Implementing Functional Logic Languages Using Multiple Threads and Stores*

Andrew Tolmach
apt@cs.pdx.edu

Sergio Antoy
antoy@cs.pdx.edu

Marius Nita
marius@cs.pdx.edu

Dept. of Computer Science, Portland State University
P.O.Box 751, Portland, OR 97201

Abstract

Recent functional logic languages such as Curry and Toy combine lazy functional programming with logic programming features including logic variables, non-determinism, unification, narrowing, fair search, concurrency, and residuation. In this paper, we show how to extend a conventional interpreter for a lazy functional language to handle these features by adding support for reference cells, process-like and thread-like concurrency mechanisms, and a novel form of multi-versioned store. Our interpretation scheme is practical, and can be easily extended to perform compilation. The language specified by our interpreter is designed so that programs are deterministic in a novel and useful sense.

Categories and Subject Descriptors: D.3.2. **[Programming Languages]**: Language Classifications—*Multiparadigm languages, Applicative (functional) languages, Constraint and logic languages*; D.3.4. **[Programming Languages]**: Language Processors—*Interpreters, Run-time environments*

General Terms: Languages, Design

Keywords: Functional logic languages, narrowing, residuation, multi-versioned stores

1 Introduction

Functional logic programming (FLP) integrates the most important features of functional and logic programming within a single programming model (see Hanus [8] for a detailed survey). Thus, functional logic languages provide pattern matching in the definition of functions and predicates as well as the use of logical variables in expressions. The latter feature requires some built-in search principle in order to guess the appropriate instantiations of logical variables. There are a number of languages supporting FLP in this broad sense, including Curry [13], Escher [19], Life [1], Mercury [30], Oz [29], and Toy [20], among others.

Narrowing is one promising basis for a functional logic language. It is a combination of term reduction as in functional programming and (non-deterministic) variable instantiation as in logic programming. It exploits the presence of functions without transforming them into predicates, which yields a more efficient operational behavior [7, 9]. As a simple example, consider this program (in Curry syntax, as are all the examples in this section).

```
data Color = Red | Yellow | Blue
           | Orange | Violet | Green

mix Red Blue = Violet
mix Yellow Blue = Green
mix Yellow Red = Orange

a1,a2,a3 :: Color
a1 = mix Red Blue
a2 = mix Yellow x where x free
a3 | (mix Yellow x =:= Green) = x
          where x free
```

To compute `a1`, `mix` is used as an ordinary function, and returns `Violet`. In `a2`, `free` declares `x` as a logic variable, so the call to `mix` narrows over the latter two clauses and produces *two* answers corresponding to the colors that result when `Yellow` is mixed with another primary (namely `Green` and `Orange`). In `a3`, narrowing produces the same results from `mix`, but they are then filtered by the equality constraint (`=:=`) appearing in the guard, and the return value is `x` itself—which here will be `Blue`, the color that produces `Green` when mixed with `Yellow`.

It is well known that narrowing is an evaluation mechanism that enjoys soundness and completeness in the sense of functional and logic programming, i.e., all computed solutions are correct and all correct solutions are computed. But narrowing is not able to deal with *primitive* (or *external*) functions. Therefore several authors (e.g., Ait-Kaci [1], Lloyd [19]) have proposed alternative evaluation strategies based on *residuation*. The residuation principle delays a function call until the arguments are sufficiently instantiated so that the call can be deterministically reduced. To make this useful, residuation-based languages also support concurrent evaluation in order to deal with suspended computations.

Hanus [11] proposes a seamless combination of needed narrowing with residuation-based concurrency. This is the basis of the programming language Curry [13], an attempt to provide a standard

*This work has been supported in part by the National Science Foundation under grants CCR-0110496 and CCR-0218224.

To appear in the proceedings of the Ninth Internationa Conference on Functional Programming (ICFP 2004), September 19-21, 2005, Snowbird, Utah, USA, pages 90–102.

in the area of FLP languages. To illustrate the interplay between narrowing and concurrency, suppose we have a primitive function `hue::Color->Float` that returns a numeric equivalent for each color (e.g., its position on the color wheel measured in radians), and assume that the `+` operator on `Floats` is also primitive. The following expressions both sum hues for two colors:

```
a4,a5 :: Float
a4 = hue (mix x Blue) + hue x  where x free
a5 = hue x + hue (mix x Blue)  where x free
```

Both `a4` and `a5` should produce *two* answers, `(hue Violet + hue Red)` and `(hue Green + hue Yellow)`, corresponding to the two narrowing choices for `mix`. But since `hue` is primitive, it cannot be invoked without an instantiated value for `x`. Thus, `a4` requires the `mix` call in the left operand to `+` to be evaluated before the right operand; similarly, `a5` requires the opposite order of evaluation. In general, primitive operands need to be evaluated *concurrently* in order to guarantee that we get the same answer set independent of the order in which we write them. The evaluation strategy for the "parallel and" (`&`) primitive operator in Curry, which is used to connect simultaneous constraints, is just an instance of this general policy.

Narrowing can easily generate an infinite space of alternatives to be explored. Faithful implementations of narrowing must employ a *fair* search strategy in order to guarantee completeness, i.e., they should never get stuck indefinitely computing in one narrowing alternative when there are other alternatives still to be tried. Consider the following function to compute the color-wheel complements:

```
complement :: Color -> Color
complement Red = Green
complement Yellow = Violet
complement Blue = Orange
complement x | (complement y =:= x) = y
                        where y free
a6 :: color
a6 = complement Orange
```

Here we thought to save effort by defining only half the cases explicitly, and using recursion and narrowing for the other half. This is simple and elegant, but it is dangerous unless the language uses a fair strategy. In Curry, when two or more clauses match an argument, they are *both* explored, regardless of clause order. Since the last clause for `complement` matches *any* argument, at least one computation path for `a6` will always be infinite. If the language search strategy is not fair, clause order *is* likely to matter. For example, if conventional backtracking is used, moving the last clause first is likely to get us stuck on an infinite sequence of recursive calls. A fair strategy will guarantee that we eventually consider *all* possible paths (regardless of clause order), and so get an answer. (Of course, we would have to wait forever if we wanted to make sure there were no other answers.)

These examples illustrate that both the semantic formalization and the implementation of Curry-like languages must deal with ordinary (lazy) functional evaluation, narrowing, and residuation in an integrated framework. This is a non-trivial task, especially because functional and logic programming have rather different semantic and implementation traditions. The "official" semantics for Curry [13], largely based on work by Hanus [11, 10], is an operational semantics based on definitional trees and narrowing steps. Although fairly low-level, this semantics says nothing about sharing behavior, which has an important impact on the efficiency of lazy functional programs, and is actually observable in the presence of logic variables. Moreover, this semantics is quite different in character from typical semantic formulations for functional languages, which are often given denotational or natural ("big step") operational semantics. In more recent work [2], Albert, Hanus, Huch, Oliver, and Vidal propose a natural semantics, incorporating sharing, for the first-order functional and narrowing aspects of Curry, and a small-step semantics that also covers residuation and concurrency. In previous work [31], we extended the large-step semantics of Albert, *et al.*, to handle concurrency, addressed some problems with determinism in the presence of residuation, incorporated true higher-order functions, and cast the entire semantics in the form of a modular monadic interpreter written in Haskell. However, our large-step semantics is not suitable for use as an implementation. The small-step semantics of Albert, *et al.* is more suitable, but still not very practical; for example, it treats substitution across an entire heap as a basic operation.

