Modes for Non Strict Functional Logical Languages

Matthew Mirman

April 17, 2012

Abstract

Functional logical programming is a paradigm that introduces predicate satisfaction as a first class construct into the functional setting. In consolidating logical and functional semantics, a complete logical query primitive is desirable. In the non strict, breadth first evaluation strategy, it is possible to return unground logical variables. This makes useless the notion of the scope of logical variable declaration. Given the small step operational semantics for a practical lazy functional logical language, I approximate them with natural semantics for a lambda calculus with lazy logical primitives for which I supply a type and mode system. Types are annotated with modes denoting their intended groundedness attributes. Furthermore, given that the analysis is to be performed on functional logical code, the type system allows annotation of higher order function types with modes, and thus supports grounding functions. Modes are observed as forming a natural ordering, and thus type and mode checking supports limited subtyping. I prove in Twelf the soundness of the annotations produced from type checking, and type preservation. A proof of progress depending on the existence of the resolving property of positively moded terms is discussed. It is also noted that positively moded corresponds with the notion of strict, and that the mode analysis could be considered a specialized strictness analysis.

While mode analysis prevents logical variables from leaving the scope of the search that introduced them, it does not prevent them from being declared outside of the scope of a search. Such results are meaningless in the context of the determinism necessary for meaningful I/O. A method of ensuring that logical variables are not introduced outside of the scope of search is also introduced by the introduction of a second arrow constructor denoting nondeterministic functions.

Introduction

Logic programming is a paradigm where rather than describing the action the computer should take to achieve a result, properties of the desired result are listed. Pure logic programming involves listing predicates over simple data

types. Logic programming is a powerful model for describing complex queries, and encourages programmers to very often write less verbose code. Logic programming alone is not always ideal. While logic programming is very good for describing a single static data input and output, it alone is not adept at describing dynamic interactions. Because logic programming describes so much computation in so little code by way of automation, the operational meaning of code is often obfuscated if not entirely unspecified. In cases where the action of the computer is to be specified, functional code is ideal. While it is simple to write a language where functional code can call logical code written separately, mixing the two to allow functional code to be accessed within logical code would be ideal. The ability to write mostly legible logical code and optimize a few cases would allow the programmer to choose what sacrifices to make, and allow for more interactive queries.

In this thesis, I provide static analysis for a lambda calculus with additional logical primitives, in order to inform practical functional logical languages. The language I analyze here is an extension of the lambda calculus which includes the logic primitives **free, findAll**, and **caseof**. **free** is a function which initializes a variable as a free parameter in it's scope. **findAll** searches for any or all instances of a variable which satisfy a predicate. **caseof** non-deterministically branches when given a logic variable as an argument.

Nondeterministically branching for every free variable at every switch statement could very easily result in an unnecessarily slow programs, and thus a non strict evaluation strategy is desired. The Needed Narrowing evaluation strategy optimal in this respect. It proceeds by lazily reducing a term, and only initializing logical variables if they are used as the argument to a **caseof** statement who's results are required to continue execution. Beyond simply being more efficient, Needed Narrowing is complete [4] in the same sense as non strict program evaluation. A reduction strategy for lambda calculus is complete if every expression for which there exists a reduction resulting in a value reduces under that strategy to a value. Similarly, a reduction strategy for lambda calculus is complete if it has the previous property, and also has the property that for all values such that predicate passed to **findAll** has a reduction to success when applied to that value, then that value will appear somewhere in the result of the **findAll**.

