Aspect-Oriented Programming in Higher-Order and Linear Logic

Chuck C. Liang

Department of Computer Science, Hofstra University
Hempstead, NY 11550
Email: chuck.liang@hofstra.edu

Abstract. Essential elements of aspect-oriented programming can be formulated as forms of logic programming. Extensions of Horn Clause Prolog provide richer abstraction and control mechanisms. Definite clauses that pertain to a common aspect, and which *crosscut* other program components, can be encapsulated using the connectives of higher-order intuitionistic logic. The integration or *weaving* of program fragments can be formulated using normalized forms of proof search in linear logic.

1 Introduction

Aspect-oriented programming [7] is emerging as an important advancement in software development. Its attraction lies in a new approach to modularity in the structuring of programs. Multiple concerns in the construction of software, such as security and optimization, *crosscut* each other and cannot be easily separated by traditional approaches to modular programming. AOP concerns program specifications as well as programming language characteristics. This paper focuses on the realization of AOP in a class of logic programming languages.

There is currently no widely accepted formal theory for AOP, unlike with the case of functional programming. However, much work already exist on the paradigm, including several formal specifications [2, 15, 16]. Logic programming has also been used [14] as a meta-programming device for AOP, generating code for conventional target languages (Java). Although the languages discussed here can also be used for this purpose, we are interested in writing aspect-oriented logic programs directly. One possible approach to this effect would be to extend Prolog by imitating the constructs of existing AOP languages such as AspectJ [6]. We offer a different approach here. We show the extent to which AOP concepts are already embodied in logics that are sufficiently expressive.

Using the terminology of AOP, one can consider a definite clause of a logic program as a piece of *advice* on how to proceed when certain conditions are encountered. These advice clauses are triggered at what are called *join points* in a program. The organization of a logic program also does not need to mimic the style of functionally or procedurally oriented programs. They can be grouped according to the aspect that they aim to address. For example, one may wish to consider all clauses concerned with error checking as a separate unit, regardless of

what predicate is at the head of a clause. In general, we can envision the following style of programming. Let $p_1 \ldots p_n$ be the predicates of a logic program. Let aspects (such as error checking) be represented by the symbols $t_1 \ldots t_m$. Instead of a singleton atom at the head of each definite clause we can qualify the atom using a new operator @, to indicate the aspect that the clause pertains to. The program will have the general form:

$$
\begin{array}{l}
p_1(\ldots) \ @ \ t_1 \ :- \ \mathcal{A}_1^1 \\
\vdots \\
p_n(\ldots) \ @ \ t_1 \ :- \ \mathcal{A}_n^1 \\
\qquad\qquad \vdots \\
\\
p_1(\ldots) \ @ \ t_m \ :- \ \mathcal{A}_1^m \\
\vdots \\
p_n(\ldots) \ @ \ t_m \ :- \ \mathcal{A}_n^m
\end{array}
$$

Formula $\mathcal{A}_i^j$ represents the "advice code" for predicate p_i *pertaining to* aspect t_j. Goal formulas indicate the aspects it should be solved *with respect to*, and have the form

$$G \ @ \ t_j \ @ \ \ldots \ @ \ t_k.$$

Any subset of $t_1 \ldots t_m$ can be used in a query. It will be shown in Section 3 that the operator @ can be modeled with multiplicative disjunction in linear logic. Each set of clauses pertaining to the same aspect constitute an aspect-oriented program fragment. Each such fragment may include clauses for any of the predicates $p_1 \ldots p_n$, thus *crosscutting* the organization of the base predicates.

To fully realize this form of *separation of concern* in programming, however, at least two important issues must be addressed. First, we wish to construct each aspect-oriented fragment not just as a loose collection of clauses but as a modular unit of abstraction, with the desired characteristics of locality and information hiding. Secondly and most delicately, mechanisms must be available to integrate or *weave* the various fragments into a coherent program. These issues are the focus of this paper and are addressed respectively in the following sections. Because of the lack of formal definitions for AOP concepts, our presentation relies significantly on examples. The paper culminates in the formulation of AOP as linear logic programming in Section 3.

2 Abstractions in Logic Programming

The first-order theory of Horn clauses that traditionally forms the foundation of logic programming is limited in its ability to provide mechanisms for abstraction. A formulation and classification of logic programming as deterministic proof search was given in [11]. Under such a generalized context, logics richer than Horn clauses can be considered as basis for logic programming. Higher-order,

intuitionistic and linear logics offer more complex mechanisms for expressing abstraction.

