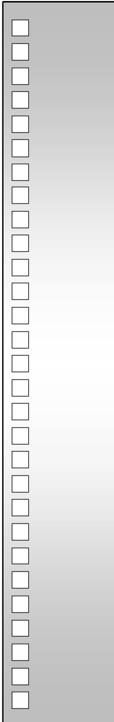


Control Localization in Domain Specific Translation

Ted J. Biggerstaff
tbiggerstaff@austin.rr.com

Overriding Problem: Antagonistic Goals

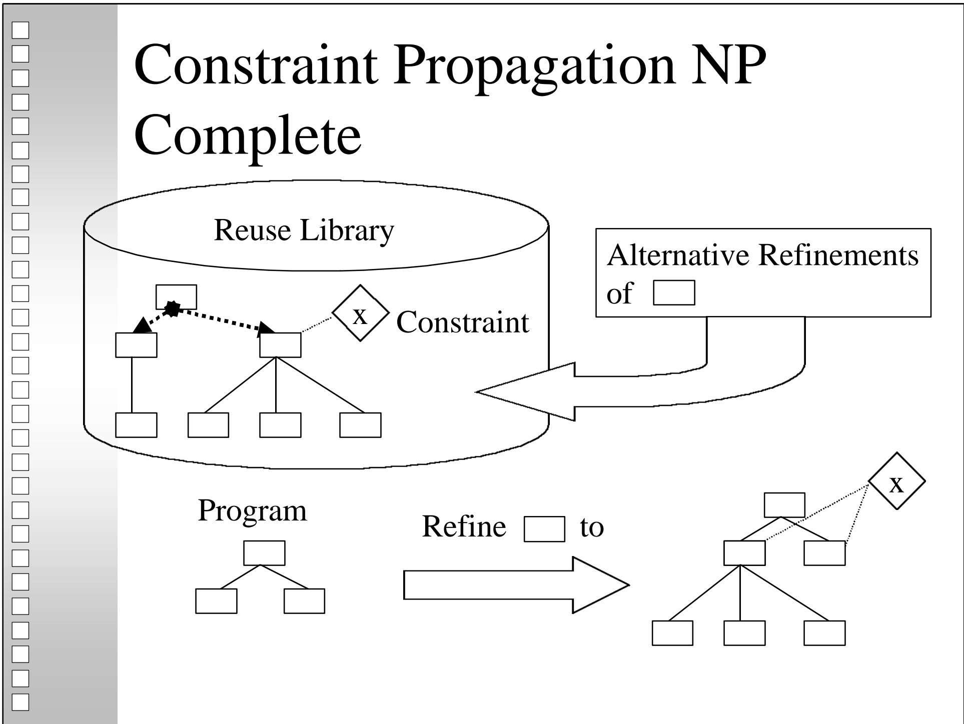
- High level operators and operands provide programming leverage & variations
  - E.g., (image ⊕ neighborhood) convolution
- But fracture and de-localize code pieces
  - $b=((a \oplus s)^2 + (a \oplus s')^2)^{1/2}$
  - Needed optimizations: code sharing, re-org. & re-weaving
- Conventional Optimization approaches induce large search spaces (EXPLOSION)

The central difficulty revolves around delocalization of information. [Letovsky and Soloway] If I factor the operators and operands into highly general constructs, I can write combinatorially many compact expressions with them that effectively form an infinite *virtual library* of reusable components, one component for each possible composite expression. For example, the convolution operator ⊕ defines the general structure of the convolution computation, i.e., it is a sum of pixels within a neighborhood times weights. The characteristics of the neighborhood is not defined by the convolution operator. They are defined by the template, s and s', in the equation. That is , what are the dimensions of the neighborhood, how are the weights calculated and what variables do they depend on, and what if any special cases are there (e.g., are boundary cases processed differently from non-boundary cases). In this example, they are. S and sp define all of these SPECIFICS of the convolution operation.

However, this means that the tightly integrated information needed by the compiler to generate high performance code is split across many operators and operands. With current technology, compiling such expressions requires huge search spaces of possible transformation sequences to assure finding the optimal localizations for high performance execution.

On the other hand, if I define less general operators and operands in which cross operator code is already localized for performance reasons, the number of possible variations that can be produced by my generator drops precipitously and my infinite virtual library very likely becomes a finite library.

In summary, the problem is trying to achieve three goals simultaneously; 1) factoring domain into high leverage operators and operands, 2) compiling expressions of these operators and operands into high performance code, and 3) doing the compiling without engendering a large search space that renders the compilation algorithms impractical.

