Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 203 (2008) 37–51

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Implementing a Domain-Specific Language Using Stratego/XT: An Experience Paper

Leonard G. C. Hamey [1,2]

Computing Department
Macquarie University
Sydney, Australia

Shirley N. Goldrei [1,2]

Computing Department
Macquarie University
Sydney, Australia

Abstract

We describe the experience of implementing a Domain-Specific Language using transformation to a General Purpose Language. The domain of application is image processing and low-level computer vision. The transformation is accomplished using the Stratego/XT language transformation toolset. The implementation presented here is contrasted with the original implementation carried out many years ago using standard compiler implementation tools of the day. We highlight some of the unexpected advantages afforded to us, as language designers and implementers, by the source-to-source transformation technique. We also present some of the practical challenges faced in the implementation and show how these issues were addressed.

Keywords: Domain-specific language, transformation, compiler implementation, language definition, computer vision

1 Introduction

This paper describes the re-implementation of a Domain-Specific Language for low-level computer vision called *Apply*. This work contributes a reflection on the experience of using source-to-source transformation tools to implement a non-embedded Domain Specific Language. This work compares the implementation experience with that of more traditional compiler implementation techniques. Both implementations were carried out by the same developer. The present *Apply* implementation was carried out over a period of five months and this paper distils the experience

[1] We thank Alan Fekete, Tony Sloane and the reviewers for their constructive comments.

[2] Email: {len,shirley}@ics.mq.edu.au

doi:10.1016/j.entcs.2008.03.043

documented in a 68 page daily log maintained by the developer throughout the implementation process [2]. The log includes descriptions of progress made each day, language design thoughts, language implementation thoughts including ways of implementing optimisations of *Apply* programs, problems and solutions as well as documentation notes on Stratego/XT usage.

In terms of the implementation patterns for Executable DSLs identified by Mernik et. al. [6], the implementation technique chosen could be described as a *Hybrid* of *Compiler/application generator*, with considerable analysis and optimisation being done on the DSL program, and *Pre-processor using source-to-source transformation* to arrive at the base language source code.

The *Apply* language was originally implemented in the 1980's at The Robotics Institute at Carnegie Mellon University. The language was designed to allow easy and efficient programming of Low-Level Vision applications using the *Apply Programming Model* [4]. This model of programming reduces the problem of writing image-to-image vision applications, which are implicitly parallel computations, to the task of writing a procedure to be applied to a window around a single pixel of the image.

The original *Apply* compiler implementation focussed on the ability to rapidly re-implement the back-end of the compiler to target a range of hardware platforms including special purpose multiprocessor parallel architectures, bit-serial processor arrays, distributed memory architectures and uniprocessor systems. Each platform offered different programming environments: operating systems, programming languages and models of parallel computation.

Whilst the original design aims were achieved, recent hardware developments suggest it is time to re-implement the *Apply* compiler. The emphasis now is less towards targeting hardware infrastructure directly, and more towards implementing high-level source optimisations and rapidly targeting a range of image processing Application Programming Interfaces (APIs).

In the original implementation of *Apply*, Lex and Yacc were used to generate a parser which constructed an Abstract Syntax Tree (AST). Hand-written C code implemented the analysis and generation of appropriate target code depending on the target platform. For example, the compiler could generate W2 code for the Warp processor, or C code for a uniprocessor UNIX machine.

In the current implementation Stratego/XT [9] was chosen to implement the transformation of *Apply* source to C source. Stratego/XT provides a complete toolkit with domain specific language support for parsing, rewriting and pretty-printing. There was no need to resort to a general purpose language (such as C) for any part of the implementation. Even with the learning curve required by a novice user, only eight working days were required to implement the core of the *Apply* language and some basic optimisations [2].

Stratego/XT provides the capability to extend its rewrite language syntax with that of an object language [8] so that rewrite rules can be expressed in the concrete syntax of the language being transformed in addition to the abstract term representation. This significantly simplifies the task of translating and optimising the

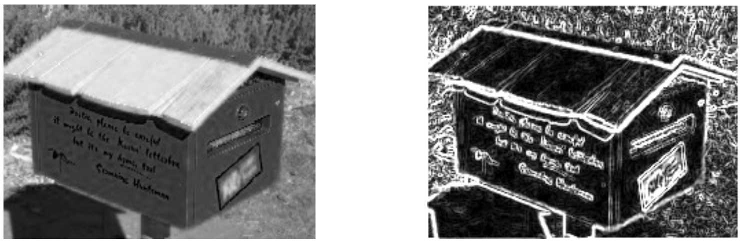

Fig. 1. Edge detection. Original image (left). Sobel edge detected image (right).

