Abstract Semantics For Functional Constraint Programming[1]

Surya Mantha, Lal George, and Gary Lindstrom

UUCS-89-022

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

1989

Abstract

A denotational semantics is given for a lazy functional language with monotonic side-effects arising from the unification of singly-bound logical variables. The semantics is based on a Scott-style information system, which elegantly captures the notion of "constraint additin" inherent in unification. A novel feature of our approach is exploitation of the representational duality of denotations defined by information systems: (i) as domain elements in the traditional sense, and (ii) as sets of propositions or constraints. Spread care is taken to express accurately the interactions of lazy evaluation (e.g. evaluation by need), and read-only accesses of logical variables defer function applications. The purpose of our semantic description is to establish language properties such as determinacy under parallel evaluation, to validate implementation strategies, and to support the design of program analysis techniques such as those based on abstract interpretation.

[1]This material is based upon work supported by the National Science Foundation under Grant No. CCR-8704778.

Parallel Programming Research Group
University of Utah, Department of Computer Science
Note PPRG-89-TR-89-022

Abstract Semantics For Functional Constraint Programming*

SURYA MANTHA (*mantha@cs.utah.edu*)
LAL GEORGE (*george@cs.utah.edu*)
GARY LINDSTROM (*lindstrom@cs.utah.edu*)
University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

Keywords: lazy functional languages, logical variables, semantics, information systems

Abstract. *A denotational semantics is given for a lazy functional language with monotonic side-effects arising from the unification of singly-bound logical variables. The semantics is based on a Scott-style* information system, *which elegantly captures the notion of 'constraint addition' inherent in unification. A novel feature of our approach is exploitation of the representational duality of denotations defined by information systems: (i) as domain elements in the traditional sense, and (ii) as sets of* propositions *or* constraints. *Special care is taken to express accurately the interactions of lazy evaluation (e.g. evaluation by need), and* read-only *accesses of logical variables defer function applications. The purpose of our semantic description is to establish language properties such as determinacy under parallel evaluation, to validate implementation strategies, and to support the design of program analysis techniques such as those based on abstract interpretation.*

1 Introduction

Functional languages are predicated on the absence of side effects, which results in both elegance and shortcomings as general purpose programming languages. The absence of side effects requires that all variables be bound immediately upon introduction. In the absence of update analysis, single threading, or other sophisticated techniques, aggregate data structures must be copied whenever a component is updated. This not only results in unacceptable space consumption, but also entails time expense in copying data values and recycling inaccessible storage.

The need to overcome these problems has long been recognized. One avenue receiving considerable attention is the *amalgamation* of functional languages with other *declarative* models of programming supporting *disciplined* side effects, e.g.

- logic programming languages and
- equational languages, e.g. term rewriting systems.

These amalgamation efforts all recognize the important role played by *logical variables*, arguably the most important concept to emerge from logic programming. The reader is referred to [10], [12] for a detailed account of the benefits of incorporating logical variables into functional languages.

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-8704778.

We shall, however, give a brief summary of the advantages of such an integration. Logical variables enhance the already powerful data structuring capabilities and modular nature of functional languages by allowing

- Monotonic side effects;
- Use before definition of data structures;
- Multi-reader multi-writer communication channels between processes, thereby providing a richer notion of communication and synchronization than is provided by the exclusively producer-consumer model in functional programming, and
- An elegant notion of functional programming with *constraints.*

2 The CoF Language

CoF (*Communicating Functions*) is a lazy functional language with logical variables. Its evaluation strategy — graph reduction — molds well onto a parallel machine, in that the *closures* used to support lazy evaluation are naturally implementable as variable grain tasks. Parameter passing is done by pattern matching, a familiar concept in functional languages. This causes a rewrite or reduction to be suspended if the actual arguments are not evaluated or are insufficiently instantiated. For example, if the actual argument happens to be an unbound logical variable where a destructuring pattern match is required, then the rewrite suspends awaiting a binding of that variable. Special care will be taken to express accurately the interactions of lazy evaluation (e.g. evaluation by need), and *read-only* accesses of logical variables defer function applications. Note, however, that in *CoF* the degree of evaluation thoroughness of expressions is semantically important, rather than being simply an efficiency or divergence control issue. In particular, over-evaluation can needlessly lead to errors, due to the gratuitous application of conflicting constraints (unifications). Hence our semantics must exhibit $\top$-avoidance as well as $\bot$-avoidance!