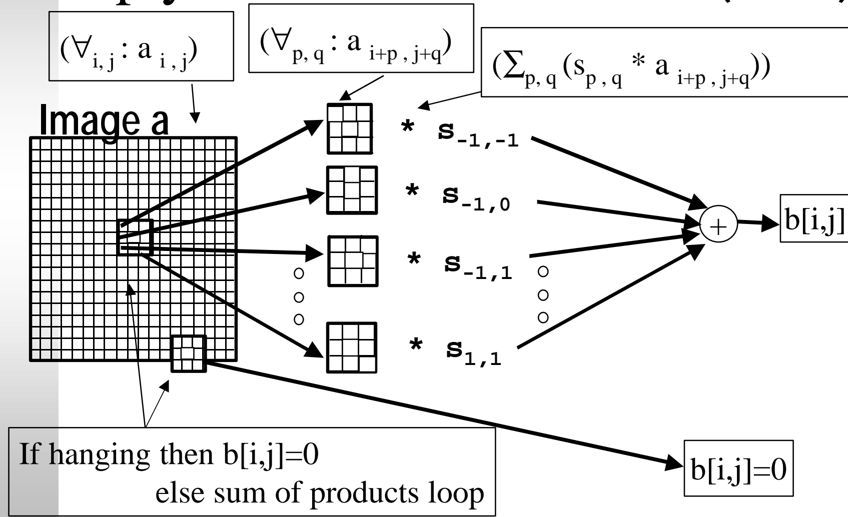


Constraint propagation is the process whereby design choices made in one part of the program are coordinated with design choices made in other parts of the program. For example, here we have a program that consists of two partially refined parts (turquoise and yellow). Looking in the reusable data base, we find that we can refine the turquoise block two ways – green or red. This might be a data structure choice between a string or linked list implementation. This decision constrains refinements elsewhere. Call this constraint, X. X will have to be propagated to other parts of the program where it might constrain other refinement (or optimization) choices. Finding a consistent set of refinements (or optimizations) is at least as hard as finding a set of values for Boolean variables that satisfy a given Boolean expression and the Boolean satisfiability problem is an NP complete problem.

- Implicit Phases based on Distinct DSLs
  - Mutually Exclusive DSL Operators & Operands
- Inter-Phase Optimization
  - Metaprogram = {Refine DSL; Optimize Refined DSL}
- Result: Group Rules by Phase
  - One large search space to several small ones

One way to reduce the search space is by grouping transformations so that at each decision point only a small number of relevant transformations are possible. Draco implicitly groups transformations by organizing translation into a set of stages each of which translates a higher level DSL into a lower level. These translation stages continue recursively until the program is fully represented in a conventional programming language such as C, C++, or Java. The staging is induced by each distinct DSL being defined by a set of distinct operator and operand types. So, this leads to a sequence of little search spaces that easily contain search space explosions.

There is a lingering PD explosion problem – automatically written code is often overly complex or simply dumb. This can lead to PD explosion or failure of correct refinements to fire because their enabling conditions are too hard to check. To thwart these difficulties, Draco uses a metaprogram that performs a code simplification (called “optimization” by the Draco literature) after each DSL to DSL translation using a separate group of transformations designed specifically for simplification.



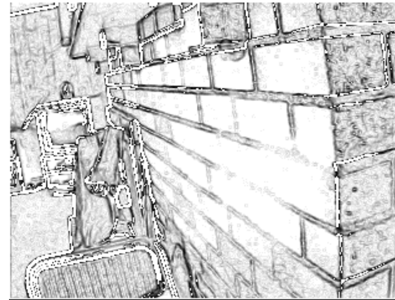
This is the definition of a convolution operator.

The problem

DSL for Sobel Edge Detection

$$b=[(a \oplus s)^2 + (a \oplus s')^2]^{1/2}$$

a



b

Lets look at an example that will motivate the kinds of problems that reorganizing generators address. Here is a concrete example of an expression of high level operators (e.g., convolution) operating on composite data structures (e.g., images). In this case, we have m X n grayscale images a and b (defined elsewhere) with pixels represented by integers within some grayscale range. S and sp are neighborhoods that are parametrically centered on some [i,j] pixel define the height, width, weights, and special case processing for a neighborhood.

The operators are:

convolution – defined as the sum of products of the weights associated with a pixel neighborhood (i.e., s or sp) that is centered on some [i,j] pixel in an image and produces a new [i,j] pixel in a new image (e.g., "$(a \oplus s)$").

Arithmetic operators (e.g., +, square, square root) – pixel by pixel arithmetic operations on images.

This expression applied to image a does a so-called Sobel line detection to produce image b as seen here. Most image and signal processing operations can be represented by such expressions.

The principles and mechanisms of this domain have wide scale application via analogs in very different domains such as UI, web, DB, and middleware interfaces.

Large Grain Extended Expressions

$(\Sigma_{p,q} (a_{i+p,\, j+q} * s_{p,q}))$

$(\Sigma_{p,q} (a_{v+p,\, w+q} * s'_{p,q}))$

$$b = ((a \oplus s)^2 + (a \oplus s')^2)^{1/2}$$

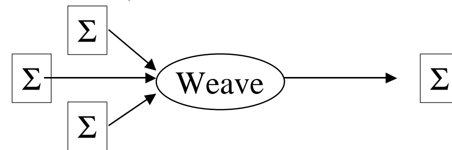
$(\forall_{\mathbf{d,e}} \mathbf{:}\ \mathbf{a}_{\mathbf{d,e}})$

$(\forall_{\mathbf{i,j}} \mathbf{:}\ \mathbf{a}_{\mathbf{i,j}})$

$(\forall_{\mathbf{v,w}} \mathbf{:}\ \mathbf{a}_{\mathbf{v,w}})$

Explosion Control

- DSL-> DSL translation phases
- DS optimizations
- Specialized metaprograms (Control Localization)



The approach is to avoid the general problem by writing a custom MetaProgram (expressed as a set of rules) with the overall objective of localizing chunks of control implied by large grain operators and operands. **Control localization means** to re-express or re-form or weave the control chunks together so that they minimize the amount of computation over the expression. Operationally, this means share as much as possible.