To provide a framework for discussion, we consider the Java language extension AspectJ [6], which seeks to support AOP in a general-purpose programming language. It is now the most popular manifestation of the paradigm. Several languages based on AspectJ have also been developed, including Aspectual CAML [13]. In AspectJ, program fragments that address a common concern can be encapsulated in modules called *aspects*. Such a structure may contain declarations or modifications of data structures that are specific to the aspect in question. For example, if the aspect concerns security, then a new field such as *encryption_key* may be added to an existing class. Join points are identified using a language (*pointcuts*) of regular expression-like patterns as well as primitives for determining more meaningful computational context. Aspects define *advice* code fragments that are executed at specified join points.

2.1 AOP in λProlog

λProlog, based on the theory of higher-order intuitionistic logic, extends traditional Prolog. Simply typed lambda terms and the associated unification algorithm are used in place of first-order terms and unification. Universal quantification, including quantification over predicates, can be used in goal clauses. The operational meaning of a goal of the form $\forall x.G$ is to prove G using a fresh constant for x. The intuitionistic connective for implication, unlike its classical counterpart, provides a stronger notion of scope, and can be used without restriction in λProlog. A goal $A \Rightarrow B$ is provable if and only if B is provable under the local assumption of A. These extensions provide a basis for expressing abstraction in programming. There is now a high-performance, compiler-based implementation [12] of λProlog.

Intuitionistic implication augments an existing program with a temporary clause, and can also be thought of as *adding a piece of advice to the existing program.* Likewise, higher-order universal quantification introduces a new constant, which can be a predicate or function symbol, to the existing signature. As early as in [9], it was demonstrated how these capabilities can be used to dynamically define new data structures in a program.

To demonstrate how AOP can be manifested in this setting, we use a predicate of the form

$$advice \quad Aspect\_name \quad Goal$$

as the head of λProlog clauses. Here, *Aspect_name* identifies the aspect or concern that the body of the clause gives advice to. *Goal* is a λProlog goal to which the advice is to be applied. We present in Figures 1 and 2 a simplified example to provide a comparison between a λProlog program and the corresponding AspectJ program. The purpose of this comparison is not to argue about the superiority of this or that language. We only wish to show how aspect-oriented concepts can be realized in entirely different contexts.

```
class aopbase
{
   static boolean divisible(int A, int B)
   { return (A % B == 0); }

   static int factorial(int N, int Accum)
   { if (N==0) return Accum; else return factorial(N-1,N*Accum); }
} // class aopbase

aspect parameters
{
   // advice to check that B is non-zero:
   boolean around(int B) :
        call(static boolean aopbase.divisible(..)) && args(..,B)
   {
        if (B==0)
        {
            System.out.println("warning: B is zero, returning false");
            return false;
        }
        else return proceed(B);
   }

   // advice to check that x is non-negative
   before(int x) : call(int aopbase.factorial(..)) && args(x,int)
   {
        if (x<0) throw new Error("invalid parameter");
   }

   // enforce that the initial value of the accumulator is 1
   int around(int N, int A) :
        call(int aopbase.factorial(..)) &&
            !withincode(int aopbase.factorial(..)) && args(N,A)
   {
        return proceed(N,1);
   }
} // aspect to check parameters

aspect trace
{
   before() : call(int aopbase.*(..))
   { System.out.println(thisJoinPoint); }

   declare precedence : trace, parameters; // trace has higher precedence
} // aspect to trace calls
```

Fig. 1. Sample AspectJ Program

```
module aopexample.

%% type declarations

type divisible int → int → o.
type fact int → int → int → o.
type advice string → o → o.
type useaspects (list string) → o → o.

%% base program

divisible A B :- 0 is (A mod B).
fact 0 A A.
fact N A B :- N1 is (N - 1), A1 is (N * A), fact N1 A1 B.

%% aspects

% clauses pertaining to aspect "parameters"
advice "parameters" G :-
      (∀A∀B (divisible A 0 :- print "warning...", !, fail))
      ⇒
      ∀ withinfact (
          (∀A∀B∀C (fact A B C :- A < 0, print "warning...", stop))
          ⇒
          (∀A∀B∀C (fact A B C :- not (withinfact), !,
                    withinfact ⇒ fact A 1 C))
          ⇒ G).

% clauses pertaining to aspect "trace"
advice "trace" G :-
      (∀A∀B (divisible A B :- printterm std_out (divisible A B), fail))
      ⇒
      (∀A∀B∀C (fact A B C :- printterm std_out (fact A B C), fail))
      ⇒ G.

%% integrating multiple aspects:

useaspects [] G :- G.
useaspects [A|As] G :- useaspects As (advice A G).
```

Fig. 2. Separation of Concerns in λProlog

The syntax of λProlog follows Prolog conventions except that applications are written in Curried form ($(f\ x)$ instead of $f(x)$). For readability we use the symbol $\forall$ for explicit universal quantification in goals. Other upper-case letters are implicitly quantified over the entire clause, as usual.