We give the abstract syntax of *CoF* and refer the reader to [5] for details on its design. *CoF* is based on an *ML* syntax with two syntactic extensions to expressions to handle unification and the introduction of logical variables.

```
exp ::= Nat
      | Bool
      | nil
      | Id
      | exp1 :: exp2
      | hd exp1
      | tl exp1
      | exp1 exp2
      | lambda Id exp
      | let Id = exp1 in exp2
      | let Id = ubv() in exp
      | exp1 assuming exp2 == exp3
      | exp1 prim_op exp2
      | if exp1 then exp2 else exp3
```

The symbol == denotes the unification operator, which occurs only within an **assuming** expression. The value of the **assuming** expression is exp_1, which is strict upon successfully unifying exp_2 and exp_3. The special form **Id** = `ubv()` binds **Id** to a new (unbound) logical variable. Logical variables are singly bound to preserve the functional semantics. Hence no backtracking or *relational* results are utilized, and unification failure is a fatal error. Logical variables are first class objects in *CoF*, e.g. they may be passed as arguments to functions and returned by function applications. While referential transparency is lost (because `ubv()` $\neq$ `ubv()`), the top level determinacy of functional programs is preserved, as we will later demonstrate. The semantics of purely functional *CoF* programs (those without the `ubv()` and **assuming** constructs) is quite conventional.

3 A Semantic Model Based On Information Systems

As observed in the previous section, logic programming and functional programming are becoming more closely related. We build here on important contributions made by other researchers seeking abstract semantics for languages combining these two paradigms.

- Pingali et al. [6] give an abstract semantics for a functional language with logical variables based on solving a set of equations involving an environment. Their language *Cid* is eager and hence does not have lazy constructors.

  Our semantic framework employs a purely functional meta-language rather than equation solving as in [6]. In fact, the notion of *constraint solving* is an inherent part of *CoF* semantics unlike in *Cid*. We adopt Saraswat's [17] notion of a global logical store as a repository for the constraints that have been imposed by program execution. In Saraswat's terminology, *ask* (read-only) constraints arise in *CoF* through parameter passing, and *tell* constraints arise from *unify* operations.

- Kieburtz was the first to use Scott's information systems [20] in this arena. Reference [8] gives a relational semantics for the logic component of a language with functions and Horn clauses, rather than the tradmtional semantics based on Herbrand models. The motivation there is to develop an integrated *functions and logic* language *F+L* that provides the flexibility of relational definitions while retaining the power of general applicative expressions.

  CoF is less ambitious in that the functional basis is preserved by requiring logical variables to be singly bound. Also, logical variables are fully integrated into our functional framework, as opposed to constituting a stylized interface between the applicative and relational sublanguages as in *F+L*. Recent work [13] on *open languages* holds the promise of a clean interface shared among languages such as these, using polymorphic type information.

Like the work of Kieburtz, our denotational semantics is based on Scott's *information systems*. Information systems are an elegant way of characterizing monotonic constraint addition. This alternative semantic foundation was first introduced by Scott [20] as an attempt to demystify the theory of domains. The underlying motivation for the study of domains is to formulate a notion of computability with respect to recursively defined types containing infinite elements. This *infinitude* necessitates the

introduction of the notions of *approximation* and *total* and *partial* elements. Scott gives an elegant formulation of domains in terms of information systems with the following advantages:

- Information system definitions are constructive, with complex domains can be obtained from simpler ones in a straightforward manner.
- Domains are given a simpler structure, based on classical set theory.
- Very few axioms are used, and they can be tailored to suit many applications.
- Elements of domains are endowed with more structure; consequently it is easier to reason about them.

The key insight in the information system approach to domains is that *possible elements* of the desired domain are computed by accumulating *propositions* that monotonically refine an approximation of the element. The computation of a domain element involves the monotonic enlargement of a set of *data objects*. A single data object contributes a fragment of information about the element being computed in the form of a *unary predicate*. The data objects are, in a sense, exclusive predicates about the elements of the domain of interest. We can look upon a set of data objects as the conjunction of a set of constraints, each contributing information about the element. A data object thus constitutes a *conjunctive proposition* about an element being computed. It is obvious that the data objects should not contribute conflicting information. If they do, then the result of the computation is *error*.

We shall define all relevant concepts as we develop the semantics for *CoF*. In the detailed and thoroughly readable paper [20] Scott describes the construction of domains from information systems, traces the topological connections and also gives examples of constructing complex domains from simpler ones.