•Share indexes where possible.

•Share passes over the large grain data structures where possible.

•Share intermediate computations where possible.

In essence, we will generate a **custom control structure for each expression** of large grain operators and operands. What is going on here, in terms of the Constraint Propagation problem, is that the localization rules are designed to **coordinate the constraints** introduced by refinement of the large grain operators and operands and to do so over just a portion or the program represented by a single expression of large grain operators and operands. Thus, rather than solving the general constraint propagation problem over the whole program, we are solving a **highly specialized subproblem** (i.e., localization) over a window of the target program (i.e., an expression).

Localizing Implied Control

$b=((a \oplus s)^2 + (a \oplus s')^2)^{1/2}$

$b=(((\forall_{i,j} : a_{i,j}) \oplus s)^2$
$\quad + ((\forall_{v,w} : a_{v,w}) \oplus s')^2)^{1/2}$

$b=(((\forall_{i,j} : (\sum_{p,q}:a_{i+p,j+q} * s_{p,q}))^2)$
$+((\forall_{v,w}:(\sum_{p,q}:a_{v+p,q+w}*s'_{p,q}))^2))^{1/2}$

Incremental rewrites of abstract syntax tree (AST) shown as an expression. Later, we will look at one of the transformations that accomplish one of these rewrites.

Expr1->Expr2: Refine from image (a) to pixels (a[i,j] and a[v,w]).

Expr2->Expr3: Introduce definition of convolution and relate its outer 2D loop to loop introduced by instances of a (e.g., ($\forall$ i,j) and ($\forall$ v,w)).

Localizing Implied Control

$$b=(((\forall_{i,j} : (\Sigma_{p,q}:a_{i+p,j+q} * s_{p,q}))^2) +((\forall_{v,w}:(\Sigma_{p,q}:a_{v+p,w+q} * s'_{p,q}))^2))^{1/2}$$

$$b=((\forall_{i,j} : ((\Sigma_{p,q}: a_{i+p,j+q} * s_{p,q}))^2) +(\forall_{v,w}:((\Sigma_{p,q}: a_{v+p,w+q} * s'_{p,q}))^2))^{1/2}$$

$$b=(\forall_{i,j} : (((\Sigma_{p,q}:a_{i,j} * s_{i+p,j+q}))^2) + (((\Sigma_{p,q}: a_{i+p,j+q} * s'_{p,q}))^2))^{1/2}$$

Expr3->Expr4: Outer 2D loop to loops introduced by instances of a (e.g., ($\forall$ i,j) and ($\forall$ v,w)) can be moved to include the square operation based on the semantics of square.

Expr4->Expr5: Merge outer 2D loop to loops introduced by instances of a retaining ($\forall$ i,j) and throwing ($\forall$ v,w) away. Coordinate uses of indexes over expression.

Localizing Implied Control

$$b=(\forall_{i,j} : (((\Sigma_{p,q}:a_{i,j} * s_{i+p,j+q}))^2) + (((\Sigma_{p,q}: a_{i+p,j+q}*s'_{p,q}))^2))^{1/2}$$

$$b=\forall_{i,j} : ((((\Sigma_{p,q}:a_{i,j} * s_{i+p,j+q}))^2) + (((\Sigma_{p,q}: a_{i+p,j+q}*s'_{p,q}))^2))^{1/2}$$

$$(\forall_{d,e}:b_{d,e})=\forall_{i,j} : ((((\Sigma_{p,q}:a_{i,j}*s_{i+p,j+q}))^2) + (((\Sigma_{p,q}: a_{i+p,j+q}*s'_{p,q}))^2))^{1/2}$$

Expr5->Expr6: Outer 2D loop to loops introduced by instances of a (e.g., ($\forall$ i,j) and ($\forall$ v,w)) can be moved to include the square root operation based on the semantics of square root (performed by same rule that moved them to the square operator).

Expr6->Expr7: Refine from image (b) to pixel (a[d,e]) and introduce control structure ($\forall$ d,e) .

Localizing Implied Control

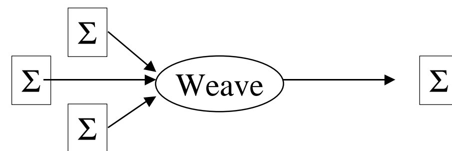
$$(\forall_{d,e}:b_{d,e}) = \forall_{i,j} : ((((\Sigma_{p,q}:a_{i,j}*s_{i+p,j+q}))^2) + (((\Sigma_{p,q}: a_{i+p,j+q}*s'_{p,q}))^2))^{1/2}$$

$$(\forall_{i,j} : (b_{i,j} = (((\Sigma_{p,q}:a_{i,j}*s_{i+p,j+q}))^2) + (((\Sigma_{p,q}: a_{i+p,j+q}*s'_{p,q}))^2))^{1/2})$$

Expr7->Expr8: Outer 2D loop introduced by instances of a (e.g., (i,j)) can be moved to include the assignment operation based on the semantics of assignment and merged to share the indexes. Indexes i and j survive and d and e are thrown away. Index usage is coordinated and revised across expression.