On the implementation front, there have been a number of practical systems, include Pakcs [12], and the Münster Curry compiler [21]. These systems are fairly full-featured and deliver adequate performance; their main drawback is that they use backtracking as a search strategy, and so are not complete. A more experimental, Java-based system that does use a fair strategy for narrowing was reported in a previous paper [3]; that work did not address concurrency for arguments to primitives (except `&`).

The major contributions of the present paper are these:

- We describe a simple and parsimonious `CCore` language for functional logic programming that supports all the features mentioned above. Programs in Curry-like source languages can be desugared into `CCore`.
- We show how a conventional functional language abstract-machine interpreter can be extended in easy stages to support FLP with fair search, by using reference cells, process-like and thread-like concurrency mechanisms, and variant heaps. Viewed as a semantics, our interpreter strongly resembles that of Albert, *et al.* [2], but our presentation is considerably more detailed, making it a good basis for a practical interpreter.
- To support narrowing, we define a novel kind of *variant heaps* that support a Unix-like `fork` operation; these may be of independent interest for applications requiring an analogue to `callcc` for stores.

The remainder of the paper is organized as follows. Section 2 gives a detailed description of our `CCore` language. Section 3 describes the translation phases that convert `CCore` to `ACore`, the form we actually interpret. Section 4 describes the definitional interpreter in detail. Section 5 discusses practical implementation issues, in particular the efficient implementation of variant heaps. Section 6 discussed ongoing and future work, and concludes. We assume reading knowledge of Standard ML [24], which is used throughout to describe the interpreter and its data structures.

2 CCore Language

Our starting point is an expression-based `CCore` language. This is essentially a call-by-need, higher-order functional language with algebraic data types, extended with logic variables, flexible case expressions, (shallow) unification, and a `seq` operator to force evaluation. We assume (but do not describe) a pre-processing step that desugars richer source languages such as Curry into `CCore`. Our primary motivation in designing `CCore` has been to keep the basic facilities of the language as simple as possible, so they will have

```
structure CCore =
struct
    type var = string
    type dcname = string
    datatype prim = ...
    type pattern = dcname * var list
    datatype flexibility = FLEXIBLE | RIGID

    datatype exp =
        Var of var
      | Int of int
      | Abs of var * exp
      | App of exp * exp
      | Capp of dcname * exp list
      | Primapp of prim * exp list
      | Case of flexibility * exp *
                (pattern * exp) list
      | Letrec of (var * exp) list * exp
      | Let of var * exp * exp
      | Seq of exp
      | Free
      | Unify of exp * exp
end
```

x	(variable name)
c	(data constructor name)
p	(primitive operator)
	(case pattern)
f | r	(case flexibility)
e :=	
x	(variable)
i	(integer)
$\lambda x.e$	(abstraction)
$e_1\ e_2$	(application)
$c(e_1,\ldots,e_n)$	(constructor application)
$p(e_1,\ldots,e_n)$	(primitive application)
case$^{\mathtt{f}\mid\mathtt{r}}$ e of { $c(x_1,\ldots,x_n)$ => e }	(case: flexible or rigid)
letrec { x = e } in e	(recursive binding)
let x = e_1 in e_2	(local binding)
seq e	(force evaluation)
free	(logic variable)
unify e_1 e_2	(shallow unification)

Figure 1. CCore Language Abstract and Concrete Syntax. Curly braces delimit zero or more repetitions.

obvious, efficient implementations; more complicated operations can be encoded as macros or subroutines in `CCore` itself. Figure 1 defines the abstract syntax for the language, and a corresponding concrete syntax used in examples.

2.1 Types and Values

`CCore` is not explicitly typed, but we assume throughout that we are dealing with typable expressions. In particular, programs are to be interpreted in the context of a set of algebraic datatype declarations, fixing the names and arities of the data constructors, which can be specified in the usual Haskell or ML style. At a minimum, we assume these algebraic types:

```
data Bool = False | True
data Success = Success
data List a = Nil | Cons (a, List a)
```

`Success` is used as the type of constraints, which either succeed (but without returning any useful value) or fail (without returning a value at all). For simplicity, the only primitive type used in this paper is `Int`, which has the usual literals; other primitive types along the same lines could easily be added. There are also the usual arrow (function) types.

In addition to the ordinary values, each non-arrow type includes an endless supply of distinct *logic variables*. These are initially uninstantiated; in the course of execution they may become instantiated to a value or to another logic variable of the same type. In the presence of logic variables, it is useful to distinguish between several different "degrees" of evaluation:

- *Head-normal form (HNF)* means the top-level constructor is known (for algebraic types), or the integer value is known (for the `Int` type), or the function closure is known (for an arrow type), or the value is an uninstantiated logic variable.
- *Normal form (NF)* means in HNF, and all subterms are also in NF.
- *Constructor-head-normal form (CHNF)* means in HNF, but *not* an uninstantiated logic variable.
- *Constructor-normal form (CNF)* means in CHNF, and all subcomponents are also in CNF.

The basic evaluation mechanism provided by our interpreter computes HNF values; stronger degrees of normalization can be encoded within `CCore` itself (see Section 2.7).

2.2 Expressions

The `CCore` expression language is very similar to that of Core Haskell [26], with the addition of logic variables (`free`), flexible `case` expressions, `unify`, and `seq`. Function abstractions and applications have exactly one argument; multiple-argument functions can be coded as nested abstractions. Data constructors (c) and primitive operators (p) have fixed arity $ar \geq 0$. All primitive and constructor applications and patterns (within `Case` expressions) must be saturated; unapplied or partially applied constructors can be expressed by η-expanding them. By default, function and constructor arguments are evaluated lazily, but eager evaluation can be forced by wrapping an argument in `seq`.

Primitive arguments are evaluated eagerly to CHNF. The arguments to a primitive or constructor are (conceptually) evaluated in parallel; this is important if evaluation of one or more arguments *residuates* (Section 2.5). A minimal set of primitive operators includes:

```
== :: Int × Int → Bool
+,-,*,etc. :: Int × Int → Int
chnf :: ∀a.a → a
pand :: Success × Success → Success
```

The last two operators are used only for their side-effects: `chnf` is just the identify function, which forces its argument to a CHNF value; `pand` forces parallel evaluation of its arguments (which are typically constraints) to CHNF but then ignores them.

`Let` introduces a local binding; it is evaluated lazily unless the right-hand-side is wrapped in a `seq`. Although, as we shall see, β-reduction is not generally valid for `CCore`, we do have `(`$\lambda x.e_1$`)` $e_2 \equiv$ `let` x `=` e_2 `in` e_1. `Letrec` introduces a set of (potentially) mutually recursive bindings, always evaluated lazily (`seq`s are not permitted on the right-hand sides).

`Case` expressions analyze values of algebraic type. Patterns are "shallow;" they consist of a data constructor name c and a corresponding list of variables. More complicated pattern matching can be expressed using nested `case` expressions. The patterns for a single `case` are assumed to be mutually exclusive, but not necessarily exhaustive; a computation that attempts to dispatch to a missing case arm *fails*. Evaluating a case causes the *scrutinee* (the expression being "cased over") to be evaluated to HNF. Each case is statically marked by the programmer as either *flexible* or *rigid*. This controls what happens when the scrutinee evaluates to an uninstantiated logic variable: flexible cases narrow (Section 2.4); rigid cases residuate (Section 2.5).

A `free` expression evaluates to a fresh, anonymous, uninstantiated logic variable. Logic variables can be instantiated either by being used as the scrutinee of a flexible `case`, or via evaluation of a `unify` (described in Section 2.7). Instantiation can occur at most once; the instantiating value is always in HNF. Once instantiated, any use of the variable transparently fetches the instantiating value.

