# Modes for Non Strict Functional Logic Languages

Matthew Mirman (mmirman@andrew.cmu.edu)
Advisor: Frank Pfenning (fp@cs.cmu.edu)

May 18, 2012

**Abstract**

Functional logic programming is a paradigm that introduces proof search as a first class construct into the functional setting. In consolidating logic and functional semantics, a complete logic query primitive is desirable. In the reasonable and efficient evaluation strategy, it is possible to determine the correctness of a statement without entirely determining the proof. This would allow searches to leave their intended scope. Given the small step operational semantics for a practical lazy functional logic language, I approximate them with natural semantics for a lambda calculus with lazy logic 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 logic code, the type system allows annotation of higher order function types with modes. 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.

## 1 Introduction

Logic programming is a paradigm for describing queries by constraining output rather than necessarily transforming input. It often encourages programmers to write less verbose and more declarative code. Pure logic programming involves listing values satisfying predicates over simple data types. However, logic programming alone is not always the correct tool of choice. 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 logic code written separately, mixing the two to allow functional code to be accessed within logic code would logic.

In this thesis, I provide static analysis for a lambda calculus with additional logic primitives, in order to inform practical functional logic languages. The language I analyze here is an extension of the lambda calculus which includes the logic primitives **free, findAll**,

**Example 1.** Consider the following haskell code for checking that a lambda term is a reduction of another.

```
data Exp = Lam (Exp -> Exp) | App Exp Exp
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 its second lambda term is a single step nondeterministic reduction of the first. It is simpler than the code that performs a single step nondeterministic reduction. Just given this predicate, we cannot easily construct a function that 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 logic language, a more efficient function to output the same set of items can be defined far more easily:

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

Figure 1.1: A case where functional logic programming would be convenient.

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; thus a non strict evaluation strategy is desired. The *Needed Narrowing*[4] evaluation strategy optimal in this respect. It iterates by lazily reducing a term, and only initializing logic variables if they are used as the argument to a **caseof** statement whose 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**.

## 1.1 Motivation

While functional logic programming has become more present with the formulation of small step semantics, it appears as though less attention has been paid to the static analysis and verification of such languages. There do exist non strict functional logic 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. As reasoning about the time complexity of logic code is undesirable and difficult in the intended setting, the non strict and complete evaluation strategy

known as needed narrowing is used. I note that in conjunction with a non strict evaluation strategy, it is possible for **findAll** to return unground logic variables, despite it being the only block against nondeterminism. Mode analysis[14] might be used as a means of ensuring results from **findAll** are ground. As logic code can be considered a non-deterministic search, the primitive **findAll** is highly concurrent. However, the unconstrained use of free variables throughout code could easily result in space leaks and unintentional non-determinism. 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 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, "( A , ? )" and "(A,[A])". We can thus infer that the type of list is $[(A+B)\times[A+B]]$ . However, if we were to ask for "right (head list)", we would have encountered a logic variable. We say that this predicate is not grounding. The effects from non resolving code can be non local and non-intuitive for a novice logic programmer, and thus preventing findAll from accepting non resolving functions is necessary.

# 2 Language Definition

It is first necessary to provide a language as a target for the analysis. To simplify recursive types, and polymorphism are omitted, although it is likely that they will not pose significant complications in the future if restricted in the usual fashions. For purposes of the proof, the term **unresolved** is also included, which is used as a logic variable which can have any negatively moded type.

## 2.1 Syntax

The syntax for the language is defined as follows.

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

**Types** $t \ ::= t \to t \mid t \twoheadrightarrow t \mid 1 \mid \text{answer} \mid t \times t \mid t +_m t$

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

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

$$(e, e) \mid \pi_1 \mid \pi_2 \mid$$

$$\text{inl} \mid \text{inr} \mid \text{case } e \text{ of } (e_1, e_2) \mid \text{unit} \mid \text{unresolved}$$

## 2.2 Semantics

Language semantics are a crucial property to specify before meaningful analysis can take place. logic languages when given naive semantics can be extraordinarily inefficient. Needed narrowing is an evaluation strategy where variables are only instantiated when continuing the lazy evaluation of a predicate requires their instantiation, such as at a branching construct. The small step needed narrowing semantics [2] as shown below have been known in practice to give good results.