4 Domains for CoF

Definition 1 *An* information system *is a 4-tuple*

$$D_A = (A, \Delta, Con_A, \vdash)$$

where A is the set of data objects, Δ is a distinguished element in A that has no information, Con_A is the set of finite and consistent subsets of A and $\vdash$ is a binary entailment relation between members of Con_A and elements of A.

The entailment relation $\vdash$ can easily be extended to a relation between elements of Con_A. It is specified by the following axioms.

1. $(u \vdash \Delta)\ \forall u \in Con_A$.
2. $(u \vdash x)$ if $(u \in Con_A)\ \wedge\ x \in u$.
3. $(u, v \in Con_A), ((u \vdash x)\ \forall x \in v)\ \wedge\ (v \vdash y)$ then $u \vdash y$.

Our set A is given by

$$A = \{\Delta\} \cup \aleph \cup Bool \cup Vars \cup \Im$$

where $\aleph$ is the set of integers, $Vars$ are logical variables and $\Im$ is the set

$$\{pair(x, y) \mid x, y \in A\}.$$

The set Con_A of finite and consistent subsets comprises:

- The singleton sets $\{\Delta\}$ and $\{x\}$ where x is an element of either $\aleph$, $Bool$ or $Vars$;
- Two-element sets of the form $\{x, \Delta\}$, where x is an element of either $\aleph$, $Bool$ or $Vars$, and
- Finite sets whose elements are either Δ or $pair(x, y)$, such that the first elements of all the pairs form a consistent set (with the relaxation that the set so formed could have more than one logical variable) and so do the second elements. An example is

$$\{\Delta, pair(\Delta, X), pair(\Delta, \Delta), pair(4, Y)\}$$

- No other sets belong to Con_A.

We add a fourth axiom to the entailment relation $\vdash$ to model the intended structure of A.

- $(u \vdash pair(x, y))$ if $\exists\{pair(x_1, y_1), pair(x_2, y_2)\} \subset u$, such that $(\{x_1, x_2\} \vdash x)$ and $(\{y_1, y_2\} \vdash y)$.

With the above machinery we are ready to define the elements of the domain $|D_A|$ generated by the information system D_A. The *elements* of $|D_A|$ are the entailment closed, consistently upwardly closed subsets of A (*ideals* of the complete lattice $\mathcal{P}(A)$). Thus, every finite subset of an element belongs to Con_A. For instance, the element $pair(3,4)$ is represented by the constraint set

$$\{\Delta, pair(\Delta, \Delta), pair(3, \Delta), pair(\Delta, 4), pair(3, 4)\}$$

We shall use elements and the constraint sets that represent them interchangeably (which is, really, the key to our semantic formulation). The partial order between elements is the familiar set inclusion relationship, applied to their associated constraint sets. The least element of the domain is given by:

$$\bot = \{x \mid \{\Delta\} \vdash x\}$$

The *total elements* of $|D_A|$ are those to which information cannot be added without introducing inconsistencies. We specially add to $|D_A|$ a top element $\top$, interpreted as *error*. Note that *error* is an element of neither A nor Con_A. *Partial elements* are those to which more information can be added without raising to *error*. Rather, it is the interpretation given to inconsistent subsets of A. The *finite elements* are obtained by the entailment closure of elements of Con_A.

We next define the notion of an *approximable mapping* (function). An approximable mapping f_{ab} between two information systems D_A and D_B is a binary relation between their finite and consistent subsets Con_A and Con_B such that

1. $\Phi\ f_{ab}\ \Phi$.

2. $u_A\ f_{ab}\ u_B,\ u_A\ f_{ab}\ v_B \Longrightarrow u_A\ f_{ab}\ (u_B \cup v_B)$.

3. $v_A \vdash_A u_A,\ u_A\ f_{ab}\ u_B,\ u_B \vdash_B v_B$ then $v_A\ f_{ab}\ v_B$.

Intuitively, the first condition states that if a function's input is totally undefined, then we may expect it not to give any answer. The second condition states that if a function gives two pieces of information about an element, then they must be consistent. The last condition states that on refining the input to a function, the computations already performed still hold and no recomputation has to be done.

Scott [20] shows that approximable mappings between D_A and D_B take *elements* to *elements*, i.e. $|D_A|$ to $|D_B|$. Constructing an information system D_{AB} whose data objects consist of binary tuples of elements from Con_A and Con_B, and whose elements are approximable mappings from D_A to D_B, is quite straightforward and we omit it here. The reader is referred to [20] for details.