2.3 Programs, Answers, and Determinism

A program is just a closed top-level expression. Each program evaluates to an unordered multiset of *answers*, each of which is an HNF value. (If NF values are desired, the program expression can be wrapped in a normalizing function; see Section 2.7.) In general, there may be one answer corresponding to each combination of narrowing choices made during the evaluation of the program; however, choices that lead to failure don't contribute an answer. The only possible sources of failure are a missing `case` arm, a failed unification, or *floundering* (Section 2.5).

Determinism Property. Although we think of individual expressions as being non-deterministic, we would like `CCore` programs as a whole to be deterministic, in the specific sense that they always produce the same multiset of answers (neglecting order), no matter how the implementation schedules evaluation of narrowing alternatives and parallel evaluation of arguments, provided that the schedule is starvation-free.

Maintaining the Determinism Property has been a key goal in our design of `CCore`, and has directed a number of important design decisions. However, we do not give a formal proof that it holds (indeed, the property would require a more formal statement before a proof could be attempted).

2.4 Narrowing and Fairness

Narrowing in `CCore` occurs when evaluating a flexible `case`, e.g.

```
case^f e of
  c_1 (x_{1,1}, ..., x_{1,ar(c_1)}) => e_1
  ...
  c_n (x_{n,1}, ..., x_{n,ar(c_n)}) => e_n
```

if the HNF v of e is an uninstantiated logic variable. At such a point, evaluation splits into n parallel alternative evaluations, one corresponding to each `case` arm. In the i-th alternative, v is instantiated to $c_i(w_1, \ldots, w_{ar(c_i)})$, where the w values are fresh uninstantiated logic variables; evaluation then proceeds by evaluating e_i and continuing with the computation that demanded the value of the `case` in the first place. Since the result of the program can depend on the value of v (and of the `case`), each of the n alternatives may contribute a value to the overall multiset of program answers, or may terminate in failure. Note that a simple non-deterministic choice operator, which we call `amb` (after McCarthy [22]), can easily be defined using a flexible case:

```
amb e_1 e_2 ≡
  case^f free of
   True => e_1
   False => e_2
```

Since flexible `case`s can be nested, each alternative may split into further sub-alternatives, forming an "or-parallel" tree for the program. Each leaf of the tree potentially contributes to the program answer, so all branches of the tree must be fully explored. The branches are completely independent, so the strategy used to explore them is unimportant in principle, provided that it copes with non-terminating branches. In other words, the strategy must be *fair*; it should guarantee that every answer will ultimately be computed. Note that most implementations of (functional) logic languages use simple backtracking, i.e., depth-first exploration, but this can get stuck in a non-terminating branch. To avoid this problem, we use multiple concurrent or-parallel threads (or simply "or-threads") which are scheduled in a breadth-first fashion. Since each or-thread is logically independent from the others, it must have its own heap; it thus resembles a *process* in a conventional operating system.

2.5 Residuation and Concurrency

Sometimes evaluation is blocked because of an uninstantiated logic variable. For example, the `+` primitive operation cannot proceed until both operands have been evaluated to integers. More generally, all operands to primitives and scrutinees of `case`$^{\mathtt{r}}$ expressions must evaluate to CHNF. What should happen when evaluation is blocked? The simplest idea might be to have blocked computations fail. But this would lead to very unattractive semantics. Recall expressions `a4` and `a5` from Section 1. Using left-to-right evaluation of the `+` operands, `a4` produces answers, but `a5` fails; using right-to-left evaluation, the converse is true. But there is nothing about the (intuitive) semantics of `+` that justifies choosing one order over the other, so surely either both expressions should fail or (more usefully) both should succeed.

More generally, it might be the case that *neither* fixed order for operand evaluation works, but that an *interleaving* of the two evaluations does. Consider

```
sum ≡ let y = free in
              let z = free in
                 + (let dummy = unify y 1 in z,
                    let dummy = unify z 1 in y)
```

Here neither operand to `+` can be fully evaluated until evaluation of the other has started, but there is an interleaving of read and write operations on the variables that produces an answer. Thus we are led naturally to the idea that operands to primitives should be evaluated *concurrently*. The primitive application produces an answer if *any* interleaving of the atomic variable read and write operations of the operand evaluations leads to an answer. Otherwise, the computation fails by *floundering*.

More operationally, we can think of each operand evaluation as a separate and-parallel thread (or just "and-thread"). Unlike or-threads, *all* and-threads must produce an answer, and they all *share* a single heap. Thus, they resemble "lightweight" (intra-process) threads in a conventional operating system. If any and-thread blocks (*residuates*) on an uninstantiated variable, evaluation of the other threads proceeds; with luck, one of these threads will instantiate the required variable, allowing evaluation of the initial thread to proceed. If all the and-threads residuate simultaneously, the computation flounders; operationally, this corresponds to thread deadlock.

The "parallel and" operator in Curry, which is ordinarily used to connect equational constraints, can now be seen as just a special case of a primitive operator, which (just) requires its operands to be evaluated successfully (i.e., not to fail). Parallel-and supports useful programming idioms originally developed in languages like Concurrent Prolog [28]. But it is interesting to note that even if we are not interested in using these idioms, the desire to maintain the Determinism Property drives us to evaluate primitive parameters concurrently: supporting concurrency seems almost unavoidable in the presence of primitive operators that cannot narrow.

Parallel evaluations can be nested, e.g., when one primitive operation consumes the result of another. In such situations, the inner computation should not flounder as long as there is a possibility that the outer computation might instantiate the necessary variables. Operationally, this means that all the and-threads from both inner and outer computations should be treated as a single pool; floundering occurs only when the entire pool is deadlocked.

In a general program, or-threads and and-threads coexist; the overall state of the computation is always an or-parallel tree whose leaves are and-parallel sets (perhaps just singleton sets if there is no concurrency). When a narrowing step occurs in any and-thread, the complete set of and-threads is replicated in each of the or-threads generated by the narrowing alternatives. As an example, consider the expression

```
let x = free in let y = free in
+ (case^f x of True => 0 | False => 1,
   case^f y of True => 2 | False => 4)
```

Figure 2 illustrates how the computation state unfolds as this expression is evaluated. Initially (a), two and-threads are created for the two arguments to `+`. Assume that the left argument is evaluated first. After the left narrowing step (b), there are two or-threads, each with its own pair of and-threads. After the right narrowing step (c), there are four or-threads and eight and-threads; at this point, the `+` operations can all evaluate, yielding four answers (2,3,4,5).

2.6 Logic Variables and Effects

Creation and instantiation of logic variables are imperative effects. They may therefore seem to be a poor ingredient to add to a lazy language. But the resulting language is reasonable to program in (and has the Determinism Property) because no program answer can depend on the order in which variable instantiations occur. Intuitively, this follows from the facts that no variable can be instantiated more than once, and that it is impossible to test whether a variable has been instantiated. (This last property is rather delicate; for example, small changes in the treatment of unification could break it; see Section 2.7).

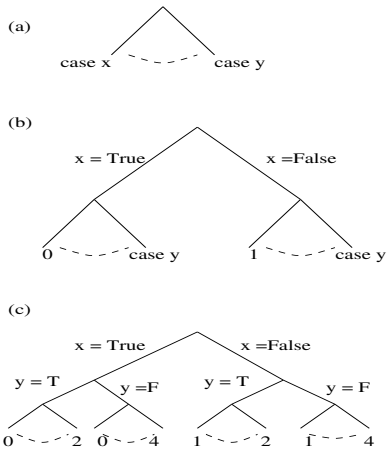

Figure 2. Threads at various stages of evaluation. Dashed lines connect groups of and-threads.