Explosion Control

- DSL-> DSL translation phases
- DS optimizations
- Specialized metaprograms (Control Localization)



- Group rules by object & phase
- Use domain knowledge

Explicit Entities & Phases

Phase (Job) / Type (Semantics)	Loop Optimization (Fusion2)	Code Generation (Fusion3)	. . .
⊕ Operator	$\phi \Rightarrow \varepsilon, \beta \Rightarrow \alpha$	$\kappa \Rightarrow \lambda, \theta \Rightarrow \rho$	. . .
Image	$\theta \Rightarrow \sigma, \theta \Rightarrow \rho$	$\beta \Rightarrow \alpha, \theta \Rightarrow \sigma$	. . .
Neighbor-hood	$\kappa \Rightarrow \lambda, \theta \Rightarrow \rho$	$\phi \Rightarrow \varepsilon, \theta \Rightarrow \sigma$	. . .
. . .	. . .	. . .	. . .

The type (or specific object) reveals the semantics, which implies domain knowledge such as data flow constraints. The explicit phase names pin down the specific step of the job at hand. So, let us look at an example …

Example Loop Opt. Transform

Phase (Job) / Type (Semantics)	Loop Optimization (Fusion2)	Code Generation (Fusion3)	. . .
⊕ Operator			. . .
Image	$\theta \Rightarrow \sigma$		. . .
Neighbor-hood			. . .
. . .	. . .	. . .	. . .

When the loop optimization phase (called "fusion2" for historical reasons) traverses the expression and happens upon the instances of the images a and b in the example expression, it will refine the a or b into a black and white pixel & attach some shorthand information characterizing the loop to that pixel. So, let's look at the transformation that does this operation.

Example (Idealized) AST Transformation

$$b=((a \oplus s)^2 + (a \oplus s')^2)^{1/2}$$

$$\Rightarrow$$

$$b=(((\forall_{i,j} : a_{i,j}) \oplus s)^2 + (a \oplus s')^2)^{1/2}$$

We will look at the transformation rule that accomplishes this AST rewrite, which was one step in the derivation that we looked at earlier. Refining an instance of the image a into the pixel a[i,j] and introducing a nugget of control ($\forall$ i,j).

Example (Actual) AST Transformation

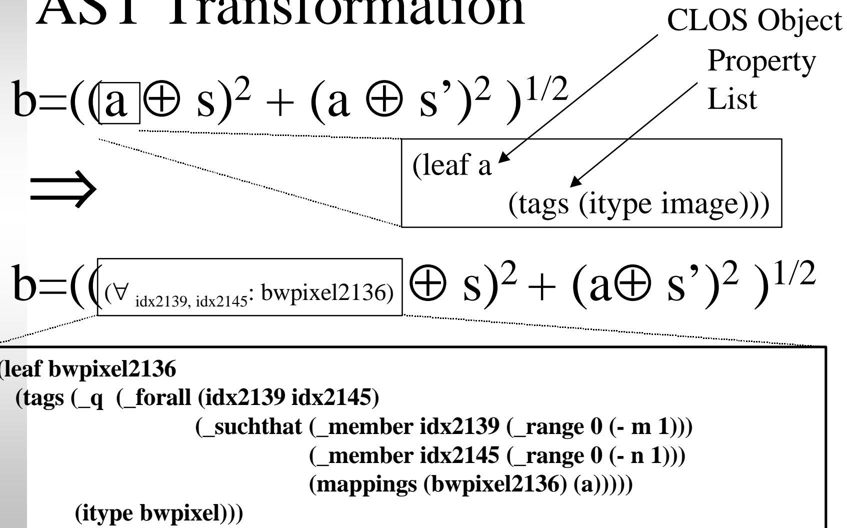
$b=((a \oplus s)^2 + (a \oplus s')^2)^{1/2}$

$\Rightarrow$

$b=(((\forall_{idx2139, idx2145}: bwpixel2136) \oplus s)^2 + (a \oplus s')^2)^{1/2}$

Where $bwpixel2136 = a_{idx2139, idx2145}$

Some information on the implementation structures. The loop indexes introduce by the translator that we idealistically call i and j are really slightly uglier translator generated symbols such as idx2139 and idx2145. The translator replaces a[idx2139,idx2145] with a translator generated black and white pixel bwpixel2136 and remembers the relationship between bwpixel2136 and a[idx2139,idx2145].



The AST tree implementation structures are shown. Every AST node is a LISP list and ends with a property list called a "tags" list. AOG stores all kinds of information on the property lists. In this example, the "before" form of the AST node just contains the inferred type of the node, which is produced by a set of type inference rules that are applied at the outset of translation.

The after form of the node contains a shorthand notation (_forall (*loop variables*) (_suchthat *constraints*)) that captures the essential information about the control structure. These will be moved, merged, and used to compute control nestings and equivalence classes of combinable elements. _member clauses indicate ranges for index variables. Mappings indicates (elements, composites) relationships and sets of equivalent entities (e.g., (bwpixel2136 and bwpixel2456) would mean that these two translator generate symbols represent equivalent entities).

