Bringing Higher-Order Abstract Syntax to Programmers Compiling Contextual Objects

Francisco Ferreira

McGill University and Université de Montréal

Compiling Beluga and Contextual Objects

- Beluga [Pientka and Dunfield, 2010] and [Pientka, 2008] is a functional programming language that supports binders in its data declarations using contextual objects.
- Compiling contextual objects and higher-order abstract syntax, requires choosing a concrete internal representation. We use a very versatile one that allows us to generate code in two styles, code using names and de Bruijn indices for bound variables.

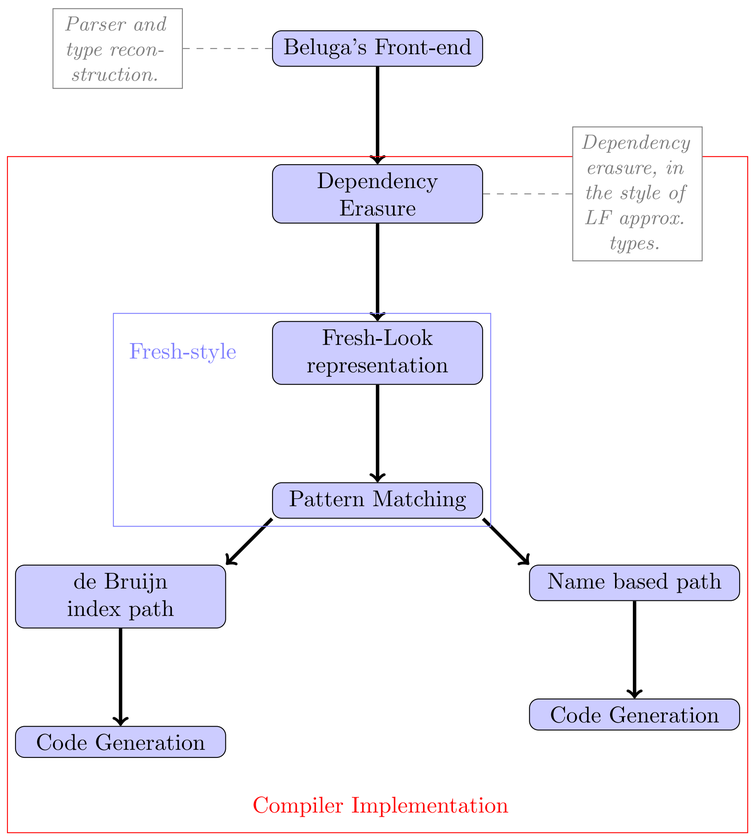
Contextual Objects

- Contextual LF [Nanevski *et al.*, 2008] is an extension of the LF logical framework.
- In Contextual LF terms and types carry around the context where they exist.
- **[Ψ**. M**]** denotes a LF object M in a context **Ψ** and has type [**Ψ**]A.
- These objects are used to support Higher-Order Abstract Syntax in the Beluga Language.

Contribution

- A framework for compiling contextual objects.
- The compilation of pattern matching of contextual objects, able to match inside binders.
- A novel connection between nominal techniques and contextual modal type theory.

The Compiling Pipeline



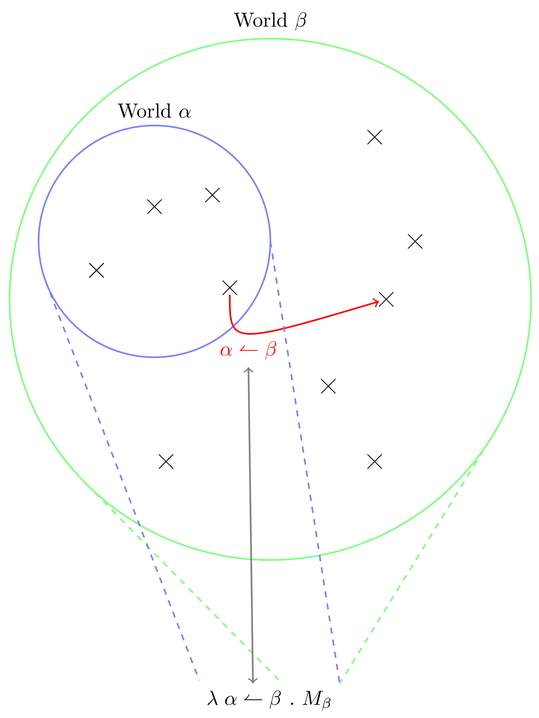
An Example Beluga Program

```
nat  : type.                                          LF-Style data type declaration.
z    : nat.
s    : nat → nat.

exp  : type.
num  : nat → exp.
app  : exp → exp → exp.
lam  : (exp → exp) → exp.
exp' : type.
num' : nat → exp'.
app' : exp' → exp' → exp'.
lam' : exp' → exp'.
one  : exp'.
shift : exp' → exp'.

schema ctx = exp ;

rec hoas2db : (g:ctx) [g . exp] → [. exp'] =          Recursive function to manipulate data.
fn e ⇒ case e of
        |[g, x:exp . x] ⇒ [. one]
        |[g, x:exp . #p..] ⇒
            let [. M'] = hoas2db ([g . #p..]) in
            [. shift M']
        |[g . app (M..) (N..)] ⇒
            let [. M'] = hoas2db [g. M..] in
            let [. N'] = hoas2db [g. N..] in
            [. app' M' N']
        |[g . lam(λx.M..x)] ⇒
            let [. M'] = hoas2db [g, x:exp. M..x] in
            [. lam' M']
        |[g . num X] ⇒ [. num' X]
;
```

We compile these functions!