Table 1: Small step operational semantics [2]

| Rule | Heap | Control | Stack | Notes |
|---|---|---|---|---|
| ST-VARC | $\Gamma[x \mapsto t]$ | $x$ | $S$ | |
| | $\Gamma[x \mapsto t]$ | $t$ | $S$ | |
| ST-VARE | $\Gamma[x \mapsto e]$ | $x$ | $S$ | e is not constructor rooted |
| | $\Gamma[x \mapsto t]$ | e | $x : S$ | $e \neq x$ |
| ST-VAL | $\Gamma$ | $v$ | $x : S$ | $v$ is constructor rooted or a |
| | $\Gamma[x \mapsto v]$ | $v$ | $S$ | variable with $\Gamma[y] = y$ |
| ST-LAM | $\Gamma$ | $(\lambda x : t.e)$ | $e' : S$ | $y$ is fresh |
| | $\Gamma[y \mapsto e']$ | $[x \mapsto y]e$ | $S$ | |
| ST-APP | $\Gamma$ | $e_1\ e_2$ | $S$ | |
| | $\Gamma$ | $e_1$ | $e_2 : S$ | |
| ST-C | $\Gamma$ | case $e$ of $LF$ $RF$ | S | |
| | $\Gamma$ | $e$ | c$LF$ $RF : S$ | |
| ST-C-L | $\Gamma$ | inl $e$ | c$LF$ $RF : S$ | |
| | $\Gamma$ | $LF\ e$ | $S$ | |
| ST-C-R | $\Gamma$ | inr $e$ | c$LF$ $RF : S$ | |
| | $\Gamma$ | $RF\ e$ | $S$ | |
| ST-G-L | $\Gamma[x \mapsto x]$ | $x$ | c$LF$ $RF : S$ | $y$ is fresh |
| | $\Gamma[x \mapsto \text{inl } y,\ y \mapsto y]$ | $LF\ y$ | $S$ | |
| ST-G-R | $\Gamma[x \mapsto x]$ | $x$ | c$LF$ $RF : S$ | $y$ is fresh |
| | $\Gamma[x \mapsto \text{inr } y,\ y \mapsto y]$ | $RF\ y$ | $S$ | |
| ST-FREE | $\Gamma$ | free | $(\lambda x.e) : S$ | $y$ is fresh |
| | $\Gamma[y \mapsto y]$ | $[x \mapsto y]e$ | $S$ | |

The semantics used in our analysis is a reduction semantics which approximates the small step semantics above. While the big step semantics do not precisely model the small step semantics, the approximation allows for the simplification of the lambda calculus rules, while maintaining the possibility of **findAll** returning logic variables. The **findAll** in the big step semantics rather than lazily constructing a list, nondeterministically returns any value which causes its predicate to succeed.

**Definition 2.** $E \Rightarrow E'$ is the non strict big step reduction defined as follows.

$$\text{(E-ID-LAM)}(\lambda x : t.F) \Rightarrow (\lambda x : t.F)$$

$$\text{(E-ID-OBJ)}(A, B) \Rightarrow (A, B)$$

$$\text{(E-ID-UNIT)unit} \Rightarrow \text{unit}$$

$$\text{(E-ID-SUCCESS)success} \Rightarrow \text{success}$$

$$\text{(E-ID-FAIL)fail} \Rightarrow \text{fail}$$

$$\text{(E-ID-INL)}\mathbf{inl}\ A \Rightarrow \mathbf{inl}\ A$$

$$\text{(E-ID-INR)}\mathbf{inr}\ A \Rightarrow \mathbf{inr}\ A$$

$$\text{(E-APP-LAM)}\frac{E_1 \Rightarrow (\lambda x : t.F) \quad [x \mapsto E_2]F \Rightarrow V}{E_1\ \ E_2 \Rightarrow V}\text{provided x is free for } E_2 \text{ in } E_1$$

$$\text{(E-GET)}\ \frac{E \Rightarrow (E_1, E_2)\ \ E_i \Rightarrow V}{\pi_i E \Rightarrow V}$$

$$\text{(E-SWITCH-LEFT)}\ \frac{E \Rightarrow \mathbf{inl}\ L \quad LF\ L \Rightarrow V}{\text{case } E \text{ of } (LF, RF) \Rightarrow V}$$

$$\text{(E-SWITCH-RIGHT)}\ \frac{E \Rightarrow \mathbf{inr}\ R \quad RF\ R \Rightarrow V}{\text{case } E \text{ of } (LF, RF) \Rightarrow V}$$

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

$$\text{(E-FINDALL FAIL)}\ \frac{E\ \text{logicVar} \Rightarrow \text{fail}}{\text{findAll}_T\ E \Rightarrow (\text{inr unit})}$$

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

$$\text{(E-FREE-FAIL)}\ \frac{E\ \text{logicVar} \Rightarrow fail}{\text{free}_T\ E \Rightarrow fail}$$

Note that in these rules we need no context, since $V$ should not contain free variables. Importantly, the big step semantics also have the useful property that the result of a reduction is always in weak head normal form.