However, logic variables *do* allow us to observe the sharing behavior of the language. For example, given

```
coin ≡ amb 0 1
coin1 ≡ (λx. +(x,x)) coin
coin2 ≡ +(coin,coin)
```

`coin1` produces the answer multiset $\{0,2\}$, whereas `coin2` produces $\{0,1,1,2\}$. This example illustrates why β-equivalence is not generally valid for `CCore`.

2.7 Full Unification and Normalization

The design of `CCore` deliberately excludes expression forms that would require ad-hoc polymorphic implementations. In particular, unification and full normalization both require traversal of arbitrary constructed values, so neither is provided as a built-in expression form. Instead, we "unbundle" these tasks: `CCore` provides only "shallow" operations for unification (`unify`) and forcing evaluation (`seq`), but these can be used to build type-indexed families of full equality and normalization functions in `CCore` itself.

Unification and equality constraints. The shallow `unify` operator evaluates its arguments in parallel to HNF, and then attempts to unify the heads of the resulting values. If either argument evaluates to a logic variable, the unification succeeds (instantiating the variable), and the `unify` expression evaluates to `True`, indicating that there is no need to inspect sub-terms of either argument. Otherwise, the top-level constructors of the two values are compared—but not the subterms. (In this context, integers are treated like nullary constructors.) If the heads are not equal, the entire computation fails without returning a value; if they are equal, the `unify` expression evaluates to `False`, indicating that shallow unification succeeded but any sub-terms still need to be compared.

Using `unify` as a building block, we can implement a "deep" structural equality constraint operator somewhat similar to Curry's operation

```
=:= : ∀a.a -> a -> Success
```

as a type-indexed family of `CCore` functions `eqsInt`, `eqsBool`, `eqsList`, etc. The basic idea is to use the `unify` expression to unify integers and head constructors and to construct explicit `CCore`

functions to traverse and unify subterms. For recursive types, the traversal functions will also be recursive. The functions for parameterized types are higher-order; they get additional arguments representing the functions corresponding to the type parameters. Some examples:

```
eqsInt e1 e2 ≡ if (unify e1 e2) Success Success

eqsList eqsA e1 e2 ≡
  letrec eqs =
           λx1.λx2.
              if (unify x1 x2)
                 Success
                 (case^r x1 of
                    Nil => Success
                  | Cons(h1,t1) =>
                      case^r x2 of
                        Cons(h2,t2) =>
                          pand(eqsA h1 h2,
                               eqs t1 t2))
  in eqs e1 e2

eqsIntList = eqsList eqsInt
eqsIntListList = eqsList eqsIntList
...
```

A Curry-like front end could derive all the equality functions automatically from the algebraic type definitions, and propagate them dynamically using dictionary passing.

This version of equality differs from Curry's in that it is *non-strict*, i.e., a logic variable always unifies successfully against *any* other value (even $\perp$). Curry's `=:=` only returns `Success` if both arguments reach (unifiable) NFs, and it also performs an "occurs check." We prefer the semantics of our operator, which seems more consistent with ordinary lazy functional programming in that it permits definition of infinite (circular) data structures via unfication. There is another, more subtle, difficulty with Curry's operator: to implement it in an "unbundled" way seems to require an expression form that detects when two values are both uninstantiated logic variables. Including this in `CCore` would violate the Determinism Property.

Normalization. Similarly, to force evaluation of an expression to full CNF, we can use (another) type-indexed family of `CCore` functions. These functions make essential use of the `seq` operator and the `chnf` primitive. Examples:

```
normInt e ≡ chnf(e)

normList normA e ≡
  letrec norm =
     λx.case^r x of
          Nil => x
        | Cons(h,t) =>
            Cons(seq (normA h),
                 seq (norm t))
  in norm e

normIntList ≡ normList normInt
normIntListList ≡ normList normIntList
```

Note that the arguments to constructors (e.g., `Cons`) are evaluated in parallel, which may be essential to reach a normal form.

3 Translation Phases

Although it would be possible to interpret `CCore` directly, it would require fairly heavy machinery within the interpreter to handle laziness and multithreading. So instead, we pass it through two preliminary translations, first to a call-by-value language `SCore`, and then into an A-normal-form variant language `ACore`. `ACore` expressions can be evaluated by an interpreter written in continuation-passing style, which makes multi-threading straightforward.

3.1 Making Thunks Explicit

We implement call-by-need by converting `CCore` programs into a call-by-value language `SCore` that has explicit support for creating and resolving thunks. The syntax of `Score` is just a small extension of `CCore`:

```
structure Score =
 datatype exp = ...
                  ...just like CCore...
              | Delay of exp
              | Force of exp
end
```

But `Score` (implicitly) uses eager evaluation semantics. The use of thunks and the `CCore` to `Score` conversion are largely standard [17], so we omit the details here.

In principle, an advantage of using explicit `delay` and `force` is that we can express the results of various possible optimizations based on analysis of CCore programs. For example, we could use strictness analysis to remove `delay`s (and the corresponding `force`s) on arguments that are guaranteed to be evaluated. We can also remove provably redundant `force`s, omit `delay`s for "cheap" expressions (i.e., expressions whose evaluation guaranteed to terminate quickly) [5], etc. We have not implemented any such optimizations. `SCore` could also be used directly as the desugaring target of a call-by-value source language.

3.2 A-normal Form

`Acore` is a variant of A-normal form [6] adapted to our language. It is essentially a subset of `Score`, stratified into several kinds of expressions to support a continuation-passing-style interpreter.A class of "trivial" expressions that require no evaluation is explicitly called out as a separate syntactic class (`vexp`). All arguments to `App`, `Primapp`, `Capp`, `Case` etc., must be `vexp`s; hence any non-trivial arguments in `Score` programs must be named. Order of evaluation is made explicit via `Let` and `Letpar` (parallel) bindings. Only a restricted set of expressions (`sexp`) can be bound in `Let` expressions, and only `vexp`'s can be bound in `Letrec` expressions, which guarantees that the right-hand sides of recursive bindings can be evaluated without requiring the values of any left-hand sides.

`Acore` also has two new expression forms. `Letpar` performs a set of two or more bindings in parallel; it is used to evaluate the arguments to `Primapp`, `Capp`, and `Unify` expressions. (Since parallel evaluation can have substantial overhead, we use sequential evaluation instead when a simple analysis shows that at most one of the expressions instantiates a logic variable.) `CheckInstantiated` blocks evaluation if its argument is an uninstantiated logic variable; it is used to ensure that the arguments to `Primapp`s are in CHNF. Constructors `SofV` and `EofS` serve to inject `vexp`s into `sexp`s and `sexp`s into `exp`s, respectively.

```
datatype vexp =
  Var of var
| Int of int
| Abs of var * exp
| Delay of exp

and sexp =
  SofV of vexp
| App of vexp * vexp
| Primapp of prim * vexp list
| Capp of dcname * vexp list
| Unify of vexp * vexp
| Force of vexp
| CheckInstantiated of vexp
| Free

and exp =
  EofS of sexp
| Case of flexibility * vexp *
        (pattern * exp) list
| Let of var * sexp * exp
| Letrec of (var * vexp) list * exp
| Letpar of (var * exp) list * exp
```

Figure 3. Acore Abstract Syntax

The translation of `Score` into `Acore` is also fairly standard [6], so again we omit the details. Since `Acore.Case` expressions can only appear in tail-position within functions, it is often necessary for the translation to create new functions representing join points following a `case`.