Apply source.

The use of concrete syntax facilitated optimisations that would not have been feasible in the C version of the compiler. This encouraged the development of the *Apply* language to further enhance its expressivity without fear of sacrificing application performance. It also made it possible to compile the base language for more efficient execution on a uniprocessor machine than could have been achieved with the original design.

Using concrete syntax in the transformation description makes it a very straightforward task to replace target code utilising one API with code utilising another API. It only requires re-implementing one small Stratego module that transforms specific aspects of the code and the entire *Apply* compiler can be regenerated. This makes it feasible to use the *Apply* language as a way of succinctly specifying low-level vision processing algorithms in an implementation neutral way, allowing the establishment of reusable and retargetable libraries of such algorithms.

In the sections that follow we provide an outline of the domain of application. We introduce the *Apply* language and domain specific features and discuss the influence that using Stratego/XT has had on the language design. Then we discuss the implementation of the current *Apply* compiler and reflect on its development with reference to the original C-based implementation.

2 Low-Level Vision and the *Apply* Language

To motivate and illustrate the implementation design we begin by describing our domain of application and our transformation's source language *Apply*.

Low-Level Vision involves processing image colour and contrast information at a pixel level to identify features such as edges or corners of objects, ridges or blobs that can be used to identify objects or track moving objects. Figure 1 shows an example of an image and the output image after processing for edge detection using a Sobel operator [1, pp 418-420].

There are a large number of known algorithms for processing images and detecting features, and developing new algorithms continues to be an active research area. Kernel-based algorithms are one common class of global algorithms where each output pixel is computed from a small neighbourhood of the input. These algorithms can be described simply and concisely using *Apply*. The language allows the algorithm designer to focus on the details of the computation that will apply to a

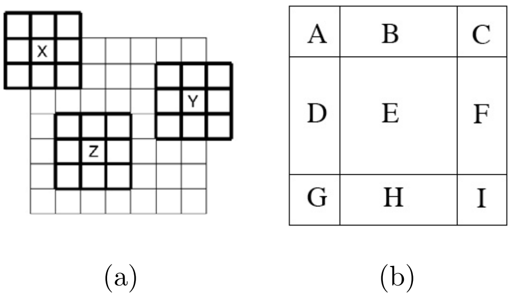

Fig. 2. (a) Illustration of applying a kernel computation at the corner of an image (X), edge of the image (Y) and clear of the border (Z). (b) Nine regions of an image, labelled A-I, each requiring different bounds checking code.

window of pixels surrounding a single pixel. For example, Figure 2(a) illustrates the application of a 3 x 3 kernel to an image at three different pixels. *Apply* abstracts away entirely the details of repeatedly applying a computation across an entire image. The *Apply* compiler generates the looping structures, handles the exceptional cases at the border of the image, deals with the internal representation of the image (which changes from system to system and API to API) and generates code to take advantage of the parallel computational capability of the target platform.

To illustrate the abstraction power of *Apply* consider the Sobel operator [1, pp 418-420] which detects the edge of objects by comparing the intensity of a pixel to its eight immediate neighbours. The algorithm used to apply this operator to vertical edges across the image requires looping over the rows and columns of the image and at each pixel multiplying, entry-wise, a matrix $K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ with an equally sized window of pixels taken from the image and centred around the 'current' pixel. The sum of the elements in this matrix of products is the vertical component of the output value for the current pixel location. This type of calculation is known as 'convolution' in image processing. The process is repeated with a different matrix K for horizontal edges and the horizontal and vertical components are combined by adding their absolute values. Pixel values greater than a byte are truncated.