**Definition 3.** Let $E \Rightarrow_S R$be the non strict single step reduction

$$\text{(ES-APP-LAM1)}\frac{E_1 \Rightarrow_S E_1'}{E_1\ \ E_2 \Rightarrow_S E_1'\ E_2}$$

$$\text{(ES-APP-LAM2)}(\lambda x : t.F)\ \ E_2 \Rightarrow_S [x \mapsto E_2]F \text{ provided x is free for } E_2 \text{ in } E_1$$

$$\text{(ES-GET1)}\ \frac{E \Rightarrow_S E'}{\pi_i E \Rightarrow_S \pi_i E'}$$

$$\text{(ES-GET2)}\ \pi_i(E_1, E_2) \Rightarrow_S E_i$$

$$\text{(ES-SWITCH1)}\ \frac{E \Rightarrow_S E'}{\text{case } E \text{ of } (LF,RF) \Rightarrow_S \text{case } E' \text{ of } (LF,RF)}$$

$$\text{(ES-SWITCH-LEFT) case inl L of (LF,RF)} \Rightarrow_S \text{LF L}$$

$$\text{(ES-SWITCH-RIGHT) case inr R of (LF,RF)} \Rightarrow_S \text{RF R}$$

$$\text{(ES-FINDALL-SUCC)}\ \frac{\vdash V : T\ \ (E\ V) \Rightarrow_S *\text{success}}{\text{findAll}_T\ E \Rightarrow_S (inl\ V)}$$

$$\text{(ES-FINDALL FAIL)}\ \frac{E\ \text{logicVar} \Rightarrow_S *\text{fail}}{\text{findAll}_T\ E \Rightarrow_S *(\text{inr unit})}$$

$$\text{(ES-FREE-SUCC)}\ \frac{\vdash V : T \quad E\ V \Rightarrow_S *\text{success}}{\text{free}_T\ E \Rightarrow_S \text{success}}$$

$$\text{(ES-FREE-SUCC)}\ \frac{\vdash V : T \quad E\ \text{logicVar} \Rightarrow_S *fail}{\text{free}_T\ E \Rightarrow_S fail}$$

Note, for sake of brevity, the full single step reduction shall not be defined here. Instead it will be assumed that it is defined as it usually is and has the following operational properties.

## 2.3 Type and Mode Checking

Modes are a system for characterizing what arguments are input and outputs for a predicate. We say that a value is ground or resolved if all of its constituents are known. We can say that an argument is intended to be an output to a predicate if, upon satisfying that predicate, it must be ground. An argument is an input to a predicate if it will be ground on every instantiation of the rule.

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 use 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 4.** The immediate mode of a type $t \sim m$ is defined as follows.

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

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

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

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

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 5.** The safe mode of a type $t \approx m$ is inductively defined as follows

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

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

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

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

**Lemma 6.** *All types have a unique immediate mode.*

*Proof.* The proof is trivial by induction on the structure of a type. □

**Lemma 7.** *If a type has a safe mode, it has the same immediate mode.*

*Proof.* The proof is by induction on the structure of the predicate $t \approx m$. The intuition here is that $t \sim m$ produces $m$ in the same way that $t \approx m$ produces $m$, but $t \approx m$ performs extra checks. □

Before we can discuss type checking, we need to explain the subtype relation.

**Definition 8.** $T \leq T'$ simply means that $T$and $T'$ have the same structure but different modes. At each sum, $m \leq m'$.

$$1 \leq 1$$

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

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

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

The following lemmas are relied on heavily thought the proof of preservation and progress, and have been entirely formalized in Twelf in the TypeTheorems file.

