GF as functional-logic language

Krasimir Angelov

Chalmers University of Technology

April 16, 2010

The Dichotomy in GF

- Abstract Syntax
  - Defines the abstract ontological structure of the domain
  - Turing-complete functional language
  - Dependent types
- Concrete Syntax
  - Defines a rendering of the abstract syntax into some language
  - Restricted recursion-free functional language
  - Simpe polymorphic types, but - overloading, records, subtyping

Note: In this talk we will focus on the abstract syntax

Abstract Syntax

Turing-complete functional language:

$$
\begin{array}{l}
\textbf{abstract}\ \mathit{Nat} = \{ \\
\quad \textbf{cat}\ \mathit{Nat}; \\
\quad \textbf{data}\ \mathit{zero} : \mathit{Nat}; \\
\qquad \mathit{succ} : \mathit{Nat} \longrightarrow \mathit{Nat}; \\
\quad \textbf{fun}\ \mathit{plus} : \mathit{Nat} \longrightarrow \mathit{Nat} \longrightarrow \mathit{Nat}; \\
\quad \textbf{def}\ \mathit{plus}\ \mathit{zero}\ n = n; \\
\qquad \mathit{plus}\ (\mathit{succ}\ m)\ n = \mathit{succ}\ (\mathit{plus}\ m\ n); \\
\}
\end{array}
$$

Concrete Syntax

As an example a natural number in ASCII is a sequence of underscores.

$$\textbf{concrete}\ \mathit{NatAscii}\ \textbf{of}\ \mathit{Nat} = \{$$

$$\textbf{lincat}\ \mathit{Nat} = \mathit{Str};$$

$$\textbf{lin}\ \mathit{zero} = "";$$

$$\mathit{succ}\ x = "\_" +\!\!+\ x;$$

$$\}$$

Note: We will use this in the N-Queens solver

Type System

The abstract syntax is a **first-order type theory:**

- dependent types - $(x : A) \to B\ x$
- implicit arguments - $(\{x, y\} : A) \to B$ New!
- inaccessible patterns - $(\sim x)$ New!

Note: The last two were introduced only in the last months. This features are borrowed from Agda but the syntax is changed to avoid ambiguities.

Example: Dependent Types & Implicit Arguments

$$\textbf{cat}\ \mathit{Category};$$
$$\mathit{Obj}\ \mathit{Category};$$
$$\mathit{Arrow}\ (\{c\} : \mathit{Category})\ (\mathit{Obj}\ c)\ (\mathit{Obj}\ c);$$

$$\textbf{fun}\ \mathit{dom} : (\{c\} : \mathit{Category}) \to (\{x, y\} : \mathit{Obj}\ c) \to \mathit{Arrow}\ x\ y \to \mathit{Obj}\ c;$$
$$\textbf{def}\ \mathit{dom}\ \{x\}\ \{y\}\ \_\ = x;$$

$$\textbf{fun}\ \mathit{codom} : (\{c\} : \mathit{Category}) \to (\{x, y\} : \mathit{Obj}\ c) \to \mathit{Arrow}\ x\ y \to \mathit{Obj}\ c$$
$$\textbf{def}\ \mathit{codom}\ \{x\}\ \{y\}\ \_\ = y;$$

Example: Dependent Types & Inaccessible Patterns

$$\textbf{cat}\ EqAr\ (\{c\} : Category)\ (\{x, y\} : Obj\ c)\ (f, g : Arrow\ x\ y);$$

$$\begin{aligned}
\textbf{data}\ eqRefl : {} & (\{c\} : Category) \\
& \to (\{x, y\} : Obj\ c) \\
& \to (f : Arrow\ x\ y) \\
& \to EqAr\ f\ f;
\end{aligned}$$

$$\begin{aligned}
\textbf{fun}\ eqSym : {} & (\{c\} : Category) \\
& \to (\{x, y\} : Obj\ c) \\
& \to (\{f, g\} : Arrow\ x\ y) \\
& \to EqAr\ f\ g \\
& \to EqAr\ g\ f;
\end{aligned}$$

$$\textbf{def}\ eqSym\ \{\sim c\}\ (eqRefl\ \{c\}\ f) = eqRefl\ \{c\}\ f;$$

Why First-Order Type Theory?