The *"base program"* for our example consists of two simple operations: that of checking for divisibility and the familiar tail-recursive factorial relation. We have deliberately left out the checking for invalid parameters in the base program. In the case of the factorial predicate, we have also not constrained that the initial value of the second parameter should be 1. We leave these separate concerns to an aspect module called "parameters". The second, "trace" aspect, which traces procedure calls, is perhaps the most popular example of AOP. These small programs may not be best-suited to illustrate the advantages of AOP over conventional methods, but it suffices to demonstrate the principle of separation-of-concerns and the kind of programming devices that can realize the aspect-oriented paradigm.

The use of the control primitive *!* is required in these examples. Also required is that in solving a goal of the form $A \Rightarrow B$ the new clause A is consulted first. These extra-logical characteristics are required to ensure that the advice clauses *must* be applied, as well as to specify the precedence ordering among advice. In other words, they control the *weaving* of the aspect-oriented fragments into the program. We shall use linear logic in Section 3 to achieve this purpose declaratively. However, λProlog currently provides a more practical implementation. The *withinfact* predicate, being quantified inside the body of the first clause, is local to the clause and represents another instrument for weaving. It serves to identify recursive calls to *fact,* for which the advice should not be applied, and is comparable to the *cflowbelow* pointcuts of AspectJ. Integration of multiple aspects is achieved with the *useaspects* clauses. The order of the aspect names in the list argument determines the precedence of the aspects. For example, calling

?- useaspects ["trace","parameters"] G

will apply the *trace* aspect first while solving G. Critically, however, the use of either aspect with the base program is optional.

Since λProlog was not implemented with AOP in mind, one cannot reasonably expect features such as *thisJoinPoint,* even as extra-logical additions. However, the essential aim of the separation of concerns is achieved. The advice clauses clearly *crosscut* the base program procedures.

In addition to declaring advice, we can use second-order quantification to introduce a new construct to a program. The purpose of the following clause is to implement a password-checking aspect for some arbitrary predicate $q\ A$:

```
advice "password protection" G :- ∀ pw ∀ passed (
  (∀A∀X (q A :- not(passed), !,  print "enter password:",
                 read X,  pw X,  passed ⇒ q A))
  ⇒
  (print "set password:",  read W,  pw W ⇒ G)).
```

The predicate *pw* is introduced to assert the password, and *passed* is used to signify that a valid password has been given. The scoping rules of the logical connectives are crucial to the validity of this clause. In particular, *pw* is a predicate symbol that is unique and local to the advice clause. It cannot appear free in G, and thus cannot be circumvented. Likewise, the *passed* predicate cannot be asserted arbitrarily except by the advice clause. The scope of $\Rightarrow$ restricts its assertion to individual goals. That is, multiple calls to *q,* excluding recursive calls, which are within the scope of $\Rightarrow$, will all require password checks.

Using logical abstraction mechanisms to reflect the aspect-oriented approach to program organization has obvious benefits. One of the criticisms of the AspectJ manifestation of AOP has been that it conflicts with the conventional notions of abstraction and information hiding found in Java-like languages. By formulating advice in light of lambda abstraction, universal quantification and implication, we can reconcile aspect orientation with well-understood notions of abstraction. This observation suggests that the perceived conflict between AOP and traditional abstraction principles are due to ad-hoc characteristics of non-declarative systems such as Java and AspectJ.

2.2 Join Points in the Continuation Passing Style

There are certainly features of Java and AspectJ that cannot be emulated easily in a logic programming language. On the other hand, there are also examples where an enriched logic programming language can offer AOP-related capabilities that are not found in conventional settings. Higher-order languages of both the functional and logic-programming varieties support the *continuation passing style* of programming. CPS introduces the sequential ordering of execution to a logic program. CPS in λProlog, given its ability to inspect the structure of λ-terms via higher-order unification, gives rise to interesting possibilities.