A hand-written Sobel operator must implement the loop that traverses the appropriate data-structure containing the image, and handle the behaviour of the algorithm at nine distinct regions to ensure correct computation at the corners and edges, see Figure 2(b). The C code for this operator, using two loops, is given in Figure 3. Using only one set of nested loops produces unacceptably slow code with some C compilers, while nine loops often perform best. The implementation in Figure 3 assumes that the images are each stored in a single array in row-major order and that the computations are being performed on a standard uniprocessor machine and no truncation of byte values is being done. In comparison Figure 4 shows the code that would be written in *Apply* to achieve a similar computation with byte truncation as well. The *Apply* compiler can of course generate a nine loop version automatically.

```
/* in_bounds returns true when window pixel coordinates (i,j) fall inside     */
/* the bounds of the input image. The current pixel position is (row,column)  */

#define in_bounds(i,j) \
   (row+i >= 0 && row+i < height && column + j >= 0 && column + j < width)


/* FROM(i,j) computes access to 1-dimensional array "from" for window */
/* pixel coordinates (i,j) and current pixel position (row,column) */

#define FROM(i,j) from[(row+i)*width + column + j]


/* FROM_R(i,j) returns input pixel at window coordinates (i,j) if the */
/* coordinates are in bounds, otherwise return zero */

#define FROM_R(i,j) (in_bounds(i,j) ? FROM(i,j) : 0)


/* TO computes access to 1-dimensional array "to" for current pixel */
/* position (row,column) */

#define TO to[row * width + column]

void sobel(unsigned char from[], unsigned char to[],
                                 int height, int width) {
        int row;
        int column;
        int x, y;

        /* Process the window positions that overlap the edge/corners */
        /* of the input image. Bounds checks are performed on all input */
        /* window accesses */

        for (row = 0; row <= height - 1; row++) {
                for (column = 0; column <= width - 1; ) {
                        x = FROM_R(-1,-1) + 2 * FROM_R(-1,0) + FROM_R(-1,1) - FROM_R(1,-1) -
                                FROM_R(1,1) - 2 * FROM_R(1,0);
                        y = FROM_R(-1,-1) + 2 * FROM_R(0,-1) + FROM_R(1,-1) - FROM_R(-1,1) -
                                FROM_R(1,1) - 2 * FROM_R(0,1);
                        if (x < 0) x = -x;
                        if (y < 0) y = -y;
                        TO = x + y;
                        if (column == 0 && row > 0 && row < height-1) {
                            column += width-1;
                        } else {
                            column++;
        }       }       }

        /* Process the window positions that do not overlap the */
        /* edges/corners of the input image. Access to input pixels */
        /* are sped up because there is no need for bounds checking. */

        for (row = 1; row < height - 1; row++) {
                for (column = 1; column < width - 1; column++) {
                        x = FROM(-1,-1) + FROM(-1,1) + 2 * FROM(-1,0) - FROM(1,-1) -
                                FROM(1,1) - 2 * FROM(1,0);
                        y = FROM(-1,-1) + 2 * FROM(0,-1) + FROM(1,-1) - FROM(-1,1) -
                                FROM(1,1) - 2 * FROM(0,1);
                        if (x < 0) x = -x;
                        if (y < 0) y = -y;
                        TO = x + y;
}       }       }
```

Fig. 3. C code for computing Sobel edge detection.

3 *Apply* Language Design and Stratego/XT Influence

The syntax of *Apply* is based on a subset of Ada [5]. In summary, *Apply* provides the following language features which are similar to Ada: arithmetic and boolean expressions; control flow structures: `if`, `if else`, `while` and `for`; primitive datatypes: `byte`, `real` and `integer`; multidimensional `array` types with index ranges; procedures with formal parameters each marked with one of the access modes `in`,

```
procedure sobel(from: in window (-1..1, -1..1) of byte border 0,
                               to: out window of byte)
is
  x, y : integer;
begin
  x := from(-1,-1) + 2 * from(-1,0) + from(-1,1) - from(1,-1) -
         2 * from(1,0) - from(1,1);
  y := from(-1,-1) + 2 * from(0,-1) + from(1,-1) - from(-1,1) -
         2 * from(0,1) - from(1,1);
  if x < 0 then x := -x; end if;
  if y < 0 then y := -y; end if;
  x := x + y;
  if x > 255 then x := 255; end if;
  to := x;
end sobel;
```

Fig. 4. *Apply* code for computing Sobel edge detection.

`out` or `in out`. Special features available in *Apply* include an abstract data type, called `window` which can only be used as the type of a formal parameter of a procedure. A `window` can either hold a single element of a specified primitive data-type or a two dimensional array of a specified primitive data-type with index ranges. Formal parameters of this type are declared with the syntax `window of` ***`Type`*** or `window(` ***`Range,Range`*** `) of` ***`Type`*** `border` ***`expr`*** for scalar or subscripted instances respectively. The `border` ***`expr`*** modifier in the declaration is a succinct way of defining how windows should be handled at the edges of the image. When a location of the window falls outside the image (as it would at the edges) the constant expression given is substituted for the value that would otherwise have been taken for the image itself. Often this constant expression is zero.