Example. To illustrate a case where functional logical programming would be convenient, consider the following Haskell code:

```
step (App (Lam f) e2) res =
    res == f e2
step (App e1 e2) (App e1' e2') =
    (e1' == e1 `and` step e2 e2')
 `or` (e2' == e2 `and` step e1 e1')
```

This code defines a predicate that ensures it's second lambda term is a single step nondeterministic reduction of the first. It is far simpler and more usefull than the code that performs that single step nondeterministic reduction. Just given

this predicate, we can not easily construct a function which lists all possible single step reductions of a term. A naive attempt might list all possible terms, and use this predicate as a filter.

```
reductions term = filter (step term) listAllTerms
```

In a functional logical language an efficient function to output the same set of items can be defined far more easily:

```
reductions term = findAll red in step term red
```

Motivation

While functional logical programming has become more present with the formulation of small step semantics, it appears as though less attention has been made to the static analysis and verification of such languages. There do exist non strict functional logical programming language implementations[5], but no practical language appears to statically verify runtime safety entirely. If the language is intended to be efficiently compiled to safe code, statically ensuring safety is essential. Furthermore, functional logical programming presents an incredible paradigm shift from traditional imperative or functional paradigms. It is thus necessary for the compiler to provide as much feedback as possible to inform the programmer of the correct use of the language, without hindering the readability of code. As reasoning about the time complexity of logical code is undesirable and difficult in the intended setting, the non strict and complete evaluation strategy known as Needed Narrowing is used. In consolidating logical and functional semantics, the primitive **findAll** is both an accessible and necessary means to make use of successful queries. I note that in conjunction with a non strict evaluation strategy, it is possible for **findAll** to return unground logical variables, despite being the only block against nondeterminism. Mode analysis**(author?)** [13] is suggested as a means of ensuring results from findAll are ground. As logical code can be considered a non-deterministic search, the primitive **findAll** can be easily parallelized. However, the unconstrained use of free variables throughout code could easily result in space leaks and unintentional non-deterministic IO. I present a type level system for constraining the scope of non determinism in plausible programs.

In a non strict evaluation strategy, logic variables are never initialized until they are needed to make progress. It is important that if there are values that when input to a predicate result in success, that the primitive **findAll** will output those values rather than diverging before outputting them. This completeness property of findAll can be ensured by a breadth first search of possible variable initializations and evaluations. While both depth first and breadth first searches are possible in Curry and Prolog in the absence of a **findAll** primitive, I show that the introduction of **findAll** as a language primitive make controlling non-determinism in the I/O monad possible. The introduction of **findAll** as a complete primitive in the presence of non strict semantics makes mode checking a necessity.

Example. Consider the following Hypothetical Korma code

```
list = findAll $ \a -> free $ \z -> case z of
    A -> left a =:= A
    B -> (right a,left a) =:= ([A],A)
```

Given the needed narrowing evaluation strategy, list would contain two values, "{left = A & right = ? }" and "{ left = A & right = [A] }". We can thus infer that the type of list is "[(A+B) * [A + B]]". However, if we were to ask for "right (head list)", we would have encountered a logical variable. This code is thus non resolving. The effects from non resolving code can be non local and unintuitive for a novice logic programmer, and thus preventing findAll from accepting non resolving functions is necessary.

Language Definition

It is first necessary to provide a language as a target for the analysis. To simplify, unification, recursive types, and polymorphism are omitted, although it is likely that they will not pose much of a problem when reconsidered.

The grammar is as follows:

$$m \ ::= \oplus \mid \ominus$$

$$t \ ::= t \to t \mid t \twoheadrightarrow t \mid \emptyset \mid \text{answer} \mid t \times t \mid t +_m t$$

$$e \ ::= x \mid \lambda x : t.e \mid \lambda v.e \mid (e \ e) \mid$$

$$\text{findAll}_t \mid \text{free}_t \mid \text{success} \mid \text{fail} \mid$$

$$\text{obj} \mid \text{getLeft} \mid \text{getRight} \mid$$

$$\text{left} \mid \text{right} \mid \text{caseof} \mid \text{unit} \mid \text{unresolved}$$

For purposes of the proof, the term **unresolved** is also included.