4 Definitional Interpreter

The interpreter is fairly complex; to ease understanding, we present it in several stages. Section 4.1 shows the full code for interpreting just the functional substrate of `ACore`. Section 4.2 describes how logic variables, narrowing, and "or-threads" can be added. Section 4.3 adds support for concurrent "and-threads" and residuation. The code relies on several ADTs, whose signatures are given in Figure 4.

4.1 Functional Substrate

Figure 5 shows the interpreter code for the basic functional substrate. (A few standard features such as `letrec` are omitted to save space.) This is not novel; it is essentially a $C_\alpha EK$ machine [6], with support added for thunks, and with explicit attention to heaps. The basic interpreter function, `interp`, evaluates a top-level expression `exp`. If the evaluation is successful, the result value is stored into the global list `answers`. The real work of interpretation is done by functions `evalE` and `evalS`, which evaluate `exp`s and `sexp`s (respectively) to HNF values. These functions are written in a form of continuation-passing style; they are explicitly parameterized by an *environment* (`env`), which is a mapping from program variables to `value`s, and by a stack of *continuation frames* (`kont`), which tells what to do with the result value. Evaluation is also implicitly parameterized by the *heap*, which is a mutable map from heap pointers (`hptr`) to `value`s. The heap supports functions `new`, `alloc`, `fetch`, and `update`, with the obvious semantics, and a further operator, `fork`, which is described in Section 4.2. Only mutable values (e.g., thunks) are allocated in the explicit `heap`; immutable values (e.g., records) live in the implicit (ML) heap.

```
signature E = sig
  type 'a env
  val empty : 'a env
  val extend : 'a env * (string * 'a) list ->
               'a env
  val lookup : 'a env * string -> 'a
end

signature H = sig
  type 'a hptr
  type 'a heap
  val new : unit -> 'a heap
  val alloc : 'a heap -> 'a -> 'a hptr
  val fetch : 'a heap -> 'a hptr -> 'a
  val update : 'a heap -> 'a hptr * 'a -> unit
  val fork : 'a heap -> 'a heap
end

signature Q = sig
  type 'a q
  val empty : 'a q
  val enqueueRear : 'a q * 'a -> 'a q
  val enqueueFront : 'a q * 'a -> 'a q
  val dequeue : 'a q -> ('a * 'a q) option
  val dequeueIf : ('a -> bool) -> 'a q ->
                  ('a * 'a q) option
end
```

Figure 4. Signatures for Environment, Heap, and Queue ADTs.

The `value` type is fundamental. The first three value constructors correspond roughly to CHNF's of `CCore` expressions; in particular, `VClos` represents a λ-abstraction as a closure which includes the (entire) lexical environment. The remaining value constructors have to do with the mutable heap. A thunk for expression e in environment *env* is created by allocating a new heap entry containing the value `VThunk(`*env*`,`*e*`)`, say at heap pointer p, and returning `VDelay` p as the value of the `delay` expression. `VEmpty` is used to "black-hole" thunks during forcing [18, 27]; this converts some non-terminating programs into failing ones, removes some possible space leaks, and prevents certain synchronization problems in the presence of and-parallelism (see Section 4.3).

The continuation parameter to the evaluation functions tells them where to deliver the evaluation result. Using continuations allows us to make these functions completely tail-recursive, so calls to them are essentially just jumps, and no implicit control stack is needed. (This will prove to be valuable when we introduce multiple threads in the next section.)

4.2 Logic Variables and Narrowing

To add support for narrowing and unification to the interpreter, we introduce several new features: a `value` form representing logic variables; a queue of `othread`s corresponding to narrowing alternatives; and a mechanism for representing variant versions of the heap, one for each `othread`. The added and altered code is shown in Figure 6.

A `free` expression evaluates to `VLogic` p, where p is the heap pointer of a newly allocated entry in the (current) heap. To instantiate this logic variable, the interpreter updates the heap at p. Narrowing (`case`$^{\mathtt{f}}$) always causes instantiation to a CHNF, but unification

```
datatype value =
  VRecord of dcname * value list
| VClos of env * var * exp
| VInt of int
| VDelay of hptr
| VThunk of env * exp
| VEmpty
withtype
  env = value E.env
  hptr = value H.hptr

data kontframe =
     KBind of var * env * exp
   | KUpdate of hptr
type kont = kontframe list

val answers : value list ref = ref []

fun noteAnswer (w:value) : unit =
       answers := w::(!answers)
fun fail () : unit = ()

val heap : value H.heap ref = ref H.empty
fun fetch h = H.fetch (!heap) h
fun update (h,a) = H.update (!heap) (h,a)
fun alloc a = H.alloc (!heap) a

fun evalV (env:env) (v:vexp) : value =
    case v of
      Var x => E.lookup (env,x)
    | Int i => VInt i
    | Abs(x,e) => VClos(env,x,e)
    | Delay e => VDelay(alloc (VThunk(env,e)))

fun matchPattern (pes:(pattern * exp) list)
                 (c0:dcname)
                 : (var list * exp) option = ...

fun doPrimapp (p:prim,ws:value list) : value = ...

fun enterBH _ : unit = fail ()

fun interp (e:exp) : unit =
   (heap := H.new ();
    answers := [];
    evalE (E.empty,[],e))
```

```
and evalE (env:env,kont:kont,e:exp) : unit =
    case e of
      EofS s => evalS (env,kont,s)
    | Case(RIGID,v,pes) =>
       let val VRecord(c,ws) = evalV env v
       in case matchPattern pes c of
            SOME(xs',e') =>
              evalE (E.extend (env,zip (xs',ws)),
                     kont, e')
          | NONE => fail()
       end
    | Let(x,s,e) =>
       evalS (env,KBind(x,env,e)::kont,s)
    | Letrec(xvs,e) =>  ...

and evalS (env:env,kont:kont,s:sexp) : unit =
    let val return = continue kont
    in case s of
         SofV v => return (evalV env v)
       | App(vop,v) =>
          let val VClos(env',x',e') = evalV env vop
              val w = evalV(env,v)
              val env'' = E.extend (env',[(x',w)])
          in evalE (env'',kont,e')
          end
       | Force v =>
          (case evalV env v of
            VDelay h =>
             (case fetch h of
                VThunk(env',e') =>
                 (update (h,VEmpty); (* set BH *)
                  evalE (env', KUpdate h::kont,e'))
               | VEmpty => enterBH (env,kont,EofS s)
               | w => return w)
          | w => return w)
       | Primapp(p,vs) =>
          return (doPrimapp (p,map (evalV env) vs))
       | Capp (_,c,vs) =>
          return (VRecord(c,map (evalV env) vs))
    end

and continue (kont:kont) (w:value) : unit =
    case kont of
      [] => noteAnswer w
    | KBind(x,env',e')::kont' =>
        evalE (E.extend (env',[(x,w)]), kont',e')
    | KUpdate h'::kont' =>
        (update (h',w); continue kont' w)
```

Figure 5. Interpreting Functional Subset of ACore.

may cause instantiation to another `VLogic` value; auxiliary function `chaseLogic` (not shown) chases down chains of unified variables until a CHNF or uninstantiated logic variable is found.

Variant Heaps. Each narrowing variant requires an independent heap to store values that depend on the narrowing choice. The key idea behind our narrowing implementation is that different threads "see" different instantiations for a narrowed logic variable, even though they all use the same heap pointer to refer to that variable. Similarly, different threads may "see" different values for updated thunks, since the value of the thunk may depend on a narrowed logic variable. Because logic variables are "first-class" entities that can be used interchangeably with ordinary values, it is not possible (in general) to determine statically which thunks depend on variables in this way, so the interpreter assumes that every thunk might.