Polymorphic types:

$$\textbf{fun}\ id : (A : Type) \rightarrow A \rightarrow A$$

are not allowed, because:

- what is the **lincat** of A?
- parsing with polymorphic types would not be tractable.

Note: this also allows us to use GF as efficient logic-based programming language

Why First-Order Type Theory?

So far this looks like cut down version of Agda with different syntax, but:

- we allow partial definitions
- we want to have nondeterministic computations in the future

Partial Definitions

We could have definition like this:

$$\textbf{fun}\ pred : Nat \longrightarrow Nat;$$
$$\textbf{def}\ pred\ (succ\ x) = x;$$

then what is the value of *pred zero*?

Answer:

$$pred\ zero \rightsquigarrow pred\ zero$$

This lets us to render sentences like this:

The predecessor of zero is not defined

Nondeterminism

Currently only in the **concrete syntax**:

$$\textbf{lin}\ \textit{don't} = \text{"don't"}\ |\ \text{"do not"};$$

, which helps to capture redundancies in NL.

Would be interesting in the **abstract syntax**:

$$\textbf{fun}\ \textit{call\_V} = V\ (\textit{call\_by\_phone\_P}\ |\ \textit{has\_name\_P});$$

, could handle semantic ambiguities.

Note: still not clear how this should interact with the dependent types. Perhaps union types?

Logic Programming in GF

Two of the fundamental functionalities in GF are:

- Exhaustive search for terms of given type
- Random search for term of given type

Note: since we have dependent types the set of all type signatures is a first-order logic program

Exhaustive Search

The generate_tree (gt) command generates all trees of given category:

$ gt -cat=Nat
zero
succ zero
succ (*succ zero*)
...

Note: the term is the stack trace of a logic-based program

Random Search

The generate_random (gt) command generates random tree of given category:

$ gr -cat=Nat -number=3

succ (succ zero)

zero

succ zero

...

Note: running a randomized algorithm

Reconstruction of Parse Trees

Naive approach for semantic restrictions:

cat $Kind;$

$Switchable\ Kind;$

data $light, fan : Kind;$

$switchOn, switchOff : (k : Kind) \rightarrow Switchable\ k \rightarrow Action\ k;$

lin $switchOn\ k\ \_ =$ "switch on" $++ k;$

Wouldn't work (meta variables):

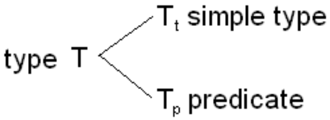
concrete : switch on the ~~bank~~

abstract : *switchOn bank ?*

Solution - Try to prove:

$$Switchable_p\ bank$$

Every nonfunction type could be dissected into a simple type and a predicate:



$$x : T \quad \text{iff} \quad T_p(x) \text{ where } T_p : T_t \rightarrow o$$

The implementation of the predicate requires logic programming and something more than Prolog i.e. Lambda Prolog

Lambda Prolog is an extension of Prolog where:

- the Horn clauses are generalized to Hereditary Harrop formulae
- the programs are statically type checked
- the object terms could have lambda abstractions
- quantification over function symbols is allowed

Hereditary Harrop formulae

Just enough extensions to realize what we need in GF. We will see examples later.

A-formulae (consequent)

any atom

$A :- G$

A, A

pi $x \backslash A$

G-formulae (antecedent, goal)

any atom

$G :- A$

G, G

$G; G$

pi $x \backslash G$

sigma $x \backslash G$

Translation to Lambda Prolog

$$\frac{}{e : C\ e_1 \ldots e_n \vdash C_p\ e\ e_1 \ldots e_n} \quad C \text{ - category}$$

$$\frac{\forall j.\exists i_j.free(x_{i_j})\ x_{i_j} : T_{i_j} \vdash F_j \qquad f\ x_1 \ldots x_n : T \vdash F}{f : (x_1 : T_1) \to \ldots (x_n : T_n) \to T\ \vdash\ \text{pi}\ x_1 \ldots x_n \backslash\ F \;{:}{-} F_1, \ldots F_m}$$