Example Loop Opt. Transform

❖Refine Image to BWPixel with loop shorthand tags

(=> compositeleaf fusion2 image Name, Phase, Location

LHS

RHS

PreRoutine PostRoutine)

AOG provides a macro (=>) that creates each PD transformation. It requires a transformation name (in this case, compositeleaf), a phase name (fusion2), and a type name (image). The left hand side is a pattern written in a pattern language. When applied to a position of the AST, a successful pattern match will produce a binding list of design variables and bindings that are pieces or places in the AST. Those bindings are used to instantiate the rhs expression which is then used to rewrite the match expression.

This is the rule that refines an image such as “a” into a pixel (in our example a black and white pixel) (e.g., a[i,j]) and introduces a shorthand for the implied control structure.

Example Loop Opt. Transform

❖Refine Image to BWPixel with loop shorthand tags

(=> compositeleaf fusion2 image — Name, Phase, Location

`$(pand $(por (leaf ?op) ?op))

LHS ⇒ ***…sub-pattern to get type …***

…sub-pattern to get dimensions of ?op…)

RHS

PreRoutine PostRoutine)

Now, let's look at some of the pattern of this transformation. The pattern language is an inverse quoting language in which unadorned structures are literal data to be matched against the AST (e.g., "leaf"). Expressions adorned with a question mark are variables that if unbound, will match and become bound to anything at the corresponding position in the AST. If the variable is already bound, then that value acts like a literal and must match that position in the AST.

Those expressions that are adorned with a dollar sign represent matching operators such as And (Pand) and Or (por). There are about 25 to 30 such operators and the pattern language is "open" allowing arbitrary new operators to be defined by the user. This pattern requires that three clauses match successfully. The first will bind the pattern variable ?op to the instance of the image (e.g., a or b in our example). This instance of a or b may be stored in a "leaf" list so that it can have a property list (called a "tags" list) tacked onto the end of the instance of a or b. Such structures are just implementation details that are not fundamental to the design. The deeper property is that any AST structure can have a property list.

The second expression will get the type structure mainly to bind context around the type structure that will have to be recreated in the rewritten expression. The third pattern determines the dimensions of the image (which may be code expressions) for use in constructing the eventual loop.

Example Loop Opt. Transform

❖Refine Image to BWPixel with loop shorthand tags

Name, Phase, Location

```
(=> compositeleaf  fusion2  image
    `$(pand $(por (leaf ?op) ?op))
          ($(spanto ?pre (tags))
                  (tags $(spanto ?pretags (itype ?itype))
                  ?spaceover $(remain ?posttags)))
          …sub-pattern to get dimensions of ?op…)

    RHS

    PreRoutine PostRoutine)
```

LHS

This reveals more examples of the pattern language, in particular the spanto operator which searches down a list looking for a particular pattern. Every structure in the AST has a property list for storing various pieces of translation information. This is just a list whose first element is the atom "tags". This second pattern is spanning down to find the tags list and then spanning to find the type structure. In the course of this, it binds the sections of the tags list before and after the type structure to ?pretags and ?posttags, respectively. These will be copied verbatim into the tags list of the rewritten expression.

Example Loop Opt. Transform

❖Refine Image to BWPixel with loop shorthand tags
❖Enablecompositeleaf creates ?newleaf, ?idx1, ?idx2

Name, Phase, Location

```
(=> compositeleaf  fusion2  image
    `$(pand $(por (leaf ?op) ?op))
           ($(spanto ?pre (tags))
                  (tags $(spanto ?pretags (itype ?itype))
                  ?spaceover $(remain ?posttags)))
           …sub-pattern to get dimensions of ?op…)
```

LHS

RHS

enablecompositeleaf nil)

Pre &Post Routines

Before the rhs is used to rewrite the matched portion of the AST, the PreRoutine (if a routine name is present) is called to further check enabling conditions and perform translation chores such as creating translation variables. For example, in this case the routine enablecompositeleaf will create a black and white pixel object (bound to ?newleaf) and two iterators (bound to ?idx1 and ?idx2) that will be candidates for the resultant loop indexes.

RHS Rewrite Form

```
`(leaf ?newleaf
      (tags (commasplice ?pretags)
          (_q (_forall (?idx1 ?idx2)
                    (suchthat (_member ?idx1 (_range ?ilow ?ihigh))
                              (_member ?idx2 (_range ?jlow ?jhigh))
                              (mappings (?newleaf) (?op)))))
         (itype BWPixel) (commasplice ?posttags)))
```

The RHS of the rule is a structure with variables and some meta-operators (e.g., commasplice, which will splice together lists and is equivalent to the LISP ,@ operator). When the binding list from the pattern match and the preroutine is substituted for the variables and the meta operators executed, this will produce the new RHS that will replace the portion of the AST matched by the LHS of the rule.