**Lemma 9.** *$T \leq T$ is admissible.*

*Proof.* This is a straightforward proof by induction. □

**Lemma 10.** *$\frac{A \leq B \quad B \leq C}{A \leq C}$ is admissible.*

*Proof.* By lexicographic induction on the structure of the first and second subtyping relations. □

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

*Proof.* By lexicographic induction on the structure of the subtyping relation and mode derivations. □

## 2.4 First Pass

Given that modes are an annotation on the type system, mode checking and type checking are done in two passes. The first pass ensures that the types with mode annotations are used reasonably, while the second pass ensures that functions marked as resolving use their arguments at least once somewhere that resolves inputs, and will be subsequently called. In order to keep type checking simple, we provide it as a lemma that the types produced from type checking are mode safe.

**Definition 12.** $E : T$ means that expression $E$ has type $T$ and is defined as follows.

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

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

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

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

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

$$\text{(OF-PROJ)}\frac{\Gamma \vdash a : v_1 \times v_2 \quad v_1 \times v_2 \approx m}{\Gamma \vdash \pi_i a : v_2}$$

$$\text{(OF-Left)}\frac{\Gamma \vdash a : v_1 \quad v_1 +_m v_2 \approx m}{\Gamma \vdash \text{inl } a \;\; : v_1 +_m v_2}$$

$$\text{(OF-RIGHT)}\frac{\Gamma \vdash a : v_2 \quad v_1 +_m v_2 \approx m}{\Gamma \vdash \text{inr } a \;\; : v_1 +_m v_2}$$

$$\text{(OF-OBJ)}\frac{\Gamma \vdash e_1 : v_1 \;\; v_1 \sim m \quad \Gamma \vdash e_2 : v_2 \;\; v_2 \sim m}{\Gamma \vdash (e_1, e_2) : \;\; v_1 \times v_2}$$

$$\text{(OF-UNIT)}\frac{}{\Gamma \vdash \text{unit} : 1}$$

$$\text{(OF-CASE-OF)}\frac{\Gamma \vdash l : v_1 \to v_3 \quad \Gamma \vdash r : v_2 \to v_3 \;\; \Gamma \vdash e : v_1 +_m v_2 \;\; v_1 \approx m \quad v_2 \approx m}{\Gamma \vdash \text{case } e \text{ of } l \; r : \;\; v_3}$$

**Proposition 13.** *$\Gamma \vdash E : T$ implies that there is a unique $m$ such that $T \approx m$.*

*Proof.* The proof is by induction on the structure of a proof of type. There are only three interesting cases however. In the case of subsumption, we already ensure the type is well moded. In the case of application, the resulting safe mode proof for the left hand side after an application of induction must end in

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

since it is the only case for an arrow type. Since the type of $E_1 \; E_2$ is $t_2$ we have a proof of $t_2 \approx m_2$.

The rest of the proof is formalized in the associated Twelf file, TypeCheckingTheorems.elf. □

As our analysis does does not involve **free, findAll**, **success** or **fail**, we will give their types in polymorphic terms for ease of understanding their actions.

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

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

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

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

We can now introduce some short hand for the remainder of the paper. Given that $E$ has already been type checked, we can write that it has a unique type $T$ as $E :: T$. This can be ensured in an implementation by annotating each term in the abstract syntax tree with its associated type during type checking. Also, the shorthand $E/m$ is used for $E :: T$ and $T \sim m$.

## 2.5 Second Pass

In the second pass, we ensure that functions annotated as resolving use their arguments as input to another resolving function in a place that will get used. The predicate $usedCorrectly_m(x, E)$ ensures that $x$ is used as a negative argument to some function in $E$. We can generalize even further to allow object arguments by creating another predicate $objectUsedCorrectly(x, E)$. 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.

**Definition 14.** $usedCorrectly_m(X, E)$ is defined according to the following rules, on type annotated terms $E$ and $X$ and mode $m$.