Winskel [23] shows that recursive domain equations based on information systems can be solved effectively. The domain that we are interested in is given by the set of elements $|Dom|$ of the information system Dom which is the solution to the recursive equation

$$Dom = D_A + (Dom \rightarrow Dom)$$

where D_A is as defined above.

5 Semantic Functions

The explicit construction of domain elements from data objects affords us a finer level of reasoning than is typically afforded by the usual notion of an element. We can operate directly on the data objects in order to formalize the notion of *value refinement.* As stated earlier, the basic model of computation is that of value refinement through constraint addition. In accordance with this philosophy, program constructs will be viewed as imposing constraints on a *logical store,* comprising a mapping from logical variables (cf. "locations") to constraint sets (cf. value approximations). We first define the various domains used in the semantic functions.

D	=	$\|Dom\|$	overall domain
Id			lexical identifiers
Exp			expressions
$Vars$			logical variables $\subset A$
Den	=	$Vars$	denotable values
Env	=	$Id \rightarrow Den$	environment
Ans	=	$D + (Vars \times D)$	answers
Exp_val	=	$(Ans \times Lstore) + error$	expressible values
$Suspension$	=	$Lstore \rightarrow Exp\_val$	suspensions
$Constraints$	=	$D + Suspension$	constraints
$Lstore$	=	$((Vars \rightarrow Constraints) \times Vars) + error$	logical stores

As suggested above, $Vars$, the domain of *logical variables,* is analogous to the domain of *locations* in imperative semantic formulations. Den is the domain of *denotable*

values, i.e. those values that can be bound to identifiers. For convenience in our semantic specification, identifiers can denote only logical variables. This is exploited to achieve *laziness,* i.e. to model the call by need argument evaluation strategy of *CoF*. *Env* is the domain of environments, which map identifiers to logical variables (denotable values). *Ans* is the domain of *answer values,* representing the side-effect free aspect of expression evaluation. The second summand of *Ans* reflects the fact that the value of a variable is a pair with the name of the variable as its first component and its value as the second component. This will permit the *unify* function to deal properly with both bound and unbound variables. Non-*error* expressible values, the complete result of expression evaluation, are pairs including a logical store conveying any variable binding side-effects. *Suspensions* is the domain of closures. These are functions that take a logical store and return an expressible value.

Constraints is analogous to the domain of *storable values.* This is the domain of objects that constrain the values of variables in the logical store. The store includes a $Vars$ component used as a basis for generating fresh logical variables. The domain $Vars$ can be modeled using natural numbers; in fact any flat (countably infinite) domain will serve our purpose. The only operation we use in posting the constraints to the logical store is set theoretic union. It has the desired properties of idempotence and monotonicity, and hence forms a *closure operator.*

In cases were disjoint sums are used, the summands must be ordered to ensure the monotonicity of our semantic functions. In *Exp_val* and *Lstore* we set *error* to be greater than its summand partner; in *Ans*, we stipulate $Vars \times D \sqsubseteq D$

The semantic function $\mathcal{E}$ takes an expression, an environment and a logical store, and returns an answer value and a new logical store. Since *CoF* is higher order and logical variables are first class objects, top-level answers can be functions or unbound logical variables. The signature of $\mathcal{E}$ is:

$$\mathcal{E}\colon Exp \rightarrow Env \rightarrow Lstore \rightarrow \langle Ans, Lstore \rangle$$

We now discuss several auxiliary functions that support our formal semantics.

1. *Lookup_env* takes an identifier, an environment, and a logical store as arguments, and calls *Lookup_LS_and_force* with the logical store and the binding of the identifier in the environment.

   $$\text{lookup\_env}\colon Id \rightarrow Env \rightarrow Lstore \rightarrow Exp\_val$$

2. *Lookup_LS_and_force* takes a variable and a logical store as arguments and returns the constraints (the set of data objects) that apply on it along with a possibly new store. If the value of the variable is a suspension (i.e. an unevaluated expression), then it forces evaluation (application to the current logical store) and constructs a new logical store reflecting the result of the evaluation.

   $$\text{lookup\_LS\_and\_force}\colon Vars \rightarrow Lstore \rightarrow Exp\_val$$

   Note, that suspensions (or closures) are evaluated only once. All subsequent references to the variable use the value incorporated into the resulting logical store.