The following example is partially motivated by the image processing example described in [7]. Compared to the examples of the previous section, it better illustrates why one may wish to consider the AOP approach to program organization. The λProlog clauses of Figure 3 implement the typical higher order predicates, *map, fold* and *filter,* using a form of CPS. The last parameter of each predicate is a λ-term that relates the result of the current computation to a continuation goal.

The base program clauses are relatively elegant but lack refinement. When boolean operators are folded over lists, short-circuiting can be applied. Similarly, when an operation such as *filter* is immediately followed by one such as *map,* it is often possible to combine the operations, avoiding the generation of an intermediate list and improving efficiency. Adding such special-case clauses to the base program directly would compromise its elegance. The conventional, procedurally oriented approach would be to declare new procedures that encapsulate these cases for special treatment. Problems occur, however, when multiple features are required in combination. That is, combining the *short circuit* and *merge traversals* features would require yet another procedure. There are also situations, such as when no lists of booleans are present, when some refinements are not desired.

```
%% base program

type map (A → B) → (list A) → ((list B) → o) → o.
type fold (A → A → A) → A → (list A) → (A → o) → o.
type filter (A → o) → (list A) → ((list A) → o) → o.

map M [] G :- (G []).
map M [H|T] G :- map M T λx(G [(M H) | x]).

fold Op Id [] G :- (G Id).
fold Op Id [A|T] G :- fold Op Id T λx(G (Op A x)).

filter P [] G :- (G []).
filter P [H|T] G :- (P H), !, filter P T λx(G [H|x]).
filter P [H|T] G :- filter P T G.


%% aspects

advice "short circuit" G :-
      (∀L∀C ( fold and true [false|L] C :- !, (C false)))
      ⇒
      (∀L∀C ( fold or false [true|L] C :- !, (C true)))
      ⇒ G.

advice "merge traversals" G :-
      (∀P∀L∀Op∀Id∀Cg (
            map P L λx(fold Op Id x Cg) :- !, fold λaλb(Op (P a) (P b)) Id L Cg))
      ⇒
      (∀P∀L∀M∀Cg (
            filter P L λx(map M x Cg) :- !,
                  (∀H∀T (map M [H|T] Cg :- not (P H), !, map M T Cg))
                  ⇒ map M L Cg))
      ⇒ G.
```

Fig. 3. Optimization Aspects in Continuation Passing Style

For n distinct refinements, it is unlikely that one can foresee which of the 2^n possible subsets should be encapsulated. These problems are avoided by encapsulating the refinements not as ordinary procedures but as *aspects of separate concern.* They can be decoupled from a program as the situation demands.

Critical to this program is the use of higher-order unification, which identifies the join points where the advice clauses are applicable. We note that the pointcut language of AspectJ has no facility to identify situations when one function is called immediately after another, (such as in *f(); g();* or even just *g(f()).* The higher-order patterns of the *merge traversals* aspect not only identify such cases but also the condition that the result of the first operation is not used elsewhere in the continuation goal (i.e, x is not free in Cg).

A further implication of CPS is that it becomes possible to logically distinguish between advice that should be applied *before* and *after* a join point.

3 Weaving in Linear Logic

Logic programming languages have also been devised for linear logic [3], among them Forum [10], Lolli [5] (an executable fragment of Forum), LinLog [1] and Lygon [4]. Linear logic have been used to declaratively express computational properties such as side effects and concurrency. Forum in particular is complete with respect to linear logic, although formulas must be converted to a certain form. We have seen how the primitives of λProlog can provide a basis for aspect-oriented abstraction, although extra-logical features were needed to precisely control the *weaving* of aspects. Linear logic encompasses intuitionistic logic and the abstraction mechanisms described in the forgoing. In this section we describe how weaving can be formulated as proof search in linear logic. We shall write abstract program clauses in the form $Head \circ\!\!- Body$ where $\circ\!\!-$ is the reverse linear implication symbol. For sake of illustrations we assume the availability of arithmetic operations and the IO primitives *read* and *print*. That is, we assume that goals such as *read W* are provable from the empty linear context.

Linear logic requires the *accounting of resources* during proofs. This sensitivity can be used to formulate mechanisms for controlling the synthesis or weaving of program fragments.

