M, N := x \mid \textit{lam } x{:}T.M \mid M\ N$$

datatype `tp` : **type** =
| `bool` : `tp`
| `arr` : `tp` → `tp` → `tp` ;

datatype `tm` : **type** =
| `lam` : `tp` → (`tm` → `tm`) → `tm`
| `app` : `tm` → `tm` → `tm` ;

$$\frac{\Gamma \vdash M : T \to S \qquad \Gamma \vdash N : T}{\Gamma \vdash M\ N : S}\ \textit{t\_app} \qquad \frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash \textit{lam } x{:}T.M : T \to S}\ \textit{t\_lam}$$

datatype `oft` : `tm` → `tp` → **type** =
| `t_app` : `oft M (arr T S)` → `oft N T` → `oft (app M N) S`
| `t_lam` : `(Π x:tm. oft x T → oft (M x) S)` → `oft (lam T M) (arr T S)` ;

Theorem (type uniqueness): If $\Gamma \vdash E : T$ and $\Gamma \vdash E : S$, then *eq* T S

Example: type uniqueness

$$\textit{Type } T, S := \textit{bool} \mid T \to S \qquad \textit{Term } M, N := x \mid \textit{lam } x{:}T.M \mid M\ N$$

```
datatype tp : type =
| bool : tp
| arr : tp → tp → tp ;
```

```
datatype tm : type =
| lam : tp → (tm → tm) → tm
| app : tm → tm → tm ;
```

$$\frac{\Gamma \vdash M : T \to S \qquad \Gamma \vdash N : T}{\Gamma \vdash M\ N : S}\ t\_app \qquad \frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash \textit{lam } x{:}T.M : T \to S}\ t\_lam$$

```
datatype oft : tm → tp → type =
| t_app : oft M (arr T S) → oft N T → oft (app M N) S
| t_lam : (Π x:tm. oft x T → oft (M x) S) → oft (lam T M) (arr T S) ;
```

Theorem (type uniqueness): If $\Gamma \vdash E : T$ and $\Gamma \vdash E : S$, then $eq\ T\ S$

$$(\Gamma{:}ctx)\ [\Gamma \vdash oft\ (E..)\ T] \to [\Gamma \vdash oft\ (E..)\ S] \to [\,\vdash eq\ T\ S]$$

Example: type uniqueness

$\mathbf{rec}\ \texttt{unique} : (\Gamma{:}\texttt{ctx})[\Gamma \vdash \texttt{oft}\ (\texttt{E}..)\ \texttt{T}]^* \to [\Gamma \vdash \texttt{oft}\ (\texttt{E}..)\ \texttt{S}] \to [\ \vdash \texttt{eq}\ \texttt{T}\ \texttt{S}] =$

Example: type uniqueness

```
rec unique : (Γ:ctx)[Γ ⊢ oft (E..) T]* → [Γ ⊢ oft (E..) S] → [ ⊢ eq T S] =
fn d ⇒ fn f ⇒ case d of

|  [Γ ⊢ t_app (D1..) (D2..)] ⇒
     let [Γ ⊢ t_app (F1..) (F2..)] = f in
     let [ ⊢ e_ref] = unique [Γ ⊢ D1..][Γ ⊢ F1..] in
     [ ⊢ e_ref]

|  [Γ ⊢ t_lam (λx.λu. D.. x u)] ⇒
     let [Γ ⊢ t_lam (λx.λu. F.. x u)] = f in
     let [ ⊢ e_ref] = unique [Γ, b:block x:tm, t:oft x _ ⊢ D .. b.1 b.2]
                             [Γ, b ⊢ F .. b.1 b.2] in
     [ ⊢ e_ref]

|  [Γ ⊢ #q.2..] ⇒                                      % d : oft #q.1 T
     let [Γ ⊢ #r.2 ..] = f in                         % f : oft #r.1 S
     [ ⊢ e_ref] ;
```

Contributions

Totality = Coverage + Termination

Contributions

$$\text{Totality} = \text{Coverage} + \text{Termination}$$

Theory:

- Describe a measure for contextual objects (Context weakening and reordering)

Contributions

$$\text{Totality} = \text{Coverage} + \text{Termination}$$

Theory:

- Describe a measure for contextual objects (Context weakening and reordering)
- Design an algorithm for the generation of simultaneous pattern matching constructs and valid structural recursive calls

Contributions

$$\text{Totality} = \text{Coverage} + \text{Termination}$$

Theory:

- Describe a measure for contextual objects (Context weakening and reordering)
- Design an algorithm for the generation of simultaneous pattern matching constructs and valid structural recursive calls
- Prove weak normalization using logical relations

Contributions

$$\text{Totality} = \text{Coverage} + \text{Termination}$$

Theory:

- Describe a measure for contextual objects (Context weakening and reordering)
- Design an algorithm for the generation of simultaneous pattern matching constructs and valid structural recursive calls
- Prove weak normalization using logical relations

Implementation:

- Simple generation of recursive calls on contextual objects works (Support many proofs including type preservation, value soundness, type uniqueness)

Contributions

$$\text{Totality} = \text{Coverage} + \text{Termination}$$

Theory:

- Describe a measure for contextual objects (Context weakening and reordering)
- Design an algorithm for the generation of simultaneous pattern matching constructs and valid structural recursive calls
- Prove weak normalization using logical relations

Provides a **generic** proof language for first-order logic with a domain-specific recursion principle and a consistency proof

Future Work

Future Work

Theory:

- Support lexicographical orderings and mutual recursion

Future Work

Theory:

- Support lexicographical orderings and mutual recursion
- Generate recursive calls on recursively defined objects

Future Work

Theory:

- Support lexicographical orderings and mutual recursion
- Generate recursive calls on recursively defined objects
- Design a recursion principle for more general recursive calls

Future Work

Theory:

- Support lexicographical orderings and mutual recursion
- Generate recursive calls on recursively defined objects
- Design a recursion principle for more general recursive calls

Implementation:

- Generate more sophisticated recursive calls

References

- A. Cave and B. Pientka. Programming with binders and indexed data-types. In 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL12), pages 413 424. ACM Press, 2012.
- R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Journal of the ACM, 40(1): 143184, January 1993.
- A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. ACM Transactions on Computational Logic, 9(3):149, 2008.

Thank You