3. *Unify* takes two terms (elements of the domain Ans), a logical store, and returns a new logical store that is obtained as a result of the bindings performed.

$$\text{unify: } Ans \rightarrow Ans \rightarrow Lstore \rightarrow Lstore$$

4. *Merge_constraints* takes two variables, their respective constraints, a logical store, and returns a new logical store in which the constraints operating on the two variables have been merged.

$$\text{merge\_constraints: } Vars \rightarrow D \rightarrow Vars \rightarrow D \rightarrow Lstore \rightarrow Lstore$$

5. *Union* takes two sets of constraints (data objects) and a logical store, and returns the union of the two sets and a new logical store after ensuring consistency between the two sets of constraints.

$$\text{union: } D \rightarrow D \rightarrow Lstore \rightarrow \langle D \times Lstore \rangle$$

6. *New_LV* takes a logical store and returns a fresh logical variable and a new store.

$$\text{new\_LV: } Lstore \rightarrow \langle Vars \times Lstore \rangle$$

7. *Is_Consistent* utilizes set union to merge constraint sets, after ensuring consistency.

$$\text{Is\_Consistent: } D \rightarrow D \rightarrow LS\ rightarrow\ LS$$

We also define a function *fix_store* that takes a functional over a logical store, a current logical store as argument and iterates it until a fixpoint is obtained. Such a fixpoint is guaranteed to exist since the operators that we use in constructing our functionals are monotonic and continuous. Hence *fix_store* is crucial to our semantic formulation. By taking the fixpoint over a logical store, we ensure that the final logical store obtained is independent of the evaluation order. We present a proof sketch of this claim later in the paper.

$$\text{fix\_store: } (Lstore \rightarrow Lstore) \rightarrow Lstore \rightarrow Lstore$$

The definitions of the auxiliary functions are now given. For clarify and space economy, shortcuts are taken in certain cases, e.g. through the omission of *error* treatment, and in ignoring the $Vars$ component of $Lstores$ when it is irrelevant.

fix_store $F\ arg = \bigcup_{i=0}^{\infty} F^i(arg)$

lookup_env $x\ [x \mapsto v]\rho\ LS =$
  if IsVars(v) **then**
    lookup_LS_and_force $v\ LS$
  else
    error

lookup_LS_and_force $v\ LS =$
  let $[v \mapsto preds]\rho = LS$

```
    in
        if IsVars(preds) then
              lookup_LS_and_force preds ρ
        else
              if IsSuspension(preds) then
                  let
                      ⟨val , LS1⟩ = preds LS
                  in
                      ⟨⟨v , val⟩, LS1[v ↦ val]⟩
              else
                  ⟨⟨v , preds⟩, LS⟩

unify {△} y LS = LS
unify x {△} LS = LS
unify (a :BOOL) (b :BOOL) LS =
    if (a = b) then
        LS
    else error
unify (a :INT) (b :INT) LS =
    if (a = b) then
        LS
    else error
unify pair(x1, y1) pair(x2, y2) LS =
    fix_store (λtemp_ls.
                  let*
                      ⟨v1 , temp_ls1⟩ = lookup_LS_and_force x1 temp_ls
                      ⟨v2 , temp_ls2⟩ = lookup_LS_and_force x2 temp_ls1
                      ⟨v3 , temp_ls3⟩ = lookup_LS_and_force y1 temp_ls2
                      ⟨v4 , temp_ls4⟩ = lookup_LS_and_force y2 temp_ls3
                      temp_ls5 = unify v1 v2 temp_ls4)
                  in
                      (unify v3 v4 temp_ls5))
                  ) LS
unify ⟨v1 , con1⟩ ⟨v2 , con2⟩ LS =
    merge_constraints v1 con1 v2 con2 LS
unify ⟨v , con⟩ val LS =
    let ans = union con val LS
    in
        if IsError(ans) then
              error
        else
              snd(ans)[v ↦ fst(ans)]
unify val ⟨v , con⟩ LS =
    let ans = union con val LS
    in
        if IsError(ans) then
              error
        else
              snd(ans)[v ↦ fst(ans)]

merge_constraints vname1 conset1 vname2 conset2 L_env =
    if (conset1 = {△}) then
        L_env[vname1 ↦ {vname2, △}]
```

```
        else
            if (conset_2 = {△}) then
                L_env[vname_2 ↦ {vname_1, △}]
            else
                let f_ans = union conset_1 conset_2 L_env
                in
                    if IsError(f_ans) then
                        error
                    else
                    snd(f_ans)[vname_1 ↦ {△, vname_2} , vname_2 ↦ fst(f_ans)]

    union cons_1 cons_2 LS =
        if cons_1 = {△} then
            ⟨cons_2 , LS⟩
        else
            if cons_2 = {△} then
                ⟨cons_1 , LS⟩
            else
                if Is_Function(cons_1) or Is_Function(cons_2)
                    then error
                else
                    let
                        new_LS = Is_Consistent cons_1 cons_2 LS
                    in
                    if IsError(new_LS) then
                        error else
                        ⟨cons_1 ∪ cons_2 , new_LS⟩
    Is_Consistent {} y LS = LS
    Is_Consistent ({x} ∪ rest) cons_2 LS =
        let*
            fun is_c x {} LS = LS
                is_c x ({y} ∪ ys) LS =
                    let LS_1 = unify x y LS
                    in
                        if IsError(LS_1) then
                            error
                        else
                            is_c x ys LS_1
                    end
            val t_LS = is_c x cons_2 LS
        in
            if IsError(t_LS) then
                error
            else
                Is_Consistent rest cons_2 t_LS
```