Semantics

The semantics for this language we attempt to analyze are an approximation of the small step Needed Narrowing semantics explained in [**?**]. The semantics of the language are defined as follows:

Definition 1. $E \Rightarrow E'$ is the lazy single step relation defined as follows:

$$\text{(E-AP}P_1\text{)}\frac{E_1 \Rightarrow E_1'}{E_1\ E_2 \Rightarrow E_1'\ E_2}$$

$$\text{(E-APP LAM)}(\lambda x : t.e_1)\ e_2 \Rightarrow [x \mapsto e_2]e_1 \text{provided t is free for } e_2 \text{ in } e_1$$

$$\text{(E-GET LEFT) getLeft (obj } e_1\ e_2) \Rightarrow e_1$$

$$\text{(E-GET RIGHT) getRight (obj } e_1\ e_2) \Rightarrow e_2$$

$$\text{(E-SWITCH-LEFT) (caseof } LF\ RF)\ (\text{left}L) \Rightarrow LF\ L$$

$$\text{(E-SWITCH-RIGHT) (caseof } LF\ RF)\ (\text{right } L) \Rightarrow RF\ R$$

$$\text{(E-FINDALL-SUCC)}\ \frac{\vdash V : T \quad (E\ V) \Rightarrow \text{success}}{\text{findAll}_T\ E \Rightarrow (\text{left } V)}$$

$$\text{(E-FINDALL FAIL)}\ \frac{V : T \vdash (E\ V) \Rightarrow \text{fail}}{\text{findAll}_T\ E \Rightarrow (\text{right } ())}$$

$$\text{(E-FREE-SUCC)}\ \frac{\vdash V : T}{\text{free}_T\ E \Rightarrow E\ V}$$

Type and mode Checking

In order to discuss the type checker, it is necessary to first describe the relationship between types and modes. The mode $\oplus$ describes values which do not need to become ground, and the mode $\ominus$ describes values which must become ground. We can use a value which does not need to become ground anywhere we know we will ground the value, but we should not us a value which must become ground anywhere we do not know we will ground it (some programs which violate this rule will be correct, but this rule makes life easier). Thus, $\oplus \leq \ominus$. As usual, $\sqcup$ will describe the least upper bound.

We also need to define what it means for a type to specify a mode.

Definition 2. $t \sim m$ shall mean that the type t has mode m.

$$\text{(TP-MODE/UNIT)}\ \emptyset \sim \oplus$$

$$\text{(TP-MODE/SUM)}\ t_1 +_m t_2 \sim m$$

$$\text{(TP-MODE/PROD)}\ \frac{t_1 \sim m_1 \quad t_2 \sim m_2}{t_1 \times t_2 \sim m_1 \sqcup m_2}$$

$$\text{(TP-MODE/ARROW)}\ t_1 \to t_2 \sim \oplus$$

Not all types are necessarily well moded. In order for a type to be well moded, sums must have a mode that is an upper bound of the modes of it's constituent types.

Definition 3. $t \approx m$ shall mean that the type t is well moded with mode m.

$$\text{(TP-MODE-SAFE/UNIT)}\ \emptyset \approx \oplus$$

$$\text{(TP-MODE-SAFE/SUM)}\ \frac{t_1 \approx m_1 \quad t_2 \approx m_2 \quad m_1 \leq m \quad m_2 \leq m}{t_1 +_m t_2 \approx m}$$

$$\text{(TP-MODE-SAFE/PROD)}\ \frac{t_1 \approx m_1 \quad t_2 \approx m_2}{t_1 \times t_2 \approx m_1 \sqcup m_2}$$

$$\text{(TP-MODE-SAFE/ARROW)}\ \frac{t_1 \approx m_1 \quad t_2 \approx m_2}{t_1 \to t_2 \approx \oplus}$$

Lemma 4. *for all t there exists some m such that $t \sim m$. Furthermore, m is unique.*

Proof. Formalized in Twelf. □

Lemma 5. *$t \approx m$ implies that $t \sim m$.*

Proof. Formalized in Twelf □

Given that modes are an annotation on the type system, mode checking and type checking are done together. In order to keep type checking minimal, we provide it as a lemma that the types produced from type checking are mode safe. The rules for type checking are as follows

Definition 6. $E : T$ means that expression E has type T and is defined as follows. It depends on the predicate usedCorrectly(T_1, T_2, e) which measures given that e is a function, whether it's argument is used correctly.