$$\text{(USED-CORRECTLY/VAR)}\frac{}{usedCorrectly_{\ominus}(X, X)}$$

$$\text{(USED-CORRECTLY/LAM)}\frac{usedCorrectly_m(X, [y/x]E) \quad \text{x is new}}{usedCorrect_m(X, \lambda y : t.E)}$$

$$\text{(USED-CORRECTLY/APP-L)}\frac{usedCorrectly_m(X, E_1)}{usedCorrectly_m(X, E_1 E_2)}$$

$$\text{(USED-CORRECTLY/APP-R)}\frac{usedCorrectly_{\ominus}(X, E_2) \;\; E_1 :: \; t_1 \to t_2 \;\; t_1 \sim \ominus}{usedCorrect_m(X, E_1 E_2)}$$

$$\text{(USED-CORRECTLY/OBJ)}\frac{usedCorrectly_m(X, E_1) \;\; usedCorrectly_m(X, E_2)}{usedCorrect_m(X, (E_1, E_2))}$$

$$\text{(USED-CORRECTLY/OBJ-L)}\frac{usedCorrectly_m(X, E_1)\ \ (E_1, E_2)/\ominus}{usedCorrect_m(X, (E_1, E_2))}$$

$$\text{(USED-CORRECTLY/SUM-ARG)}\frac{usedCorrectly_m(X, E)}{usedCorrect_m(X, \text{case } E \text{ of } E_1\ E_2)}$$

$$\text{(USED-CORRECTLY/SUM-C)}\frac{usedCorrectly_m(X, E_1)\ \ usedCorrectly_m(X, E_2)}{usedCorrect_m(X, \text{case } E \text{ of } E_1\ E_2)}$$

We also can show the expansion of an object.

**Definition 15.** $objUsedCorrectly_m(X, E)$ is defined according to the following rules.

$$\text{(OBJ-USED/EXP)}\frac{usedCorrectly_m(X, E)}{objUsedCorrect_m(X, E)}$$

$$\text{(OBJ-USED/PROD)}\frac{objUsedCorrectly_m(getLeft\ X, E)\ \ objUsedCorrectly_m(getRight\ X, E)}{objUsedCorrect_m(X, E)}$$

**Definition 16.** The predicate $wellModed(E)$ traverses the syntax of a term and only has two rules that are of interest:

$$\text{(WM-LAM/POS)}\frac{t \sim \oplus \quad \Gamma, wellModed(x) \vdash wellModed(E)}{\Gamma \vdash wellModed(\lambda x : t.E)}$$

$$\text{(WM-LAM/NEG)}\frac{t \sim \ominus \quad \Gamma, wellModed(x) \vdash wellModed(E) \quad objUsedCorrectly(x, E)}{\Gamma \vdash wellModed(\lambda x : t.E)}$$

**Lemma 17.** *A decidable algorithm to infer types and check well modedness implies mode inference is also 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. □

## 2.6 Preservation

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

**Theorem 18.** *Preservation*
$E \Rightarrow E'$ *and* $\vdash E : T$ *then* $\vdash E' : T'$ *and* $T' \leq T$.

*Proof.* The proof is by lexicographic induction on the reduction and proof of type, and it has been formalized in Twelf. The intuition behind the subsumption is that as reductions occur, we might find out more about the resulting value, but nothing we already know about the resulting value will become invalidated. In the case of subsumption of a right hand side, the argument for termination is more subtle. The invariant is made that in an inductive step, no new the size of the proof of $of$ never increases, and $of/subsump$ instances always move to the right hand side in the inductive call or disappears entirely. □

## 2.7 Runtime Properties

The first important theorem about these semantics is that they always result in precisely weak head normal form of a value, and thus capture an entire computation and not just a computational step.

**Definition 19.** $whnf(E)$ is defined as follows:

$$\text{(WHNF-OBJ)}whnf((A, B))$$

$$\text{(WHNF-UNIT)}whnf(\text{unit})$$

$$\text{(WHNF-INL)}whnf(\text{inl } A)$$

$$\text{(WHNF-INR)}whnf(\text{inr } A)$$

$$\text{(WHNF-SUCC)}whnf(\text{success})$$

$$\text{(WHNF-FAIL)}whnf(\text{fail})$$

**Theorem 20.** *If* $E \Rightarrow V$ then $whnf(V)$

*Proof.* Simply by lexicographic induction on the structure of a reduction. In cases where we reduce twice, the induction argument is invoked on the second, output term. The E-ID-$VTP$ cases form the base cases of the proof, where the output proof of $whnf$ is WHNF-$VTP$. □

**Lemma 21.** *if* $E \Rightarrow R$ *then* $E \Rightarrow_S *R$

**Lemma 22.** *if* $E \Rightarrow_S *R$*and* $whnf(R)$ *then* $E \Rightarrow R$

These two lemmas show an equivalence between the big step and single step semantics that will be useful to state the progress theorem.