Example Loop Opt. Transform

- ❖Refine Image to BWPixel with loop shorthand tags
- ❖Enablecompositeleaf creates ?newleaf, ?idx1, ?idx2

Name, Phase, Location

LHS

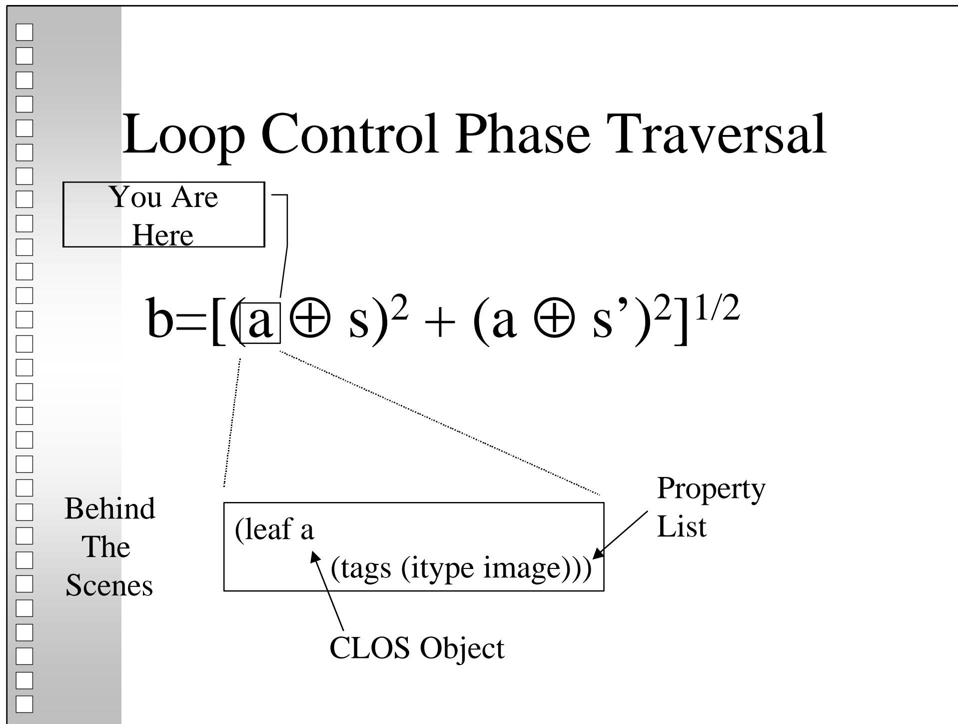
RHS

```
(=> compositeleaf  fusion2  image
    `$(pand $(por (leaf ?op) ?op))
          ($(spanto ?pre (tags))
                 (tags $(spanto ?pretags (itype ?itype))
                 ?spaceover $(remain ?posttags)))
          …sub-pattern to get dimensions of ?op…)
   `(leaf ?newleaf
       (tags (commasplice ?pretags)
          (_q (_forall (?idx1 ?idx2)
                  (suchthat (_member ?idx1 (_range ?ilow ?ihigh))
                        (_member ?idx2 (_range ?jlow ?jhigh))
                        (mappings (?newleaf) (?op)))))
          (itype BWPixel) (commasplice ?posttags)))
   enablecompositeleaf nil)
```

Pre &Post Routines

This shows the specification of the rewritten structure. The rhs will be instantiated with the bindings from the pattern matching and the preroutine. The language provides some metaoperators to make the construction easier. Here, the commasplice operator splices lists together to produce a tags property list containing whatever was in the pre-rewrite property list plus the new type pair and the loop shorthand expression.

So, let's look at how this rule would operate in the context of doing loop control inference on the Sobel edge detection example.b



Let's follow through the of the compositeleaf rule applied to the first "a" in this AST.

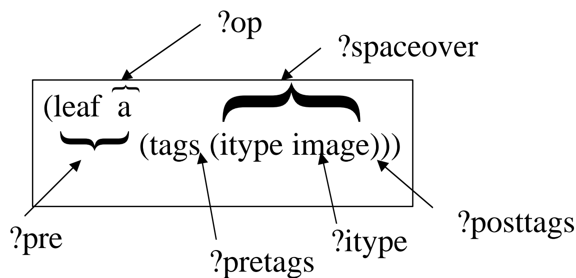
We will assume that in the course of traversing the expression, we have just encountered the first instance of the image "a". "a" is a CLOS object with a variety of instance variables and properties. This instance of "a" also has a property list (i.e., the tags list) and internally this instance is stored as a "leaf" list to provide a place to hang the property list. The only thing on the property list at this moment is a type pair.

Pattern Match Result

$b=[(\boxed{a} \oplus s)^2 + (a \oplus s')^2]^{1/2}$

```
`$(pand $(por (leaf ?op) ?op))
          ($(spanto ?pre (tags))
            (tags $(spanto ?pretags (itype ?itype))
                  ?spaceover $(remain ?posttags)))…)
```

Behind The Scenes



Looking at the pattern portion of the compositeleaf transform, we can follow thru the pattern matching. The pattern matching is a backtracking pattern matcher and thus, will find an instance of the pattern if one exists. Backtracking means that upon failure of any pattern element, the match will backup to the last choice point and restart the match by trying to find the next choice. The Pand operator requires that all of its subpatterns match the data. So, the first conjunct matches the leaf element and a, binding a to ?op. The second pattern spans over to the tags list binding (leaf a) to ?pre. It then matches the "(tags" and spans in the tags list over to the itype property binding ?itype to image. Everything before the itype pair (in this case nil) is bound to ?pretags, the ?spaceover variable absorbs the type pair, and the remainder of the list is bound to ?posttags.