To support this per-thread state, the interpreter implements *variant heaps*, which differ from each other on some, but usually not many, entries. A new variant heap is created by applying the `fork` operation to an existing heap. (`fork` *h*) produces an independent copy of *h* that can subsequently be changed (by `update` or `alloc` operations) without affecting *h*; its action on heaps is similar to the action of Unix `fork` on address spaces. Within the interpreter, all heap operations are interpreted with respect to the current variant stored in the global reference `heap`. Efficient implementation of `fork` is discussed in Section 5.1.

```
datatype value = ...
               | VLogic of hptr

fun mkLogic _ = VLogic (alloc VEmpty)

fun chaseLogic (w:value) : value  = ...

val initialSlice : int = ...
val remainingSlice : int ref = ref initialSlice

type computation = env * kont * exp
type othread = value H.heap * computation

val othreads : othread Q.q ref = ref Q.empty

fun onext () : unit =
   case (Q.dequeue (!othreads)) of
     SOME((heap',comp'),rest) =>
       (othreads := rest;
        remainingSlice := initialSlice;
        heap := heap';
        evalE comp')
   | NONE => ()  (* done! *)

fun ospawn (comp:computation) : unit =
   othreads := Q.enqueueFront (!othreads,
                               (!heap,comp))

fun oyield (comp:computation): unit  =
   (othreads := Q.enqueueRear (!othreads,
                               (!heap,comp));
    onext ())

fun fail () : unit = onext ()
fun noteAnswer (w:value) : unit =
     (answers := w::(!answers); onext ())

fun narrow (heap0:heap,env:env,kont:kont,h0:hptr)
           ((c,xs):pattern,e:exp) : unit =
   (heap := H.fork heap0;
    let val ws = map mkLogic xs
        val env' = E.extend (env,zip (xs,ws))
    in update (h0,VRecord(c,ws));
       ospawn (env',kont,e)
    end)
```

```
fun interp (e:exp) : unit =
   (othreads := Q.empty;
    remainingSlice := initialSlice;
    ...)

and evalE (env:env,kont:kont,e:exp) : unit =
   case e of
     ...
   | Case(flx,v,pes) =>
       (case chaseLogic (evalV (env,v)) of
          VRecord(c,ws) => ...
        | VLogic h0 =>
           (case flx of
              FLEXIBLE =>
                (app (narrow (!heap,env,kont,h0))
                     pes;
                 onext ())
            | RIGID => fail ()))

and evalS (env:env,kont:kont,s:sexp) : unit =
   (remainingSlice := !remainingSlice - 1;
    if !remainingSlice = 0 then
      oyield (env,kont,EofS s)
    else
     let val return = ...
     in case e of
          ...
        | Free => return (mkLogic ())
        | Unify(v1,v2) =>
           let val w1 = chaseLogic(evalV(env,v1))
               val w2 = chaseLogic(evalV(env,v2))
           in case (w1,w2) of
                (VLogic h1, _) =>
                  (update (h1,w2); return trueV)
              | (_, VLogic h2) =>
                  (update (h2,w1); return trueV)
              | (VInt i1, VInt i2) =>
                  if i1 = i2 then return falseV
                  else fail()
              | (VRecord(c1,_),VRecord(c2,_)) =>
                  if c1 = c2 then return falseV
                  else fail ()
           end
     end)
```

Figure 6. Interpreter changes for logic features. New and altered definitions are marked in the margin.

Or-thread queue. The execution state of the interpreter includes a double-ended queue `othreads` of "or-threads" that are waiting to execute. Each `othread` is represented by a pair containing a `computation` (environment, continuation, and expression) and a `heap` in which the computation should be run. Evaluation of a top-level expression begins with an empty queue. New `othreads` generated by narrowing are enqueued at the front of the queue (`ospawn`). When an `othread` completes (by delivering an answer or failing) or yields (`oyield`), the next `othread` to execute is taken from the front of the queue (`onext`). Execution terminates when no `othreads` remain on the queue.

To implement fair search, we arrange to bound the amount of computation any `othread` can do at any one time. This bound is implemented using a time-slice counter decremented on each `sexp` evaluation[1] and a polling mechanism whereby a thread calls `oyield` when its time slice is exhausted. `Othreads` thus behave essentially like *engines* [15]. The Determinism Property requires that program behavior is completely independent of the choice of value for `initialSlice` (except possibly for the order in which answers are produced). In general, performance will be enhanced by choosing a very large value for `initialSlice`, so that most computations complete before they exhaust their first slice. This minimizes the cost of switching between computations; more importantly, it minimizes the amount of live data held onto by computations pending on the queue. Choosing a large value for `initialSlice` essentially means that the interpreter will perform depth-first exploration of the narrowing options, except when it hits a very lengthy computation.

[1] It would actually suffice to decrement and check the counter only for `App` and `Force` expressions, because any non-terminating loop must involve one of these expressions.

Narrowing. Narrowing itself is implemented by extending the evaluation code for `case`$^{\mathtt{f}}$ expressions to cover the possibility that the scrutinee is a logic variable. Auxiliary function `narrow` is invoked for each case arm: it spawns a new `othread` with a forked copy of the current heap in which the scrutinized logic variable has been appropriately updated.

4.3 Concurrency and Residuation

We next describe how to support `ACore`'s concurrency and residuation features:

- The expression `(letpar` $\{x_i$ `=` $e_i\}$ `in` e`)` specifies that the expressions e_i should be evaluated in parallel, sharing the same heap. The resulting values are bound to the x_i; *all* the values must be produced before evaluation of e can proceed.
- Evaluation of a `case`$^{\mathtt{r}}$ or `checkInstantiated` expression residuates if the scrutinee is an uninstantiated logic variable (i.e., the scrutinee must evaluate to CHNF, not merely to HNF). The residuated computation blocks until the logic variable is instantiated by another parallel computation. If all parallel evaluations are blocked, the entire computation fails by floundering.
- The implementation of `enterBH` is changed to cause residuation rather than failure. The use of black-holing prevents two or more computations from attempting to update the same thunk simultaneously.

Figure 7 shows the added and altered interpreter code. Concurrency is implemented straightforwardly using a round-robin queue of "and-threads" (`athreads`), each corresponding to an expression being evaluated in parallel. Each `athread` consists of a `computation` and an integer update count (explained below). The result of each `athread` is stored in the heap, where it can be accessed by the common continuation of the `athreads`. An `othread` is redefined to be a set of `athreads`, all sharing a common heap.

A group of parallel `athreads` is spawned when a `letpar` expression is evaluated. Evaluation begins by creating a synchronization counter `hc` in the heap to track the number of uncompleted `athreads`, and a list of heap locations (all initialized to `VEmpty`) in which the `athread` results are to be stored. An `athread` is then spawned for each parallel expression. Each `athread` is given a `KUpdate` continuation frame that stores its answer in the heap. It then encounters a special `KSynch` frame, which implements a barrier synchronization with the other `athreads` in its spawn group; the last `athread` to complete continues by executing the body of the `letpar` in a suitably extended environment.

An `athread` that has residuated onto the queue (by calling `ayield`) should be restarted when the variable on which it is waiting has been instantiated. As a simple approximation to this, we arrange that the interpreter restarts an `athread` whenever *any* variable update has been performed since it residuated. The global `updateCount` tracks the number of updates so far, and the current value of this counter is recorded in the integer field of each `athread` when it yields. The `anext` routine selects the first *runnable*, i.e., potentially unblocked, `athread` from the queue. (`(Q.dequeueIf` p q`)` returns the first element a of queue q for which $p(a)$ is true.)