We are now ready to define the semantic function $\mathcal{E}$: $Exp \rightarrow Env \rightarrow Lstore \rightarrow \langle Ans, Lstore \rangle$

```
E[[const]] env = λLS.⟨D(const) , LS⟩        (* const = Nat ∪ Bool ∪ {nil} *)
E[[x]] env = λLS.lookup_env(x env LS)
E[[e_1 :: e_2]] env =
```

```
        λLS.(let*
            ⟨v1 , LS1⟩ = new_LV(LS)
            ⟨v2 , LS2⟩ = new_LV(LS1)
            in
            ⟨pair(v1, v2) , LS2[v1 ↦ InSuspension(E[[e1]] env), v2 ↦ InSuspension(E[[e2]] env)]⟩)
E[[hd e]] env =
        λLS.(let ⟨v , LS1⟩ = E[[e]] env LS
            in
            if v = pair(v1, v2) then lookup_LS_and_force v1 LS1
            else error)
E[[tl e]] env =
        λLS.(let ⟨v , LS1⟩ = E[[e]] env LS
            in
            if v = pair(v1, v2) then lookup_LS_and_force v2 LS1
            else error)
E[[e1 e2]] env =
        λLS.(let ⟨f , LS1⟩ = E[[e1]] env LS
            in
            if isFunction(f) then
                let ⟨v , LS2⟩ = new_LV(LS1)
                in
                    f v LS2[v ↦ InSuspension(E[[e2]] env)])
            else error
E[[lambda I e]] env =
        λLS.⟨InFunction(λd : Vars. λls.E[[e]] env[I ↦ d] ls) , LS⟩
E[[let I = e1 in e2]] env=
        λLS.(let*
            ⟨v1 , LS1⟩ = new_LV(LS)
            env1 = env[I ↦ v1]
            in
            E[[e2]] env1 LS1[v1 ↦ InSuspension(E[[e1]] env1)]
E[[ubv()]] env = λLS.⟨{△} , LS⟩
E[[e1 assuming e2 == e3]] env =
        λLS.( let f_ls =
            fix_store (λtemp_ls.
                        let*
                            temp_ls1 = snd(E[[e1]] env temp_ls)
                            ⟨v2 , temp_ls2⟩ = E[[e2]] env temp_ls1
                            ⟨v3 , temp_ls3⟩ = E[[e3]] env temp_ls2
                        in
                            unify v2 v3 temp_ls3
                        ) LS
            in
            E[[e1]] env f_ls)
E[[e1 prim_op e2]] env =
        λLS.(let f_ls =
            fix_store (λtemp_ls.
                        let*
                            temp_ls1 = snd(E[[e1]] env temp_ls)
                        in
                            snd(E[[e2]] env temp_ls1)
                        ) LS
            in
```

$$\langle(\mathrm{fst}(\mathcal{E}[\![e_1]\!]\ env\ f\_ls))\ prim\_op\ (\mathrm{fst}(\mathcal{E}[\![e_2]\!]\ env\ f\_ls))\ ,\ f\_ls\rangle)$$

$$\mathcal{E}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\ env =$$
$$\lambda LS.(\mathbf{let}$$
$$\langle bval\ ,\ LS_1\rangle = \mathcal{E}[\![e_1]\!]\ env\ LS\ \mathbf{in}$$
$$\mathbf{if}\ bval\ \mathbf{then}\ \mathcal{E}[\![e_2]\!]\ env\ LS_1\ \mathbf{else}\ \mathcal{E}[\![e_3]\!]\ env\ LS_1)$$

6 Determinacy

It is imperative that the final answer of a program be independent of the order of evaluation of its parts. This ensures *determinacy* of programs. Determinacy of *Id* was shown in [1] by proving the *confluence* of an intermediate language *P-TAC* (with rewriting as its operational semantics) into which *Id* programs are compiled. The proof uses the sub-commutativity property of *P-TAC*.