We formulate AOP in linear logic as follows. Every aspect is associated with a unique predicate symbol or *token*, such as *trace*. Intuitively, each token identifies an aspect and represents an obligation to apply some advice. An advice clause that pertains to an aspect token t_k will have the general form

$$Head \parr t_k \parr \ldots \circ\!\!- Body$$

and goals will have the general form

$$G \parr t_1 \parr t_2 \parr \ldots \parr t_n$$

where $t_1 \ldots t_n$ represent *aspects that must be weaved into the solution of* G. Since all such tokens must be accounted for in solving G, their assertion entails the

application of the corresponding advice clauses. In other words, it is possible to associate with any goal a multiset of aspects, and we shall refer to $t_1 \ldots t_n$ as an *aspect multiset*. An equivalent scheme would be to have advice clauses of the form $H \circ\!\!- t_k \otimes \ldots \otimes Body$ and goals of the form $t_1 \multimap \ldots \multimap t_n \multimap G$. We prefer the form using $\wr$ since it names the aspect at the head of the clause.

For the aspect tokens to be distributed to the subgoals of G, G should be composed from connectives such as & and $\oplus$, which copy the linear context upon right-introduction (applied bottom-up). For goals formed from multiplicative connectives, multiple occurrences of the tokens may be required.

At first glance, the mechanism used here may seem little different from adding parameters to predicates. The role of aspect tokens, however, is to specify synchronization points during proof search. The tokens are associated not just with predicates but also with goal formulas.

As a simplified example, an advice to trace calls to the *divisible* predicate of Section 2 can be written as

$$!\forall A \forall B.\ divisible\ A\ B \wr\ trace \circ\!\!- print\ "calling\ \ldots" \otimes divisible\ A\ B.$$

The modal operator ! is intended to scope over the entire $\forall$-quantified clause [1].

The need for finer means for controlling weaving are illustrated by recursive predicates. We may wish some advice to be applied to each recursive call, and others to be applied only once and "as soon as possible." Specific to the $fact$ example, one advice checks for an invariance on the first parameter and should be applied for each recursive call. In contrast, the other advice ensures that the initial value of the accumulator is one, and must be used only at the outset. For *recursive advice,* we employ predicate tokens that are parameterized by the same inductive measure as the base predicate. This ensures synchronization with the corresponding advice clause each time the inductive measure is decreased:

$$!\forall A \forall B \forall C.\ fact\ A\ B\ C \wr\ check\ A \circ\!\!- A > 0 \otimes (fact\ A\ B\ C \wr\ check\ A–1)$$
$$!\forall B \forall C.\ fact\ 0\ B\ C \wr\ check\ 0 \circ\!\!- fact\ 0\ B\ C$$

In goal clauses, we complement this device by allowing for existential quantification over parameterized aspect tokens. Solving goals of the form $G \wr \exists x.t_k$, where G is composed from additive connectives, may use multiple instantiations for x should they be required.

The problem of ensuring that an advice is only applied at the outset is handled in AspectJ by specially designed pointcuts such as *!withincode(. . .).* Such fine-grained control over weaving can also be achieved by imposing a precedence ordering on advice clauses. We first observe that the "base" program fragment can be considered as just another aspect. We therefore introduce a *base* token and uniformly write all program clauses as advice clauses[2]. Precedence relations among advice can then determine the exact manner of weaving.

[1] The examples suggest that goals separated by $\otimes$ are called from left to right. The ordering of goals technically requires the continuation passing style. However, we forgo this refinement for sake of clarity.

[2] Implicitly there is a base token for each predicate, although it should also be possible for multiple predicates to form a common base aspect.

3.1 Proof Search, Modalities and Advice Precedence

Much of the non-determinism in linear logic proof search can be brought under control using normalized forms of proofs, such as the focused proofs of Andreoli [1] and the uniform proofs of Forum and Lolli. In such systems, the manner of proof search can be finely controlled. It is important to point out the following. Let Γ represent the multiset $\{A \parr C \circ\!\!-\!\!\bullet 1,\ B \circ\!\!-\!\!\bullet C\}$. Consider:

$$\dfrac{\vdash 1 \quad \dfrac{A \vdash A \quad \dfrac{C \vdash C \quad B \vdash B}{B \circ\!\!-\!\!\bullet C, C \vdash B}\ \multimap L}{B \circ\!\!-\!\!\bullet C, A \parr C \vdash A, B}\ \parr L}{\Gamma \ \vdash \ A, B}\ \multimap L \qquad \dfrac{B \vdash B \quad \dfrac{\vdash 1 \quad \dfrac{A \vdash A \quad C \vdash C}{A \parr C \vdash A, C}\ \parr L}{A \parr C \circ\!\!-\!\!\bullet 1 \ \vdash \ A, C}\ \multimap L}{\Gamma \ \vdash \ A, B}\ \multimap L$$

While both proofs are valid, only the right-hand one represents a *focused* proof (assuming atoms of negative polarity). all atoms at the head of the clause is found in the goal multiset. Thus the second clause in Γ must be applied first (from the bottom). Andreoli used the focusing property to define backchaining for clauses with multiple atoms at the head, thus providing a basis for linear logic programming. Uniform proofs behave similarly. The characteristic of ordered backchaining is the basis of our general scheme for weaving.