The Fresh Style Internal Representation

- The compiler uses a representation based on [Pouillard and Pottier, 2010] that must be converted to concrete representations in the resulting code.
- Thanks to this representation pattern matching uses abstract names and binders and it is oblivious to the final representation.
- Many optimizations can use this representation, (e.g. dead code elimination). However, there are no optimizations in the current schema, so which optimizations can be appropiately expressed in the framework is part of the future work.
- In the final stages the compiler generates code that uses either unique names or de Bruijn indices.



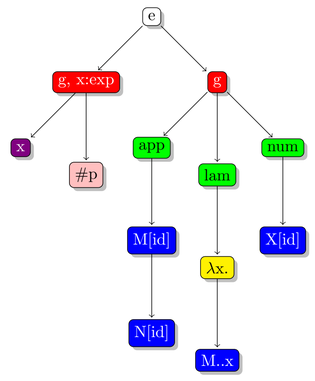
- In the compiler, bound variables in contextual objects are represented using the fresh-style representation. In this representation we have:
  - Variables that are represented as abstract names.
  - Worlds that names inhabit.
  - Links that relate two worlds and introduce a new name in the bigger world.
  - New binders (e.g. in **λ** expressions), are links from the current world to a world.
  - A chain of links may start from the empty world, or from an *abstract* world that reprsents a contextual variable. This is different from the original representation and necessary for representing contextual objects.

Pattern Matching Compilation

In order to support matching in contextual objects bound variables and inside binders, Beluga has a very rich pattern matching language.
Beluga supports:

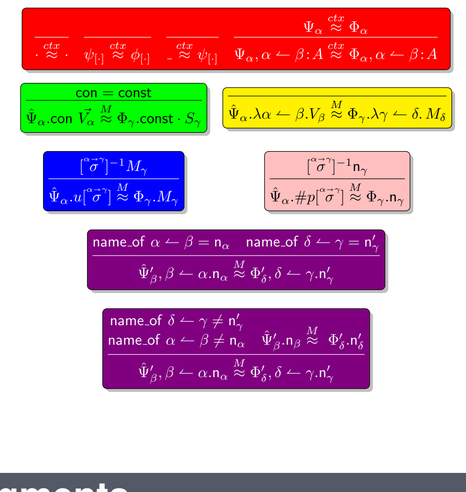
- Matching on the shape of contexts.
- Matching on specific bound variables.
- Matching on any bound variable.
- Matching on meta-variables and substitutions.
- Matching inside **λ** expressions.
- Matching on constructors.

Pattern matching compilation is based on computing the splitting tree as in [Maranget, 2008]. Because of the rich patterns, there are more rules to split on besides variables and constructors. Additionally, during runtime determining whether a value matches a pattern is more complex, the rules for performing such matchin are presented in the figure below right.




```
rec hoas2db : (g:ctx) [g. exp] → [. exp'] =
fn e ⇒ case e of
        |[ g, x:exp . x ] ⇒ [. one]
        |[ g,x:exp . #p.. ] ⇒
            let [. M'] = hoas2db ([g. #p..]) in
            [. shift M']
        |[ g . app (M..) (N..)] ⇒
            let [. M'] = hoas2db [g. M..] in
            let [. N'] = hoas2db [g. N..] in
            [. app' M' N']
        |[ g . lam (λx. M..x)] ⇒
            let [. M'] = hoas2db [g, x:exp. M..x] in
            [. lam' M']
        |[ g . num X ] ⇒ [. num' X]
;
```

- The computation of the tree (e.g. the tree below left) is done at compile time in the fresh style representation.
- The runtime comparisons are a bit more complex than for ML-style patterns. Comparisonsy obey the rules in the image below.



Conclusions and Future Work

- The compiler uses a versatile representation with abstract names that bridges higher-order to first-order representations and relates nominal approaches to higher-order abstract syntax. It supports pattern compilation for rich patterns that is able to match inside of contextual objects.
- As future work, it is pending to explore keeping the dependent types (a.k.a. removing the dependency erasure phase), studying the relationship between coverage checking and pattern matching under HOAS, and doing a performance evaluation of the name and the de Bruijn index backends.

Bibliography

- Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08. ACM, 2008.
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3), 2008.
- Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, 2010.
- Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, 2008.

Acknowledgments

- This work is possible thanks to the guidance and support of my advisors Brigitte Pientka and Stefan Monnier.
- Finally, this research was supported by Fonds de Recherche du Québec (FQRNT) and the Natural Sciences and Engineering Research Council of Canada(NSERC).

fferre8@cs.mcgill.ca