The current implementation of *Apply* defines a number of new metaprogramming style language extensions. These extensions serve one of two purposes. Some extensions are used as hints to the compiler indicating possible optimisations. Other extensions make it easier to write the Stratego rules that will generate the code to target specific APIs. These extensions were not available in the original *Apply* definition, they have been added to the language as a direct result of the experience of using transformation techniques for the compiler implementation. Examples of these extensions include `@known` expressions, `@apply` statements, `defined` expressions and `assert` statements.

The language extensions beginning with an '`@`' character were intended to be used by the language implementers and those implementing specialised transformations for target APIs or environments. They are used as an intermediate representation of the semantics of constructs that do not appear in the domain of the application programmer, but do appear in the implementation of the language. They are added as syntactical elements (rather than purely abstract terms) to make the transformations easier to both read and write for the language implementers. This technique takes advantage of the ability to extend the Stratego language with object syntax. In this case the object syntax is neither strictly the source nor the target language.

The meaning of `@known(`*`expr`*`)` is as follows: if *expr* evaluates at compile time to true then the whole expression is replaced by the value `true` otherwise if the expression either evaluates to `false` or can not be evaluated to a constant value at all, the whole expression is replaced by `false`. In conjunction with `if` and `if else` statements, `@known` can be used to provide the compiler with alternate algorithms. The compiler's standard unreachable code elimination techniques replace the complete branch statement with a single algorithm at compile time. To eliminate a possibly unnecessary modulo computation one would write:

```
if (@known(row < 255))
  x:=row;
else x:=row % 255;
```

If the programmer knows a property of a variable but the compiler could not be expected to prove it, the programmer can assert this knowledge using the `assert` *`expr`* statement e.g. `assert x>=2;`. The compiler exploits this knowledge to optimise code generation wherever possible. This statement could also be used to generate runtime checks, however currently this is not done.

The `defined(`*`id`*`(`*`expr`*`,`*`expr`*`))` expression tests whether the access to a particular pixel of a window is currently defined. For example, consider the window `from` defined in Figure 4. When processing the top left corner as illustrated in Figure 2 (a) in position X, the window access will be defined at `from(0,0)`, `from(0,1)`, `from(1,0)` and `from(1,1)` and undefined at, for example, `from(-1,-1)` and all other locations. The `defined` construct flexibly and simply expresses border handling for low-level vision algorithms and is used to implement the `border` *`expr`* modifier.

Originally `@defined` and `@assert` were added to simplify implementation, however while developing the compiler it was decided that it would be beneficial to allow domain application programmers to make use of assertions and the defined construct. The '`@`' was dropped from the syntax and they became formal parts of the language. The `@known` is another construct that was originally devised to simplify implementation, however it too is likely to be added to the *Apply* language because of the power it affords when combined with automatically generated `assert` statements. For example, since the purpose of the *Apply* language is to take a computation and to "apply" it across an image, the computation (on a uniprocessor machine) is placed inside several loops and the *Apply* compiler automatically generates assertions based on these loops. In particular the loop `for row 1..100 do loop` would generate the assertions `row >= 1` and `row <= 100`. If the earlier `@known` example were included in the computation the *Apply* compiler would be able to determine that `row < 255` was true and would be able to substitute the simpler code without the modulo operation.

Using Stratego it was easy to add language features and try them out. Changing the syntax would have been trivial even in the original implementation, but adding these features to the hand-written compiler was non-trivial and so this was never attempted.

4 Implementing *Apply* in Stratego/XT

In this section we give an overview of the implementation of the current *Apply* compiler. We highlight and illustrate how Stratego/XT was used effectively to achieve the design goals of both the language and the compiler.

Our compiler transformation consists of the following series of smaller transformations.

Desugar This stage transforms Ada style multivariable declarations into a sequence of single variable declarations.

Simplify This stage generates 'expert' assertions from the programmer's original `if`, `if-else` and `for` statements. This stage then propagates assertions, including programmer written assertions, evaluates `@known` expressions and performs constant folding and propagation and unreachable code elimination. To enable assertions to be propagated and further simplify expressions, this stage also rewrites all expressions into a canonical form.

Do Apply This and the following stage implement the heart of the *Apply* language. This stage inserts an abstract `@apply` loop which will be specialised for a target API in the next stage. This stage also uses typing information to recognise window accesses and convert them from a generic 'function call' representation to an abstract subscript operation.