Note that the routines for managing the `othread` queue are redefined to save the entire current set of `athreads` when an `othread` is spawned or yields. Because the state of `athread` completion is recorded in the variant heap, behavior of a concurrent group is completely independent in each `othread` after an `ospawn`.

5 Implementation issues

5.1 Variant Heaps

There are several possible approaches to implementing the variant Heap ADT. The most naive would be to represent heaps by (extensible) arrays, and heap pointers by array indices, and implement `fork` by simply copying the array. However, this approach is likely to be quite space-inefficient, since much of the heap contents will be unchanged from variant to variant. A better tactic is to use a kind of copy-on-write. For example, if we represent heaps by immutable search trees, and heap pointers by integer keys, updating a variant will typically require copying only $O(\log n)$ nodes. However, this approach has a garbage collection problem when implemented within ML: a heap entry becomes dead only when (all) the heap(s) containing it do. In particular, in purely functional programs with no narrowing, *nothing* ever becomes garbage! We could avoid this problem if we were generating compiled code to run under our own garbage collector (rather than ML's), but removing dead entries from tree structures is still likely to be complicated and time-consuming.

An alternative approach, which we currently favor, represents each heap pointer by an ML reference cell, containing a list of possible values, each tagged with an integer *heap number*; the list is maintained in reverse sorted order by tag. Each time a heap is created by `new` or `fork`, it is assigned a new heap number from a global counter. A heap itself is represented by a list of heap numbers, whose head is the heap's unique number and whose tail is the representation of the heap's parent (hence empty for a heap created by `new`). To perform `(alloc` h v`)`, a new reference cell is created and initialized to the singleton list containing v tagged with the heap number of h. To do `(update` h `(`p`,`v`))`, v is tagged with the heap number of h and inserted into the existing list stored in the reference cell for p. To perform `(fetch` h p`)`, the list stored in the reference cell for p is searched for an entry tagged with some heap number that appears in the representation of h. If the requested pointer is really defined for heap h, its value must have been set either in h itself or in one of its ancestors, so the search is guaranteed to succeed.

To illustrate how this heap representation works, consider interpretation of the following program:

```
let x = free in let y = free in
 case^f x of
   True  => x
 | False => case^f y of True => x | False => x
```

Evaluating the `let` bindings causes two heap pointers `p0` and `p1` to be allocated for `x` and `y`, respectively, in the initial heap `h0`, whose heap number list is `[0]`. The resulting heap state is shown in Figure 8(a). Since `x` is uninstantiated, evaluating `(case`$^{\mathtt{f}}$ `x)` causes narrowing. Two `othreads` are created and two corresponding heaps, `h1` and `h2`, are obtained by invoking `(fork h0)` twice. `x` is bound to `True` in `h1` and `False` in `h2`, bringing the system to the state illustrated in Figure 8(b). Evaluation of the first `othread` causes a lookup of `x` in the corresponding heap, `h1`, whose heap number list is `[1,0]`. Searching the list pointed to by `p0` for a value tagged with 1 successfully produces `True`. Evaluating `(case`$^{\mathtt{f}}$ `y)` in the second `othread` causes a lookup of `y` in heap `h2`, whose heap number list is `[2,0]`. Searching the list pointed to by `p1` fails

```
val updateCount : int ref := ref 0

fun update (h,a) =
 (updateCount := !updateCount + 1;
  H.update (!heap) (h,a))

datatype kontframe = ...
                   | KSynch of hptr * env * exp

type athread = computation * int
val athreads : athread Q.q ref = ref Q.empty

type othread = heap * athread Q.q

fun onext () : unit =
 case (Q.dequeue (!othreads)) of
   SOME((heap',athreads'),rest) =>
      (...;
       athreads := athreads';
       anext ())
 | NONE => ()  (* done! *)

fun ospawn (comp:computation) : unit =
   othreads :=
     Q.enqueueFront (!othreads,
       (!heap,Q.enqueueFront (!athreads,(comp,~1))))

fun oyield (comp:computation): unit  =
  (othreads :=
     Q.enqueueRear (!othreads,
       (!heap,Q.enqueueRear (!athreads,(comp,~1))));
   onext ())

fun anext () : unit =
  let fun runnable (_,uc) = !updateCount > uc
  in case (Q.dequeueIf runnable (!athreads) of
       SOME((comp,_),rest) =>
          (athreads := rest;
           evalE comp)
     | NONE => fail ()
  end

fun aspawn (comp:computation) : unit =
  athreads := Q.enqueueFront (!athreads,(comp,~1))


fun ayield (comp:computation) : unit =
  (athreads := Q.enqueueRear (!athreads,
                      (comp,!updateCount));
   anext ())

fun enterBH (comp:computation) : unit = ayield comp
```

```
fun interp (e:exp) : unit =
   (athreads := Q.empty;
    updateCount := 0;
    ...)

and evalE (env:env,kont:kont,e:exp) : unit =
   case e of
     ...
   | Case(flx,v,pes) =>
      (case chaseLogic (evalV (env,v)) of
         ...
       | VLogic h0 =>
          (case flx of
             FLEXIBLE => ...
           | RIGID => ayield (env,kont,e)))
       | Letpar(xes,e) =>
          let val hc = alloc (VInt(length xes))
              val (xs,es) = unzip xes
              val hs = map(fn _=>alloc VEmpty) xs
              val env' = E.extend (env,
                   map (fn (x,h) => (x,VLogic h))
                       (zip (xs,hs)))
              val kont' = KSynch(hc,env',e)::kont
              fun f (e,h) =
                  aspawn (env,KUpdate h::kont',e)
          in app f (zip (es,hs));
             anext ()
          end

and evalS (env:env,kont:kont,s:sexp) : unit =
   ...
   case s of
     ...
   | CheckInstantiated v =>
      let val w = evalV (env,v)
      in case chaseLogic w of
           VLogic _ => ayield (env,kont,EofS s)
         | w => return w
      end

and continue (kont:kont) (w:value) : unit =
   case kont of
     ...
   | KSynch(hc,env',e')::kont' =>
      let val VInt c = fetch hc
      in if c = 1 then
           evalE (env',kont',e')
         else
           (update (hc,VInt(c-1));
            anext ())
      end
```

Figure 7. Interpreter changes to support concurrency. New and altered definitions are marked in the margin.

to find a value tagged with 2, but does find a value tagged with 0, namely `VEmpty`. In other words, `y` is still uninstantiated, so a further narrowing step occurs. This produces two new `othreads` and corresponding heaps `h3` and `h4`, forked from `h2`; `y` is bound to `True` in `h3` and `False` in `h4`. The resulting state is illustrated in Figure 8(c). Now evaluating the `othread` corresponding to the left case arm causes of a lookup of `x` in heap `h3`, whose heap number list is `[3,2,0]`; a value tagged with 2 is found, namely `False`. Evaluating the right arm in heap `h4` is similar.

In this implementation, `update` and `fetch` take $O(m)$ time in the worst case, where m is the number of distinct heaps containing the given pointer. They take only $O(1)$ time in the frequent case when the operation is being performed on the most-recently-created heap. Functions `alloc` and `fork` are $O(1)$.