$$\text{(OF-ASSUM)}\frac{}{\Gamma, x : T \vdash x : T}$$

$$\text{(OF-SUBSUMP)}\frac{\Gamma \vdash e : T \quad T \leq T' \quad T' \approx m}{\Gamma \vdash e : T'}$$

$$\text{(OF-LAM)}\frac{\Gamma, x : T_1 \vdash e[x] : T_2 \quad \text{usedCorrectly}(T_1, T_2, e) \quad T_1 \approx m_1 \quad T_2 \approx m_2}{\Gamma \vdash (\lambda x : T_1 . e[x]) : T_1 \to T_2}$$

$$\text{(OF-APP)}\frac{\Gamma \vdash e_1 : T_1 \to T_2 \; \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2}$$

$$\text{(OF-LOGIC-VAR)}\frac{T \approx \ominus}{\Gamma \vdash \text{unresolved} : T}$$

$$\text{(OF-getLeft)}\frac{v_1 \times v_2 \approx m}{\Gamma \vdash \text{getLeft} : \; v_1 \times v_2 \to v_1}$$

$$\text{(OF-getRight)}\frac{v_1 \times v_2 \approx m}{\Gamma \vdash \text{getRight} : \; v_1 \times v_2 \to v_2}$$

$$\text{(OF-LEFT)}\frac{v_1 +_m v_2 \approx m}{\Gamma \vdash \text{left} : v_1 \to v_1 +_m v_2}$$

$$\text{(OF-RIGHT)}\frac{v_1 +_m v_2 \approx m}{\Gamma \vdash \text{right} : v_2 \to v_1 +_m v_2}$$

$$\text{right} : \; v_2 \to v_1 + v_2$$

$$\text{(OF-OBJ)}\frac{v_1 \times v_2 \approx m}{\Gamma \vdash \text{obj} : v_1 \to v_2 \to v_1 \times v_2}$$

$$\text{unit} : \emptyset$$

$$\text{(OF-CASE-OF)}\frac{v_1 \approx m \quad v_2 \approx m}{\text{caseof} : (v_1 \to v_3) \to (v_2 \to v_3) \to v_1 +_m v_2 \to v_3}$$

We defined *usedCorrectly* in the associated Twelf file (TypeChecking.elf). For the moment, it's full definition here is omitted for brevity. Essentially, *usedCorrectly* ensures that in the current environment, if it is given a supposedly negatively moded argument, it is used somewhere as a negatively moded argument. That argument would have to be used in either both sided of a switch statement, have both it's left and right constituents used negatively if it is an object, or if we are checking if it is used inside of an application, that it is either used on the left hand side, or it is used negatively on the right hand side, and the left hand side is strict.

As the heavy analysis we will make does not involve **free, findAll**, **success** or **fail**, we will give their types in System F.

$$\text{free} : \forall v_1 \; v_2 . (v_1 \to v_2) \to v_2$$

$$\text{findAll} : \forall v . (v \to \text{answer}) \to v + \emptyset$$

$$\text{success} : \mathit{answer}$$

$$\text{fail} : \mathit{answer}$$

Proposition 7. *$E : T$ implies that $T \approx m$. Furthermore, m is unique.*

Lemma 8. *If there is a decidable algorithm to infer these types, and one to perform the mode checking analysis on the annotated system, then inferring their modes is decidable.*

Proof. For every sum type inferred, annotate it with either a positive or negative mode. Perform the annotated type checking. Because there are finite sum nodes in the inferred types, there are finite possible mode assignments. □

Preservation

Because of submoding, the proof of preservation is slightly more complex than in simply typed lambda calculus.