We define for each token t_k a unique predicate symbol $\widehat{t_k}$. If aspect t_j is to have lower precedence than $t_k \ldots t_l$ with respect to H, then their respective advice clauses will have the forms

$$H \parr t_j \parr \widehat{t_k} \ldots \parr \widehat{t_l} \circ\!\!-\!\!\bullet [\mathit{advice\ code} \ldots], \text{ and}$$

$$H \parr t_k \ldots \circ\!\!-\!\!\bullet [\mathit{advice\ code} \ldots] \otimes (H \parr ?\widehat{t_k}).$$

That is, the head of a t_j clause should contain $\widehat{t_k} \ldots \widehat{t_l}$ and the body of each clause for t_k asserts $?\widehat{t_k}$. The modal operator allows for the use of partial orderings, since multiple clauses may require the token. In the context of focused or uniform proofs, the assertion of $?\widehat{t_k}$ grants permission to advice with lower precedence than t_k to become applicable. The presence of $?\widehat{t_k}$ in a goal multiset also signifies that the goal is *no longer dependent* on aspect t_k.

Given aspects $t_1 \ldots t_n$, a goal of the form

$$G \parr t_1 \parr \ldots \parr t_m \parr ?\widehat{t_{m+1}} \parr \ldots \parr ?\widehat{t_n}$$

thus represents a computation that is dependent on aspects $t_1 \ldots t_m$ and independent of aspects $t_{m+1} \ldots t_n$.

To allow maximum flexibility in combining aspects with goals, we also use clauses of the form $H \parr t_k \circ\!\!-\!\!\bullet H$, to explicitly declare that aspect t_k is independent of goals H.

To illustrate the usage of this paradigm, we present in Figure 4 a full set of clauses based on the examples of Section 2. Assume it is desired that no advice should be executed before those of the *param* aspect and that *trace* is to have precedence over *check*.

$!\forall A \forall B.\ divisible\ A\ B \parr base \parr \widehat{param} \multimap A\ mod\ B = 0$
$!\forall A \forall B \forall C.\ (fact\ A\ B\ C) \parr base \parr \widehat{param} \multimap fact\ (A{-}1)\ (A * B)\ C \parr base$
$!\forall B.\ fact\ 0\ B\ B \parr base \parr \widehat{param}$

$!\forall A \forall B \forall C.\ fact\ A\ B\ C \parr param \multimap (fact\ A\ 1\ C) \parr ?\widehat{param}$
$!\forall A \forall B.\ divisible\ A\ B \parr param \multimap B \neq 0 \otimes (divisible\ A\ B \parr ?\widehat{param})$

$!\forall A \forall B \forall C.\ fact\ A\ B\ C \parr trace \parr \widehat{param} \multimap$
$\quad print\ "\ldots" \otimes (fact\ A\ B\ C \parr ?\widehat{trace})$
$!\forall A \forall B\ \ divisible\ A\ B \parr trace \multimap print\ "\ldots" \otimes divisible\ A\ B$

$!\forall A \forall B \forall C.\ (fact\ A\ B\ C) \parr check\ A \parr \widehat{trace} \parr \widehat{param} \multimap$
$\quad A > 0 \otimes (fact\ A\ B\ C \parr check\ A{-}1).$
$!\forall B \forall C.\ (fact\ 0\ B\ C) \parr check\ 0 \parr \widehat{trace} \parr \widehat{param} \multimap fact\ 0\ B\ C$
$!\forall A \forall B \forall N.\ divisible\ A\ B \parr check\ N \multimap divisible\ A\ B$

Fig. 4. Weaving of Aspects in Linear Logic

Note that tokens such as $?\widehat{param}$ need not be re-asserted by the clauses that depend on it, since the ?-formulas are reusable The last clause of Figure 4 specifies that *divisible* goals are independent of *check.* It is possible to generate such independence clauses between known atoms and aspects automatically.