Determinacy is achieved by the Church-Rosser property in a purely functional framework where there are no side-effects. It is also a property of singly threaded imperative code. Is determinacy preserved in *CoF*? The monotonic nature of the side-effects introduced by incorporating logical variables ensures determinacy. A few observations about logical stores are in order.

Definition 2 *Let LS_1 and LS_2 be two logical stores. LS_1 is said to be $\leq$ than LS_2 iff*

$$\forall x \in Vars\ \ LS_1(x) \subseteq LS_2(x).$$

The logical stores form a complete lattice under the $\leq$ ordering with the completely unconstrained store as the bottom element. All inconsistent (or over-constrained) stores are identified as one and interpreted as $\top$. Any computation takes a program and an initial logical store and returns a term in *normal form* and a refined logical store. Our claim is that all orders of evaluation of the program lead to the same final logical store. Intuitively, the different evaluation orders take us through different paths in the lattice to the same final store.

Lemma 1 *If some evaluation of an expression in an initial logical store LS results in error, then all evaluations of that expression in all logical stores $\geq$ LS result in error.*

An *error* is obtained by a conflicting unification. There are two kinds of unification errors.

- A call to *unify* that results in *error* while trying to refine the value (by adding to the constraint set) of a variable. The error is a result of overconstraining the variable. The variable must have some constraints already operating on it by some prior unification operation. Changing the order of the unify operations will still over-constrain the variable and thus result in *error*.
- A call to *unify* of the form

$$unify\ \langle x_1,\ preds_1\rangle\ \langle x_2,\ preds_2\rangle\ LS$$

where (at least) one of $preds_1$ and $preds_2$ is a function and the other is not an unbound variable. Such a call would result in *error* independent of the order of evaluation.

The *call by need* evaluation strategy achieves lazy evaluation. Any expression that is evaluated is needed and contributes to the final answer. Laziness thus ensures that the same expressions will be evaluated in any evaluation order. Since logical variables are singly-bound, the same top level answer will be produced in all evaluation orders. Also, the final store (if the computation is error-free) will be the same.

7 Definiteness

Definiteness is the property that ensures that any results obtained during program execution are guaranteed not to have involved any erroneous unifications. This follows immediately from the fact that the **assuming** construct reports its result only when the corresponding unifications have refined the store fully, and a consistent final logical store has been obtained.

8 Example

```
                    let z = ubv() in
hd ((1 assuming z == 1) ::  (nil assuming z == 2))
```

The denotation for the program is:

$$\mathcal{E}[\![\texttt{let...}]\!]\ \bot\ \bot$$

In anticipation of the environments and logical stores required we define:

$$\begin{array}{lcl}
env_0 & = & \bot \\
ls_0 & = & \bot \\
env_1 & = & \bot[z \mapsto lv_1] \\
ls_1 & = & \bot[lv_1 \mapsto InSuspension(\mathcal{E}[\![\texttt{ubv()}]\!]\ env_1)] \\
ls_2 & = & ls_1[v_1 \mapsto InSuspension(\mathcal{E}[\![\texttt{1 assuming z == 1}]\!]\ env_1),\ v_2 \mapsto ....] \\
ls_3 & = & ls_2[lv_1 \mapsto 1] \\
ls_4 & = & ls_3[v_1 \mapsto 1]
\end{array}$$

The reduction steps are:

$\mathcal{E}[\![$**let z = ubv() in ...**$]\!]\ \bot\ \bot$
$\mathcal{E}[\![$**hd((1 assuming z == 1) :: (nil assuming z == 2))**$]\!]\ env_1\ ls_1$
let $\langle v\ ,\ LS_1\rangle = \mathcal{E}[\![$**(1 assuming z == 1) :: (nil assuming z == 2)**$]\!]\ env_1\ ls_1$ in **if** v = ...
let $\langle v\ ,\ LS_1\rangle = \langle pair(v_1, v_2)\ ,\ ls_2\rangle$ in **if** v = ...
$\mathcal{E}[\![v_1]\!]\ env_1\ ls_2$ condition succeeds
lookup_LS_and_force $v_1\ ls_2$
let $\langle Z_1\ ,\ ls_3\rangle = \mathcal{E}[\![$**1 assuming z == 1**$]\!]\ env_1\ ls_2$ **in** $\langle\langle v_1\ ,\ Z_1\rangle\ ,\ ls_3[v_1 \mapsto Z_1]\rangle$
After evaluating $Z_1 = 1$, $ls_3 = ls_2[z \mapsto 1]$ and we get
$\langle\langle v_1\ ,\ 1\rangle\ ,\ ls_4\rangle$

9 Conclusion

We shown that logical variables can be neatly incorporated into functional languages without resulting in a semantic ghoulash. The use of information systems and

a global *logical* store elegantly captures the notion of programming with constraints. It is widely agreed that a clean denotational semantics not only helps in validating implementations but also facilitates program analysis techniques such as abstract interpretation. An implementation of *CoF* on a shared memory multiprocessor is in progress. We hope to use the semantics presented above for abstract interpretation to efficiently manage the parallelism in *CoF* programs.

References

1. Zena Ariola, Arvind. *P-TAC: A Parallel Intermediate Language* Proceedings of FPCA Conference, 1989, London, UK.

2. Arvind, Rishiyur S. Nikhil, Keshav Pingali. *I-structures: Data Structures for Parallel Computing* ACM TOPLAS, Volume 11 Number 4, October 1989.

3. Doug DeGroot, Gary Lindstrom. *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, NJ, 1986.

4. Michael J. Gordon. *The Denotation Description of Programming Languages*, Springer Verlag, 1979.

5. Lal George, Surya Mantha, Gary Lindstrom. *CoF: A Language for Multiprocessors*, Unpublished Report, submitted for publication, Nov. 1989.

6. Radha Jagadeesan, Prakash Panangaden, Keshav Pingali. *A Fully Abstract Semantics for a Functional Language with Logical Variables*, Proceedings of the LICS conference, 1989.

7. Mark B. Josephs. *The Semantics of Lazy Functional Languages*, Theoretical Computer Science, 68, 1989.

8. Richard Kieburtz. *Semantics of a Functions+Logic Language*, Oregon Graduate Center, 1986.

9. Richard Kieburtz. *Functions + Logic in theory and practice*, Oregon Graduate Center, 1987.

10. Gary Lindstrom. *Functional Programming and the Logical Variable*, ACM POPL, New Orleans, 1985.

11. Gary Lindstrom, Goran Bage. *Committed Choice Functional Programming* Proceedings of the FGCS conference, Tokyo, November 1988.

12. Gary Lindstrom. *Static Analysis of Functional Programs with Logical Variables*, International Workshop on Programming Language Implementation and Logic Programming, Orleans, France 1988.

13. Gary Lindstrom, Jan Maluszynski, Takeshi Ogi. *Using Types to Interface Functional and Logic Programming*, Unpublished Report, submitted for publication, November 1989.

14. R.S. Nikhil, K. Pingali, Arvind. *Id Nouveau*, Technical Report, Computation Structures Group Memo 265, MIT LCS 1986.

15. Keshav Pingali. *Lazy Evaluation and the Logical Variable*, Proc. of Inst. on Declarative Programming, U of Texas, August, 1987.

16. Uday Reddy. *On the Relationship between Functional and Logic Languages*, in *Logic Programming: Functions, Relations and Equations*, Prentice Hall, 1986.

17. Vijay Saraswat. *Constraint Logic Programming Languages*, PhD. Thesis, CMU, 1989.

18. David Schmidt. *Detecting Global Variables in Denotational Specifications*, ACM TOPLAS, Vol. 7, No. 2, 1985.

19. David Schmidt. *Denotational Semantics: A methodology for language development*, Allyn and Bacon, Inc., 1986.

20. Dana Scott. *Domains for Denotational Semantics*, ICALP 1982.

21. Joseph Stoy. *Denotational Semantics : The Scott-Strachey approach to programming language semantics*, MIT Press, Cambridge, Mass. 1978.

22. Satish Thatte. *Towards a Semantic Theory for Equational Programming Languages*, Proceedings of ACM Lisp and Functional Programming, 1986.

23. G. Winskel, K. G. Larsen. *Using Information Systems to Solve Recursive Domain Equations Effectively*, Semantics of Data Types, International Symposium, France 1984.