- $free(x)$ - x is not used anywhere in the type

Example - simple

GF

$$\textbf{data}\ zero : Nat;$$
$$succ : Nat \rightarrow Nat;$$

Lambda Prolog

$$Nat_p\ zero.$$
$$\text{pi}\ X\backslash\ Nat_p\ (succ\ X) :\!-\ Nat_p\ X.$$

Example - high-order functions

GF

$$\textbf{data } f : (Nat \rightarrow Nat) \rightarrow Nat;$$

Lambda Prolog

$$\text{pi } G \backslash\ Nat_p\ (f\ \ G)\ {:}{-}\ (\text{pi } X \backslash\ Nat_p\ (G\ \ X)\ {:}{-} Nat_p\ X).$$

Note: quantification over function i.e. G

Example - dependent types

GF

$$
\begin{aligned}
\textbf{cat}\ & Vec\ Nat; \\
\textbf{data}\ & nil : Vec\ zero; \\
& cons : (\{n\} : Nat) \rightarrow Nat \rightarrow Vec\ n \rightarrow Vec\ (succ\ n);
\end{aligned}
$$

Lambda Prolog

$$
\begin{aligned}
& Vec_p\ nil\ zero. \\
\text{pi}\ X, L, N \backslash\ & Vec_p\ (cons\ N\ X\ L)\ (succ\ N) \mathrel{:-} Nat_p\ X, Vec_p\ L\ N.
\end{aligned}
$$

Note: no $Nat_p\ N$ because N is output variable in $Vec_p\ L\ N$

Translation of Functions to Predicates - by example

GF

$$\textbf{fun}\ plus : Nat \rightarrow Nat \rightarrow Nat;$$

$$\textbf{def}\ plus\ zero\ n = n;$$

$$plus\ (succ\ m)\ n = succ\ (plus\ m\ n);$$

Lambda Prolog

$$\textbf{exportdef}\ plus\ Nat_t \rightarrow Nat_t \rightarrow Nat_t \rightarrow o.$$

$$plus\ zero\ X\ X.$$

$$plus\ (succ\ X)\ Y\ (succ\ Z) :\!-\ plus\ X\ Y\ Z.$$

Problem: functions should be computed lazily

The encoding of functions as predicates could model only strict functions, but:

- SICStus Prolog has extensions that could emulate lazyness
- The proof search is lazy by default in Curry

Two places to look for ...

Uncomputing

Let's say that we have:

$$\textbf{fun } append : (m, n : Nat) \to Vec\ m \to Vec\ n \to Vec\ (plus\ m\ n);$$

Now try to prove:

$$Vec_p\ (succ\ (succ\ zero))$$

Obviously for *append* you have to compute *plus* backwards i.e. find $m, n : Nat$ such that $m + n = 2$

Note: we will use this to solve NQueens

Type Checking as Prolog Program

We have two type checkers one in the compiler and one in the interpreter.

The runtime type checker is actually running a Prolog program. Example:

?– Vec_p *nil zero*.
yes

?– Vec_p *nil* (*cons zero*).
no

This doesn't scale with meta-variables

Prolog uses narrowing:

$$
\begin{array}{l}
?\!-\ \mathit{Vec}_p\ (\mathit{cons}\ X\ \mathit{nil})\ (\mathit{succ}\ \mathit{zero}). \\
\mathrm{X} = \mathit{zero} \\
\text{yes} \\
\mathrm{X} = \mathit{succ}\ \ \mathit{zero} \\
\text{yes} \\
\ldots
\end{array}
$$

The typecheckers in Agda and GF need residuation. We must borrow the residuation strategy from Curry:

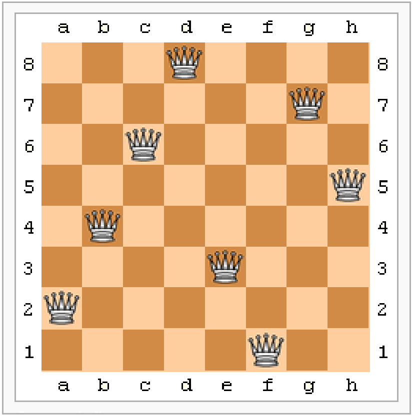
$$
\begin{array}{l}
?\!-\ \mathit{Vec}_p\ (\mathit{cons}\ X\ \mathit{nil})\ (\mathit{succ}\ \mathit{zero}). \\
\text{yes}
\end{array}
$$

Proof of Concept

I have implemented source-to-source transformation from GF to Lambda Prolog

The final goal is to integrate the virtual machine of Lambda Prolog directly in GF

Demo: N-Queens solver



The n-queens puzzle is the problem of placing n chess queens on a $n \times n$ chessboard such that none of them are able to capture any other using the standard chess queen's moves.