## 2.8 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 logic variables.

**Definition 23.** logicFree($E$) shall mean that the term $E$ contains no logic 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.

$$\text{(LFREE/APP)}\frac{logicFree(V)\ \ logicFree(F)}{logicFree(F\ \ V)}$$

$$\text{(LFREE/LAM)}\frac{logicFree(E)}{logicFree(\lambda x.E)}$$

$$\text{(LFREE/VAR)}\frac{}{logicFree(x)}$$

$$\text{(LFREE/UNIT)}\frac{}{logicFree(unit)}$$

$$\text{(LFREE/OBJ-L)}\frac{logicFree(E_1)\ \ E_2/\oplus}{logicFree((E_1, E_2))}$$

$$\text{(LFREE/OBJ-R)}\frac{logicFree(E_2)\ \ E_1/\oplus}{logicFree((E_1, E_2))}$$

$$\text{(LFREE/OBJ)}\frac{logicFree(E_1)\ \ logicFree(E_2)}{logicFree((E_1, E_2))}$$

$$\text{(LFREE/INL)}\frac{logicFree(E)}{logicFree(\text{inl } E)}$$

$$\text{(LFREE/INR)}\frac{logicFree(E)}{logicFree(\text{inr } E)}$$

$$\text{(LFREE/PROJ)}\frac{logicFree(E)}{logicFree(\pi_i\ E)}$$

$$\text{(LFREE/SWITCH)}\frac{logicFree(E)\ \ logicFree(L)\ \ logicFree(R)}{logicFree(\text{case } E \text{ of } L\ R)}$$

**Lemma 24.** *If* $E \implies_{\alpha} E'$ *and* $logicFree(E)$ *then* $logicFree(E')$

**Theorem 25.** $usedCorrectly_m(X, E)$ *and* $logicFree(E)$ *implies* $logicFree(X)$.

*Proof.* By induction on the structure of $E$.

The only case of interest here is USED-CORRECTLY/LAM along with LFREE/LAM. In this case, where we receive $usedCorrectly_m(X, \lambda x.E)$ and $logicLess(\lambda x.E)$. We then know $usedCorrectly(X, [x/y]E)$ for some new $y$. Thus, $E \implies_{\alpha} [x/y]E$ and $logicLess(E)$ so $logicLess([x/y]E)$ by the above lemma. By the induction hypothesis, we know $logicLess(X)$. □

**Corollary 26.** $objUsedCorrectly_m(X, E)$ *and* $logicFree(E)$ *implies* $logicFree(X)$.

The following theorem is vital to progress, as it ensures that when we search by applying values with logical variables in them to the search predicate, a successful result can only occur when the argument in fact has no logic variables.

**Theorem 27.** *Subterm-Resolving*

$\Gamma \vdash X : T_x$ *and* $usedCorrectly_{\oplus}(X, E)$ *and* $\Gamma \vdash E : T_E$ *and* $T_E \approx \oplus$ *and* $E \Rightarrow_* V$ *where* $\Gamma \vdash V : T_V$, *and logicFree(V) then* $logicFree(X)$

*Proof.* This has been mostly proved in Twelf without subtyping for all but a few cases. The intuition behind this theorem is that *usedCorrectly* accounts for 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. The remaining cases not currently handled in the Twelf proof are those involving expressions used negatively on the left hand side. In these cases, subterm resolving follows from preservation of *usedCorrectly* under substitution. □

**Corollary 28.** $\Gamma \vdash X : T_x$ *and* $objUsedCorrectly_{\oplus}(X, E)$ *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* $logicFree(X)$*and* $logicFree(E)$ *implies* $logicFree(X)$.

**Corollary 29.** *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$(X, X)$. □

**Corollary 30.** *WellModed-Resolving*

*wellModed*$(E)$ *and* $\Gamma \vdash E : T_E$ *and* $T_E \approx \oplus$ *and* $E \Rightarrow_S *V$ *with* $\Gamma \vdash V : T_V$ *and* $T_V \leq T_E$ *by preservation and logicFree(V) then* $logicFree(E)$

**Theorem 31.** *If* $wellModed(E)$ , $logicFree(E)$ *and* $E \implies E'$ *then* $logicFree(E')$.

*Proof.* This theorem is by lexicographic induction. The only non-trivial cases are those where a logical variable could potentially be introduced. This restricts us to the cases of $findAll$ and $free$.