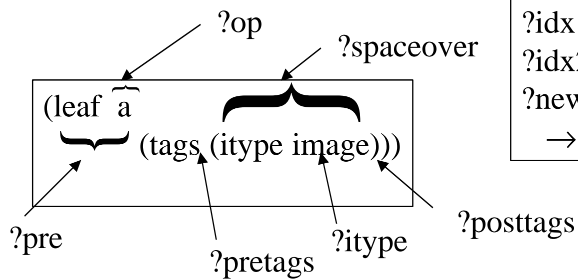
This second conjunct is really a little more complex than this since it has to account for the case where there is no tags list. I have omitted this level of complexity as it adds nothing to the discussion.

Pattern Match Result

$b=[(\boxed{a} \oplus s)^2 + (a \oplus s')^2]^{1/2}$

Define enablecompositeleaf (at, bindings)
{?idx1 = Create iterator; ?idx2 = Create iterator;
?newleaf = Create bwpixel; return extended bindings}

Behind The Scenes



?idx1 → idx2139
?idx2 → idx2145
?newleaf
→ bwpixel2136

After the successful pattern match, the preroutine (enablecompositeleaf) is called and it creates some additional bindings with some translator variables. It creates a unique black and white pixel object and a couple of iterators which may be used to scan over the image.

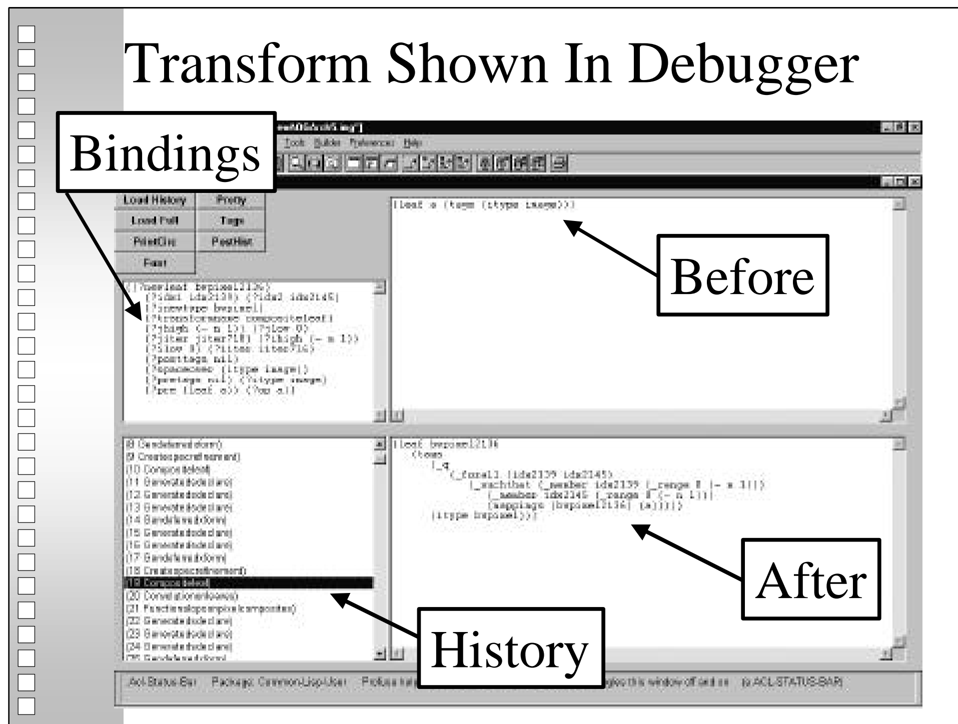
Rewrite Result

$b=[(\boxed{bwpixel2136} \oplus s)^2 + (a \oplus s')^2]^{1/2}$

```
`(leaf ?newleaf
      (tags (commasplice ?pretags)
            (_q (_forall (?idx1 ?idx2)
                         (suchthat (_member ?idx1 (_range ?ilow ?ihigh))
                                   (_member ?idx2 (_range ?jlow ?jhigh))
                                   (mappings (?newleaf) (?op)))))
                  (itype BWPixel) (commasplice ?posttags)))
```

```
(leaf bwpixel2136
  (tags (_q  (_forall (idx2139 idx2145)
                 (_suchthat (_member idx2139 (_range 0 (- m 1)))
                            (_member idx2145 (_range 0 (- n 1)))
                            (mappings (bwpixel2136) (a)))))
        (itype bwpixel)))
```

Now, AOG will create the rewrite expression using these bindings. The loop shorthand expresses the relationship among a, the bwpixel and the iterators. This will be moved up the expression tree and combined with other similar shorthands to define what looping structures can be shared.



This is a screen shot of the AOG debugger looking at an application of the compositeleaf rule. The transformation history is shown in the lower left panel and compositeleaf was the 19th transformation applied in this example. The full set of bindings created from the pattern match and the call to the preroutine are shown in the upper left panel. The before and after of the subtree are shown in the two right panels.