Image Matrix This stage expands the abstract loop of an *Apply* procedure to add all the necessary looping over an image. This entire transformation is given in a single shorthand concrete syntax to make it easy for library developers, or users wishing to interface to new libraries, to replace the existing code with code for their library. Similarly, kernel accesses are converted to the target API and the `defined` pseudo-function is converted to a suitable implementation. This stage also generates specialised loops with constant values for image widths. The widths are made available to the compiler user as command line switches on the compiler. Loops generated with constant values for image widths run considerably faster in some circumstances.

Simplify The Simplify stage is repeated to simplify expressions that may have been added during the Image Matrix stage.

Ensugar This stage transforms *Apply* constructs such as function arguments, declarations and incrementing variables into their C equivalents.

Apply Paren This stage adds parentheses as necessary to expressions to maintain the correct precedence on output to C text.

C Pretty Print This final stage formats the output C source code.

Many of these stages are largely the same as they would be for most Algol-like languages and the example code provided for Stratego/XT[3] was used as the basis of the *Apply* implementation. In particular implementing data-flow analysis, such as needed for constant propagation and unreachable code elimination, at the source code level is described in [7]. However some of the steps of transformation are unique to the *Apply* language and the design goals of its compiler and therefore deserve further discussion.

In the syntax of the *Apply* language, applied occurrences of variables of type `window` are indistinguishable from procedure calls. The Do Apply stage of the transformation uses the typing information to identify the applied occurrences of `window` variables and replaces them with `WindowAccessScalar` or `WindowsAccessElement` abstract syntax nodes for scalar and indexed variable respectively. The Do Apply stage also wraps the entire procedure body in the `@apply` language extension which represents a looping structure. The complete syntax for this extension is:

```
@apply LoopType window( Exp ..  Exp, Exp ..  Exp )
loop
        Statements
end loop;
```

Here the *LoopType* indicates whether or not the application programmer's code uses the row or column variables. If not, the compiler has greater freedom in the selection of looping structures and may even use a single loop treating the entire image as a single dimensional array. The window given in the `@apply` statement is computed from all the windows given as formal parameters to the enclosing procedure and represents the extreme dimensions of the combined windows.

This abstract representation reveals that the procedure body is to be applied in a loop across the entire image. This abstraction becomes concrete in the stage known as 'Image Matrix'. Figure 5 shows one such concretisation. There are in fact other implementations, with different performance characteristics, that can be selected with command line switches at *Apply* compile time. The rewritten `@apply` construct makes use of extensions that were added to the concrete syntax to aid readability. These include the `@cfor` and `@:=` constructs which together mimic a C-style `for` loop. This intermediate representation and transformation is what enables us to achieve the design goal of easily retargeting alternate APIs and target environments. The 'Image Matrix' stage and, in particular, transformations such as the one shown in Figure 5 are where changes are made to retarget the compiler. Figure 6 illustrates an implementation which would target a parallel architecture where `cpucount` contains the number of available CPUs and `cpunumber` contains the CPU ID number that the particular code is being executed on. The language extensions `@setup_buffers`, `@getrow` and `@putrow` are abstract representations of the specific API calls that read and write images to and from buffers and would be expanded by other rules in the Image Matrix stage.

Figure 5 also illustrates the use of 'expert' inserted `assert` statements. These assert the possible values of the automatically pre-declared variables `column` and `row`

[3] Stratego/XT is available from http://www.program-transformation.org/Stratego

```
FixLoop2NoIndex :
 |[ @apply ~looptype window (i1..i2,j1..j2) loop ~s end loop; ]| ->
 |[ for row in 0..height-1 loop
     @cfor column @:= 0; column <= width-1; loop
         assert column >= 0 and column <= width - 1;
         assert row >= 0 and row <= height-1;
         ~s
         if column = -j1-1 and row >= -i1 and row < height - i2 then
            column := column + width - j2 + j1 + 1;
         else
            column := column + 1;
         end if;
     end loop;
    end loop;
    for row in -i1..height-i2-1 loop
      for column in -j1..width-j2-1 loop
        ~s
      end loop;
    end loop; ]|
```

Fig. 5. The heart of the *Apply* language implementation transforms an abstract looping construct into a concrete implementation.

```
FixLoop2NoIndex :
  |[ @apply ~looptype window (i1..i2,j1..j2) loop ~s end loop; ]| ->
  |[ @setup_buffers(i1,i2,j1,j2);
      for row in 0..height-1 loop
       @getrow(row);
        for column in
         width*cpunumber/cpucount .. width*(cpunumber+1)/cpucount-1
       loop
         ~s
       end loop;
       @putrow(row);
      end loop; ]|
```

Fig. 6. Sketch of an *Apply* loop transformation that targets a parallel architecture with abstract API calls to manage images.

based on the loop invariants. Constant folding and propagation and unreachable code elimination can then simplify the statements inside the loops, with the benefits of readability and execution efficiency.