Definition 9. $T \leq T'$ simply means that $T$and T' have the same structure but different modes. At each sum, $m \leq m'$.

$$\frac{T_1 \leq T_1' \quad T_2 \leq T_2'}{T_1 \times T_2 \leq T_1' \times T_2'}$$

$$\frac{T_1 \leq T_1' \quad T_2 \leq T_2' \quad m \leq m'}{T_1 +_m T_2 \leq T_1' +_{m'} T_2'}$$

$$\frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \to T_2 \leq T_1' \to T_2'}$$

Lemma 10. *$T \leq T$ is admissible.*

Proof. Formalized in Twelf. □

Lemma 11. *$T \leq T'$ and $T \sim m$ and $T' \sim m'$ implies $m \leq m'$.*

Proof. Formalized in Twelf. □

Theorem 12. *Preservation*
$E \Rightarrow E'$ and $\vdash E : T$ then $\vdash E' : T'$ and $T' \leq T$.

Proof. Formalized in Twelf. □

Progress

Before we can prove progress for the full language, we need to prove that the sublanguage not considering findAll has the property that it can not reduce away logical variables.

Definition 13. logicFree(E) shall mean that the term E contains no logical variables in cases where it matters, for example when E is a value, or when E is not an application of a not necessarily grounding function to any argument. Again, the full definition is omitted here for brevity, but included in the Twelf file Global.elf.

Conjecture 14. *Subterm-Resolving*

$\Gamma \vdash X : T_x$ *and* $usedAsNeg(X, E)$ *and* $\Gamma \vdash E : T_E$ *and* $T_E \approx \oplus$ *and* $E \Rightarrow_k V$ *with* $\Gamma \vdash V : T_V$ *and* $T_V \leq T_E$ *by preservation and logicFree(V) then* $logicFree(X)$

Proof. Note that this has been mostly proved in Twelf with all but a few cases left to handle. The intuition behind this theorem is that $usedAsNeg$ measures strict occurrences of X in E, and by definition, an occurrence can only be strict if it will be used at least once in the computation required to reduce E completely. □

Corollary 15. *Term-Resolving*

Provided Subterm-Resolving holds in general, $\Gamma \vdash X : T_X$ *and* $T_X \approx \oplus$ *and* $X \Rightarrow_k V$ *with* $\Gamma \vdash V : T_V$ *and* $T_V \leq T_X$ *by preservation and logicFree(V) then* $logicFree(X)$

Proof. Given that subterm resolving holds, we can simply apply subterm resolving with X for E and usedAsNeg/e for usedAsNeg(X, X). □

Theorem 16. *Progress Holds*

Provided Term-Resolving holds in general, $E : T$ *and* $T \approx \oplus$ *implies either* E *is a value or* $E \Rightarrow E'$ *for some* E'*. Furthermore, if* E *does not contain logical variables, then* E' *will also contain no logical variables.*

Proof. Without the **findAll** or **free** primitives or logical variables, the language given is just the simply typed lambda calculus, and the progress theorem follows nearly trivially. The introduction of logical variables, **findAll** makes the analysis a bit more complex. In the case of a **findAll**, we show that it must only accept grounding arguments. This implies that if we apply a value to the argument and it evaluates to success, it could not originally have had any logical variables by the Term-Resolving corollary. Thus, no results from **findAll** will contain logical variables. **Free** is not defined as containing no logical variables, and no other primitives introduce logical variables. □

Strictness Analysis

Definition 17. A function E is strict if $E \circ hnf \equiv E$, where hnf is the function that evaluates it's arguments to head normal form.

Proposition 18. E *is strict iff* E *is grounding.*

Proof. Suppose E is grounding, then upon completion of E, every aspect of it's argument must be used, thus evaluating that argument to head normal form will not expose any divergent computations not already exposed.

Suppose E is strict. then suppose we were to pass an argument to E with logical variables. Substitute every logical variable for a divergent computation. Then because E is strict, evaluating this argument to head normal form first will not change the result of E when passed this argument. However, evaluating the

argument to head normal form first causes the computation to diverge. Thus, passing the argument to E will cause the computation to diverge. Thus the original argument with logical variables passed to Ewill either diverge converge to a value with logical variables. Thus $E$is grounding. □

Corollary 19. *ClosedE is strict if* $\vdash E : I \to O$ *and* $I \sim \ominus$ *and* $O \sim \oplus$.