Given the above logic program, a goal such as

$$\exists M(divisible\ 6\ 2\ \&\ fact\ 5\ 3\ M) \parr param \parr \exists N.(check\ N) \parr trace \parr base$$

would be solved as follows by a uniform-proof interpreter. The aspect multiset of the goal would be copied for both atomic subgoals upon $\&Right$. The independence clause for *divisible* eliminates the *check* obligation for the left subgoal. Since no precedence relation was defined between the *param* and *trace* clauses for *divisible*, either is applicable first. However, *base* is only applicable after $?\widehat{param}$ is asserted. For the $fact$ subgoal, the order of advice execution is necessarily *param*, *trace* and *check*. Each advice rewrites the aspect multiset to a new state. For example, after *trace*, the multiset becomes

$$\exists N.(check\ N) \parr ?\widehat{param} \parr ?\widehat{trace} \parr base$$

The parameter of *check* can only be instantiated with 5. Every recursive call to $fact$ will invoke the *check* advice.

As a variation, suppose we desired that tracing is not to be included in the computation. In that case *trace* should be replaced by $?\widehat{trace}$ in the initial goal.

The above scheme is not the only means for specifying precedence among advice. To enforce that t_k has precedence over $t_m \ldots t_n$, the advice clauses for t_k can also be of the form

$$Goal \parr t_k \parr t_m \parr \ldots \parr t_n \multimap Body \parr t_m \parr \ldots \parr t_n$$

That is, the head of the t_k advice clause should include the tokens for all aspects that t_k is to have precedence over. Backchaining over such a clause would be necessary *before* the tokens $t_m \ldots t_n$ are consumed. In this scheme, the body of advice clauses for aspect t_k must reassert the tokens $t_m \ldots t_n$. Suppose we wish to add an advice that takes user input for *divisible* goals. This advice should have precedence over *param*. Suppose further that we wish to add the advice without modifying the existing clauses (a desirable, though not always possible benefit of AOP). This *io* advice can be written as:

$$!\forall A \forall B.\ \mathit{divisible}\ A\ B\ \parr\ \mathit{io}\ \parr\ \mathit{param} \circ\!\!- \\ \mathit{read}\ A\ \otimes\ \mathit{read}\ B\ \otimes\ (\mathit{divisible}\ A\ B\ \parr\ \mathit{param}).$$

The new clause is consistent with those of Figure 4: no modification of the existing program was necessary. However, here the *io* aspect must always be used together with *param*. The scheme described above, using $?\widehat{t_k}$ formulas, will allow arbitrary aspects to be coupled with goals.

Additional control mechanisms for weaving can also be encoded. For example, control flow information, which in the context of proof search amounts to the subproof relation, can be captured using a pair of special tokens in_q and out_q for each predicate q. An advice that is only applicable outside of the flow control of q will include out_q at the head and assert in_q. An advice that is only applicable under the flow control of q can then check for the presence of the in_q token.

As a final example, we reformulate the password-protection aspect of Section 2 as a linear logic specification. Taking advantage of linear logic, we also add the ability to change passwords. The formulation again critically relies on second-order quantification:

$$\begin{array}{l} \exists\ \mathit{pw}\ \forall\ W\ \forall\ W'\ [\\ \quad (\mathit{read}\ W) \multimap \\ \quad (\ \mathit{pw}\ W\ \otimes\ (\mathit{pw}\ W' \multimap 1) \otimes \\ \qquad !\forall X \forall Y \forall G(\mathit{changepasswd}\ G \circ\!\!- \\ \qquad\qquad \mathit{read}\ X\ \otimes\ \mathit{pw}\ X\ \otimes\ \mathit{read}\ Y\ \otimes\ (\mathit{pw}\ Y\ \multimap G))\ \otimes \\ \qquad !\forall X(\mathit{checkpasswd} \circ\!\!-\ \mathit{read}\ X\ \otimes\ \mathit{pw}\ X\ \otimes\ (\mathit{pw}\ X \multimap \bot))\ \ )\ ] \end{array}$$

The specification can be used alongside any set of clauses as a *password-protection aspect*. Since the specification is to be kept on the left side of sequents, the existential quantification of *pw* ensures its locality (see [8] for thorough discussion on such uses of $\exists$-quantification). The *read W* clause sets the initial password. Since each clause rewrites the *pw* clause, the inclusion of $(\mathit{pw}\ W' \multimap 1)$ prevents the clause from becoming an unaccounted-for resource at the completion of proofs, as can be seen from the following derivation:

$$\dfrac{\dfrac{A \vdash A}{A, 1 \vdash A}\ 1L \qquad P \vdash P}{A, P, (P \multimap 1) \vdash A}\ \multimap\!L$$

Existing linear logic programs commonly use $\top$ to abort programs, even in the presence of unclaimed resources. Such a usage could neutralize the obligations

imposed by the aspect tokens. In particular, an advice could even be activated *after* the completion of the base program. Finer means are therefore preferable for the maintenance of resources.