Speculative Refinement

- Dynamically build coordinated rules to express constraint set

$bwpixel2034 \Rightarrow b_{idx2037,\ idx2043}$

$idx2139 \Rightarrow idx2037$

$idx2145 \Rightarrow idx2043$

$bwpixel2987 \Rightarrow bwpixel2136$

$bwpixel2136 \Rightarrow a_{idx2037,\ idx2043}$

Speculative refinement is a process of dynamically generating and modifying rules (stored on generated symbols e.g., **bwpixel2034**) that AOG uses to create the proper map rules to coordinate the set of combination and merging decisions made incrementally as the expression tree was traversed. When executed in a follow-on phase called **SpecRefine**, these generated rules map away redundant symbols and map the remaining symbols to the proper target code symbols (e.g., array names and loop indexes).

This set of rules is an operational representation of the full set of constraints propagated over the DSL expression. These constraints were introduced either by the definitions of the larger data structures and operators, or introduce by control merging decisions made in the course of doing loop introduction and localization.

Thus, at the end of the loop localization phase all speculative refinement rules are coordinated to reflect the current state of localization combinations. The follow-on speculative refinement phase recursively applies any rules that are attached to AST abstractions (e.g., **bwpixel2034**). The result is a consistent and coordinated expression of references to common indexes, pixels, field names (e.g., **red**), and etc.

Explosion Control

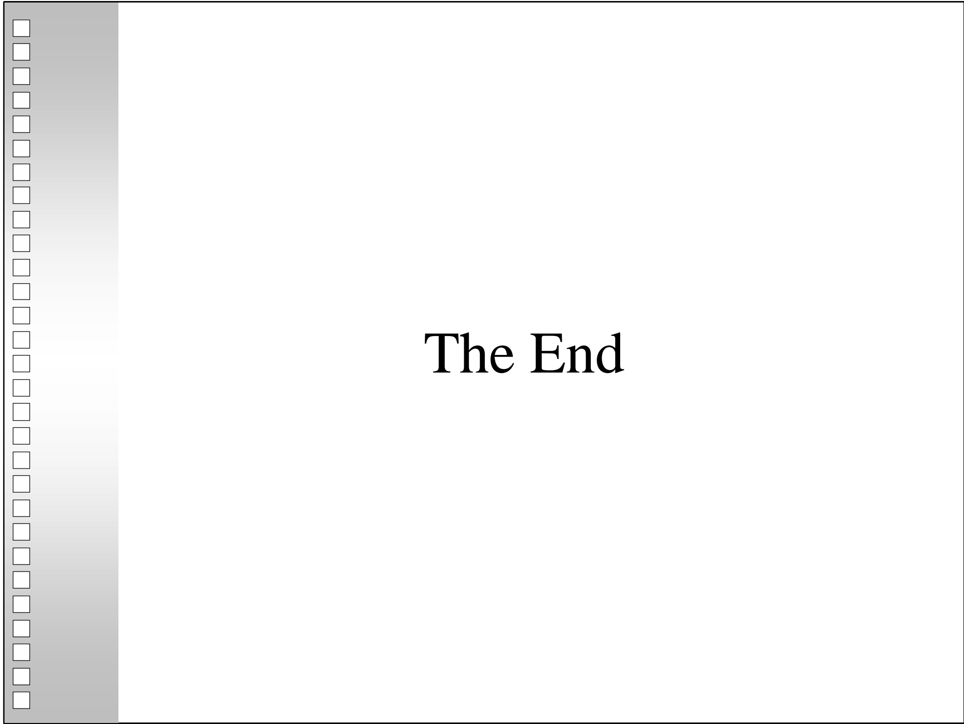
- DSL-> DSL translation phases
- DSL specific optimizations
- Simplification & DS Optimizations
  - Key Point: DS Reductions Not Combinations
- Localization
  - Separate Parts Woven into Computational Form
  - Group Rules by Object & Phase
  - Use Domain Knowledge
- Architectural Shaping
  - Interdependent, Composed Parts Changed per Global Constraints

So, we have examined a new kind of optimization (reorganization and weaving together of implied control structures) and looked at the machinery necessary to make this a tractable ability. This approach applies far more broadly than just loops in graphics domains. It can be adapted to other implied control structures in other domains (e.g., DBs, UI, Web, and middleware interfaces).

Now, let's look at a rather different kind of optimization and the machinery necessary to effect this. Often we have global constraints that may affect the structure of the computation as a whole without altering the computational results. The transformations that effect this kind of change will be called "architectural shaping" transforms. The complexity that this introduces is that such transformations must change in concert a number of interdependent pieces of code to accomplish the overall change in the computational structure. These interdependencies complicate such shaping.

- Katz & Volper, Constraint Propagation in Software Libraries of Transformation Systems, *IJSE&KE 2,3,* 1992.
- Biggerstaff, Fixing Some Transformation Problems, *Proc. Of Automated Software Engineering, 1999.*
- Biggerstaff, A New Control Structure for Transformation-Based Generators, *ICSR 2000.*
- Neighbors, Draco: A Method for Engineering Reusable Software Systems, in *Software Reusability, 1989.*

Katz, M. D. and D. Volper (1992), “Constraint Propagation in Software Libraries of Transformation Systems,” *International Journal of Software Engineering and Knowledge Engineering*, 2, 3


The End