Ramifications

Because mode checking and strictness checking are equivalent, mode checking algorithms can be used for strictness analysis. It is unlikely however, that strictness analysis would be as applicable to mode checking, as they need not be decidable nor sound. Mode checking is performed when well modedness is a requirement and not simply a benefit, and thus deciding when code is improperly moded is as important as deciding when it is properly moded.

The Scope of Non-determinism

To show that the **findAll** primitive can be used to control the scope of free variables, it is easiest to use notions from subtyping systems, and to construct a syntactic transformation. I also present a method for ensuring the unifiability of the free variables. In order to unify, free variables must be unifiable, and constructed from products and sums only.

We can refine the types stated earlier for as follows, given the presence of type classes:

$$\text{free} :: \forall a\ b.(\text{Unifiable}\ a) \implies (a \to b) \to b$$

$$\text{findAll} :: \text{Unifiable}\ a \implies (a \to \text{Success}) \to [a]$$

Note here that **findAll** returns a list. The typeclass **Unifiable** can be automatically derived for types which do not contain arrows.

In a pure lazy functional setting, IO has traditionally been accomplished using the IO monad. In order to not deviate from this pattern, it is necessary to ensure that computations still make sense from within the IO monad, and that the scope of the free variables does cause unexpected nondeterminism of effects. Ensuring that the nondeterminism of free does leek into the IO monad statically can be made into a type level problem by the following system.

Suppose we have two parameterized types Many a and Single a, with the following subtyping rules:

$$\text{(S-ReturnQuantity)}\ \text{Single}\ a <:\ \text{Many}\ a$$

Single and Many are instances of the built in ReturnQuantity typeclass, and the following application function's type is given as a primitive.

$$\$\$ :: \text{ReturnQuantity } m \implies (a \to m\ b) \to (m\ a \to m\ b)$$

The "main" action for the program will has type Single (IO ()), and return :: $a \to$ Single $(m\ a)$

$$\text{free} :: (\text{Unifiable } a) \implies (a \to \text{Many } b) \to \text{Many } b$$

$$\text{findall} :: (\text{Unifiable } a) \implies (a \to \text{Many Success}) \to \text{Single } [a]$$

This way, any function which is deterministic can be used anywhere a nondeterministic function can be used, but not vise versa. To make this feature not cumbersome to the user, we apply the inductively defined transformation ϕ to the abstract syntax tree.

$$\phi(e_1\ e_2) = \phi(e_1)\ \$\$\ \phi(e_2)$$

$$\phi(\lambda x.e) = \lambda x.\phi(e)$$

References

[1] Samson Abramsky and Thomas P. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In *In Proc. ACM Symposium on Principles of Programming Languages*, pages 49–54. ACM Press, 1991.

[2] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and German Vidal. A deterministic operational semantics for functional logic programs. *Joint Conf. on Declarative Programming*, pages 207–222, 2002.

[3] Penny Anderson and Frank Pfenning. Verifying uniqueness in a logical framework. *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, pages 18–33, September 2004.

[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *ACM Symposium on Principles of Programming Languages*, pages 268–279, 1994.

[5] S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(April):74–85, 2010.

[6] Erik Barendsen and Sjaak Smetsers. Strictness analysis via resource typing. In *Reflections on Type Theory, Lambda Calculus and the Mind*, pages 29–40, December 2007.

[7] B. Braßel and M. Hanus. Nondeterminism analysis of functional logic programs. In *Proceedings of the International Conference on Logic Programming (ICLP'05)*, pages 265–279. Springer LNCS 3668, 2005.

[8] Tim Freeman and Frank Pfenning. Refinement types for ml. In *SIGPLAN Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[9] Ed. Hanus, M. Curry: An integrated functional logic language.

[10] Michael Hanus and Frank Steiner. Type-based nondeterminism checking in functional logic programs. In *In Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000*, pages 202–213. ACM Press, 2000.

[11] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[13] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. *Proceedings of the European Symposium on Programming*, pages 296–310, April 1996.

[14] Kirsten Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In *International Static Analysis Symposium, LNCS 983*, pages 408–422, 1994.