Unlike other examples, we have chosen not to synchronize the *checkpasswd* advice with any specific predicate. This allows the advice to be weaved into any goal of the form $G \parr \mathit{checkpasswd}$. Furthermore, the solution of G can potentially proceed in parallel to the reading and checking of the password[3].

In terms of usage, several existing linear logic programming languages, such as LO and LinLog, allow for clauses whose heads are multisets of atomic formulas. However, these languages lack the abstraction mechanisms described in Section 2. Forum can of course be used for these specifications, but is too general to be interpreted efficiently. The simplified language Lolli also provides λProlog-style abstraction, and can be implemented effectively. However, clauses and goals must be rewritten with $\multimap$ and ! in place of $\parr$ and ?. All but the last of our examples can be converted to Lolli. With Lolli, clauses and constructs pertaining to a common aspect can again be encapsulated with intuitionistic implication and quantification. The abstraction scheme for the separation of concerns can thus be merged with the weaving mechanisms of linear logic.

4 Future Work

Another approach to weaving logic program fragments is through meta programming. We can use a specification language that allows us to declare aspects and weaving relations in a more natural manner, such as:

aspect trace, param o.
aspect check int $\rightarrow o$.
precedence trace check.
precedence param trace. etc ...

A meta-program can be devised to check for circularity among the precedence declarations, then transform a given logic program by adding the required $\widehat{t_k}$ and $?\widehat{t_k}$ tokens. The meta-program can also automatically generate the independence clauses for unrelated aspects and goals (such as between *check* and *divisible* in Figure 4). The task of writing advice clauses would become more intuitive. Such a device also improves the ability to incorporate new aspects and advice while minimally altering existing code. Furthermore, as Miller has noted, linear logic programs behave like ordinary Prolog programs most of the time. We can envision extending ordinary Prolog in a minimal way, by adding the *"@"* operator alluded to in the introduction. Together with a specification such as above, a

[3] A subtle point here is the use of $pwX \multimap \bot$ (equivalently $pwX^{\bot} \parr \bot$) instead of simply $pwX^{\bot}$ or using $\mathit{checkpasswd} \parr pwX$ at the head of the clause. By hiding the atom pwX under a right-asynchronous connective, we cause proof search to "loose focus" on the atom, delaying its use until needed. Without this device, the *checkpasswd* clause cannot be applied before solving G.

meta program can then compile a Prolog program into a linear logic program. Thus the mechanisms described here can also be used as a basis for adopting AOP to Prolog.

Acknowledgments

The author wishes to acknowledge Dale Miller for valuable advice and discussion.

References

1. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.
2. G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In *Fifteenth International Conference on Concurrency Theory (CONCUR 2004)*, LNCS vol. 3170, pages 209–224. Springer-Verlag, 2004.
3. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
4. James Harland, David J. Pym, and Michael Winikoff. Programming in lygon: An overview. In *AMAST*, pages 391–405, 1996.
5. Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
6. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*, LNCS vol. 2072, pages 327–353. Springer-Verlag, 2001.
7. G. Kiczales, J. Lamping, A. Menhdekar, C. Maeda, C. Lopes, J. Loingties, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-oriented Programming*, LNCS vol. 1241, pages 220–242. Springer-Verlag, 1997.
8. D. Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283. MIT Press, June 1989.
9. D. Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.
10. D. Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, 1996.
11. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
12. G. Nadathur and D. J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In *Automated Deduction–CADE-16*, number 1632 in LNCS, pages 287–291. Springer-Verlag, 1999.
13. Hideaki Tatsuzawa, Hidehiko Masuhara, and Akinori Yonezawa. Aspectual Caml: An aspect-oriented functional language. In *10th ACM SIGLAN International Conference on Functional Programming*, 2005.
14. K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In *2nd International Conference on Meta-Level Architectures and Reflection*, LNCS vol. 1616, pages 250–272. Springer-Verlag, 1999.
15. D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *International Conference on Functional Programming*, pages 127–139, 2003.
16. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, 2004.