In the case of $findAll$, there are two applicable reductions. In the case of E-FINDALL-SUCC, corollary 27 can be applied to the reduction $E\ V \Rightarrow$ success to show $logicFree(V)$. The output $logicFree(left\ V)$ is an application of LFREE-INL. The case of E-FINDALL-FAIL is trivial.

In the case of $free$ , both reductions involving it must return values which are certainly resolved. □

**Theorem 32.** *Progress Holds*

$E : T$ *and* $T \approx \oplus$ *implies either* $E$ *is a value or* $E \Rightarrow_S E'$ *for some* $E'$*. Furthermore, if* $E$ *does not contain logic variables, then* $E'$ *will also contain no logic variables.*

*Proof.* This proof is by lexicographic induction, application of theorem 28, the relationship between the small step and big step semantics, and theorem 18. Appart from the cases of $findAll$ and $free$, the proof is similar to traditional proofs of progress. We proceed as normal, until the cases of $findAll$ and $free$. It makes more sense to say here that $findAll$ of any expression is a value. A more meaningful proof of progress can be given if a queue of attempting expressions and a set of unattempted expressions is included in the semantics for $findAll$. □

## 2.9 Strictness Analysis

**Definition 33.** $hnf$ is the function that evaluates its argument to head normal form.

**Definition 34.** A function E is grounding if when $E\ V \implies C$ and $C$ is a value, then $hnf(V)$ converges and has no logic variables.

**Definition 35.** A term$E$ is strict if $E\ (hnf\ x) \equiv E\ x$ for all terms $x$ such that $E\ x$ converges to a value or diverges.

**Proposition 36.** *$E$ is strict implies $E$ is grounding.*

*Proof.* Suppose $E$ is strict. Then suppose we were to pass an argument to $E$ with logic variables. 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 logic variables passed to $E$ will either diverge or converge to a value with logic variables. Thus $E$ is grounding. □

**Corollary 37.** *A combinator $E$ is strict implies $\vdash E : I \to O$ and $I \sim \ominus$ and $O \sim \oplus$.*

### 2.9.1 Ramifications

Because mode checking and strictness checking are equivalent, mode checking algorithms can be used for strictness analysis. Natural deduction formulations of strictness and relevance logic have been given relevant proof terms and could very well be used as an alternate type system, with the addition of new syntax. In particular, our use of the predicate *usedCorrectly* coresponds the notion of a strict use of it's argument. In this formulation product types can be treated a bit more flexibly.

The polymorphic linear lambda calculus Lily has been shown to have equivalent termination properties under call-by-value and call-by-need semantics.[7] In principle, terms like **findAll** might be added to Lily and be have groundedness properties ensured.

# 3 Conclusions & Future Work

In this thesis, we discussed a lazy functional logic language similar to Curry and proved its runtime safety. We supplied a mode system and a way to separate nondeterminism from input and output. However, the mode system is not always entirely expressive enough. It is also still necessary to show the safety of extensions of polymorphism and recursion to the current mode algorithm. Finally, a full implementation of the language has yet to be completed.

Strictness type systems based on natural deduction formulations of resource logic have been defined, but type systems based on them including fixed points and polymorphism do not yet seem to have been examined. It would also be interesting to explore the sorts of safe concurrency primitives that could be created using the same principles as **findAll**.

# References


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

[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 (2002), 207–222.

[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 (2004), 18–33.

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

[5] S. Antoy and M. Hanus, *Functional logic programming*, Communications of the ACM **53** (2010), no. April, 74–85.

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

[7] G.M. Bierman, A. M. Pitts, and C. V. Russo, *Operational properties of lily, a polymorphic linear lambda calculus with recursion.*

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

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

[10] M. Hanus, *Curry: An integrated functional logic language.*

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

[12] Frank Pfenning, *Refinement types for logical frameworks*, Informal Proceedings of the Workshop on Types for Proofs and Programs (Nijmegen, The Netherlands) (Herman Geuvers, ed.), May 1993, pp. 285–299.

[13] Benjamin C. Pierce, *Types and programming languages*, The MIT Press, 2002.

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

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

*Remark.* The Twelf proofs will be available at http://github.com/mmirman/korma in the branches simple and explicit.