When implemented inside Standard ML, this scheme still has a garbage collection problem: the values associated with a given `hptr` becomes garbage as soon as that `hptr` is no longer live, but

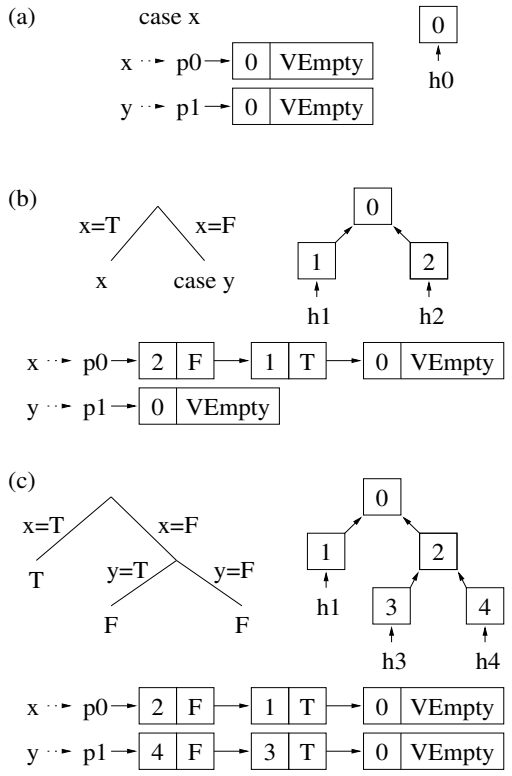

Figure 8. Heap states, showing `othread` tree, heaps, and heap pointers.

until then, *all* the values for that `hptr` stay live, even if the corresponding heaps are no longer live. Because heaps are closely associated with `othread`s, the interpreter knows when they become garbage, so it is possible to implement a form of manual garbage collection within a version of ML extended with weak pointers. The overheads of this scheme are fierce, but it does permit some examples that would otherwise exhaust memory to run to completion. However, in a compiled implementation where we can control memory management, this heap representation should be straightforward to garbage collect.

The problem of maintaining multiple variant stores has received occasional attention in the past, but there does not appear to have been any systematic treatment of the efficiency and garbage collection issues. Johnson and Duggan [16] and Morrisett [25] proposed first-class stores as an analogue to first-class continuations. Tolmach and Appel's ML debugger [32] implemented them to support roll-back. Most of these systems have a notion of a current store, which can be checkpointed and subsequently restored. A common mechanism for doing this is to record all updates in a script that can be undone or redone on demand; as Haynes [14] noted, this effectively generalizes the trail mechanism used in many backtracking-based systems. The use of copy-on-write for Unix `fork` apparently dates back to System V [23].

5.2 Fast Interpretation

The definitional interpreter as describe in Section 4 is too inefficient to be practical, but we can apply several well-known techniques to make it much more efficient.

DeBruijn indices. Two of the most obvious inefficiencies are performing string-based lookups in environments and in pattern lists at runtime. It is trivial to fix these problems by changing to DeBruijn indices for variables, and converting constructor names to numeric indices.

Trimming and closures Both `VClos` and `VThunk` values contain captured environments, which are used to resolve references to free variables in the abstraction or thunk body. Currently, the interpreter always captures the entire lexical environment at the point of definition, even though this may contain many entries in addition to the needed free variables. This policy makes environment capture cheap, and probably doesn't have much impact on lookup time but it can cause serious space leaks. Most functional language implementations therefore "trim" the environment to contain just the free variables [27]. If the free variable sets are put into closure records, variable lookup time can be reduced still further over the DeBruijn model, although there is a considerable cost in building the records.

6 Conclusions and Ongoing Work

The main goal of this work was to develop a practical FLP implementation that provides fair search based on multi-threading. We believe this goal has been achieved, although more work needs to be done to improve performance of variant heaps. We are also working on an alternative interpreter (implemented in Java) which provides fair search, but uses substitution rather than variant heaps; it should prove interesting to compare the performance of these approaches.

The design of `CCore` and `ACore` also sheds light on several dark corners of FLP semantics, and suggests some areas for improvement in the definition of Curry, currently the most visible FLP language. We expect to continue work on this front as well; in particular, we would like to formalize the Determinism Property and prove that it holds for `CCore`.

One important feature of Curry, which we have not discussed here for lack of space, is a facility for evaluating nested sub-programs and capturing their answer sets in a (lazy) list that can be inspected by the enclosing program. A distinctive characteristic of this mechanism is that sub-programs may read, but not write, heap entries created by the enclosing program. We have built an extended interpreter that faithfully implements this behavior, using an array of variant heaps. However, as currently specified in Curry, this feature clearly violates the Determinism Property, because the enclosing program's behavior can depend on the order in which answers to the sub-program are computed. We are therefore exploring alternative mechanisms that provide some of the same power without abandoning determinism.

We are also in the process of developing a compiler for `CCore` based on translation into a full continuation-passing style representation similar to that used in Standard ML of New Jersey [4]. This representation refines `ACore` by performing closure conversion (removing the need for environments) and introducing continuations (removing the need for a runtime stack and thus simplifying multi-threading). Queue and heap services are provided as part of the runtime system, using the same algorithms as in the interpreter. We plan to build a garbage collector tailored for both lazy evaluation and variant heap traversal. Our goal is to reach performance parity with the best current (non-fair) FLP compilers (such as the Münster Curry Compiler [21]).

Acknowledgements

We thanks the anonymous referees for several helpful suggestions.

7 References

[1] H. Aït-Kaci. An overview of LIFE. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.

[2] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for functional logic languages. In M. Comini and M. Falaschi, editors, *Proc. Int'l Workshop on Functional and (Constraint) Logic Programming*, volume 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[3] S. Antoy, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing strategies. In *PPDP01*, pages 207–217. ACM Press, 2001.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] K.-F. Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, June 1997.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI'93*, pages 237–247, 1993.

[7] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. of the 4th Int. Symp. on Programming Language Implementation and Logic Programming*, pages 1–23. Springer LNCS 631, 1992.

[8] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[9] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth Int. Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.

[10] M. Hanus. A unified computation model for declarative programming. In *Proc. 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)*, pages 9–24, 1997.

[11] M. Hanus. A unified computation model for functional and logic programming. In *Proc. POPL'97, 24st ACM Symp. on Principles of Programming Languages*, pages 80–93, 1997.

[12] M. Hanus, S. Antoy, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2002.

[13] M. Hanus, editor. Curry: An integrated functional logic language (version 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, Apr. 2004.

[14] C. T. Haynes. Logic Continuations. *Journal of Logic Programming*, 4(2):157–176, June 1987.

[15] C. T. Haynes and D. P. Friedman. Engines build process abstractions. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24, Aug. 1984.

[16] G. F. Johnson and D. Duggan. First-class stores and partial continuations in a programming language and environment. *Computer Languages*, 20(1):53–68, Mar. 1994.

[17] T. Johnsson. *Compiling lazy functional languages*. Ph.D. thesis, Chalmers University, 1987.

[18] R. E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[19] J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.

[20] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[21] W. Lux. The Münster Curry compiler. Available at `http://danae.uni-muenster.de/~lux/curry/`, 2004.

[22] J. McCarthy. A basis for a mathematical theory of computations. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[23] R. Miller. A Demand Paging Virtual Memory Manager for System V. In *USENIX Association Conference Proceedings*, pages 178–182, June 1984.

[24] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Standard ML Programming Language (Revised)*. MIT Press, 1997.

[25] J. G. Morrisett. Refining first-class stores. In *Proc. Workshop on State in Programming Languages*, pages 73–87, Copenhagen, Denmark, June 1993.

[26] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.

[27] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[28] E. Shapiro, editor. *Concurrent Prolog, Collected Papers, Volumes 1 and 2*. MIT Press, Cambridge, Massachusetts, 1987.

[29] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.

[30] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[31] A. Tolmach and S. Antoy. A monadic semantics for core Curry. In G. Vidal, editor, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[32] A. P. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.