The Chessboard Reduced to Vector

$$\textbf{cat}\ Matrix\ Nat;$$
$$Vec\ (s, l : Nat)\ [Nat];$$

$$\textbf{data}\ matrix : (s : Nat) \longrightarrow Vec\ s\ s\ BaseNat \longrightarrow Matrix\ s;$$

- s - the size of the chessboard
- l - the length of the vector
- $[Nat]$ - the list of already occupied positions

Need Type-Level Inequality of *Nat*

$$\textbf{cat}\ NE\ (i, j : Nat);$$

$$\begin{aligned}\textbf{data}\ zNE &: (i, j : Nat) \rightarrow NE\ i\ j \rightarrow NE\ (succ\ i)\ (succ\ j);\\ lNE &: (j : Nat) \rightarrow NE\ zero\ (succ\ j);\\ rNE &: (j : Nat) \rightarrow NE\ (succ\ j)\ zero;\end{aligned}$$

- *zNE* - induction step
- *lNE*, *rNE* - base cases

Satisfiability Condition

$$\textbf{cat } Sat\ Nat\ Nat\ [Nat];$$

$$\begin{aligned}
\textbf{data } nilS &: (j, d : Nat) \to Sat\ j\ d\ BaseNat; \\
consS &: (i, j, d : Nat) \to (c : [Nat]) \\
&\quad \to NE\ i\ j \\
&\quad \to NE\ i\ (plus\ d\ j) \\
&\quad \to NE\ (plus\ d\ i)\ j \\
&\quad \to Sat\ j\ (succ\ d)\ c \\
&\quad \to Sat\ j\ d\ (ConsNat\ i\ c);
\end{aligned}$$

- j - the position that we check
- i - the occupied position d lines above the current line

The Vector

$$\textbf{data}\ nilV : (s : Nat) \rightarrow (c : [Nat]) \rightarrow Vec\ s\ zero\ c;$$

$$\begin{aligned} consV &: (l, j, k : Nat) \rightarrow \\ &\quad \textbf{let}\ s = succ\ (plus\ j\ k) \\ &\quad \textbf{in}\ (c : [Nat]) \rightarrow Sat\ j\ (succ\ zero)\ c \rightarrow \\ &\qquad Vec\ s\ l\ (ConsNat\ j\ c) \rightarrow Vec\ s\ (succ\ l)\ c; \end{aligned}$$

- $j, k : Nat$, such that $j + 1 + k = s$

Concrete Syntax for Vector and Matrix

$$\textbf{lincat}\ \mathit{Matrix}, \mathit{Vec} = \mathit{Str};$$
$$[\mathit{Nat}], \mathit{Sat} = \{\};$$

$$\textbf{lin}\ \mathit{nilV}\ \_\ \_ = "";$$
$$\mathit{consV}\ \_\ j\ k\ \_\ \_\ v = j +\!\!+ \text{"X"} +\!\!+ k +\!\!+ \text{"\textbackslash n"} +\!\!+ v;$$

$$\mathit{matrix}\ \_\ v = v;$$

Compilation via Lambda Prolog

Generate Code:

```
$ gf -make -output-format=lambda_prolog examples/nqueens/NQueensAscii.gf
Writing NQueens.pgf...
Writing NQueens.mod...
Writing NQueens.sig...
```

Compile:

```
$ tjcc NQueens.mod
$ tjlink NQueens.lpo
$ tjsim NQueens.lp
```

Run:

?– *p_Matrix* (*succ* (*succ* (*succ* (*succ* *zero*))))

Linearize the result in GF:

l -unchars "the tree generated from Lambda Prolog"

Conclusion

The virtual machine of Lambda Prolog offers almost everything that we need:

- efficient backtracking
- high-order pattern matching unification
- hereditary Harrop formulae

but we need also:

- laziness
- residuation mode