5 Reflections on Using Stratego/XT Versus a GPL

The log maintained during the development of the current *Apply* compiler [2] prompted reflection on the comparative experiences of this development versus the experience of developing the original compiler. Since no such log was maintained during the development of the original compiler the source code and its documentation was used for comparison.

In some respects the approach taken in the original implementation mirrored the approach of transformation through walks or traversals of the abstract syntax tree, pattern matching and replacement of old tree nodes with new sub-trees. However instead of having a succinct purpose-designed notation these tasks were achieved using calls to hand-written libraries. Figure 7 shows a small fragment of the code used to implement subscript translation in the main apply loop in the original C implementation. The function `walk` (and its variants) performs a matching tree walk starting at the current tree node. Since each tree node had a number of potential arguments (although only two were used in most circumstances), the walk functions specify which argument to follow, then the expected node type (or 0 if any node is acceptable); `walk2` does two walks; `walk3` does three. Failed walks return `PT_NIL`. The function `nodei` constructs a node that has an integer value and the function `node2` constructs a tree node with two children. Note that the C code requires considerably more documentation in the form of comments in order to make the intention of the code clear.

In comparison the Stratego/XT code shown in Figure 8 handles the same transformation not just for one variant of the `window` construct but for subscripted as well as scalar variants, with or without the `border` modifier and with row/column indexing or with direct indexing. This code requires no additional commenting since the semantics of transformation are defined by the Stratego/XT language itself and tree walking details are implied.

Similarly, the original compiler's minimal optimisations were limited to constant folding since the amount and complexity of the code needed in C was too great. Stratego was selected for the re-implementation to enable more sophisticated code manipulation to target more difficult environments (such as processing the image in place and therefore having to handle the border as part of the code). We had reached the limit of our ability to manage the complexity of the parse tree manipulation process expressed directly in C code.

It is difficult to give an accurate measure of the relative complexity or effort involved, however to give a rough guide, and in the absence of accurate records of the time taken to program the original compiler, we compared the physical commented lines of code (LOC). We also counted non-commented code and present that as a proportion of the total. See [3] for further discussion.

The current implementation required significantly fewer lines of code - see Figure 9. The analysis and transformation stages for the original C implementation were smaller, however the current implementation does significantly more code manipulation and optimisation. The original compiler included a very simple C module for performing constant folding which handled 8 arithmetic operators computing constant values only when the arguments to the operator were themselves constant terms. By comparison constant folding in the current compiler handles 15 operators including boolean operators. This code more aggressively simplifies expressions, rearranging them as necessary to bring constant terms together and removes operator

```
...
 if (tree->type == PT_SUBSCRIPT && mode == MODE_STMTS)
  { register pt_node *var, *sub1, *sub2;
    register declaration *d;
    /*  Match a piece of tree that looks like
     *      PT_SUBSCRIPT -- PT_SUBLIST -- PT_SUBLIST
     *           |              |              |
     *          var            sub1          sub2       */
    var = walk (tree, ARG1, PT_VARIABLE);
    sub1 = walk2 (tree, ARG2, PT_SUBLIST, ARG1, 0);
    sub2 = walk3 (tree, ARG2, PT_SUBLIST, ARG2,
                  PT_SUBLIST, ARG1, 0);

    if (var != PT_NIL && sub1 != PT_NIL && sub2 != PT_NIL &&
          var->decl)
    {
      d = var->decl;
      if ((d->classtype == IN || d->classtype == OUT) &&
           d->numdim == 2)
      {
         t1 = nodei (d->dimensions[1].high -
                            d->dimensions[1].low + 1);
         t2 = nodei (d->dimensions[1].low);
         t3 = nodei (d->dimensions[0].low);
         /* Now set t1 to entire expression
          *      PT_PLUS  --  PT_MINUS  --  t3(minrow)
          *         |             |
          *         |           sub1(i)
          *         |
          *     PT_MULTIPLY --  t1(maxcol-mincol+1)
          *         |
          *      PT_PLUS  --  unrollid
          *         |
          *      PT_MINUS  --  t2(mincol)
          *         |
          *       sub2(j)      */
          t1 = node2 (PT_PLUS,
                     node2 (PT_MULTIPLY,
                       node2 (PT_PLUS,
                             node2 (PT_MINUS, sub2, t2),
                             nodei (unrollid)),
                       t1),
                     node2 (PT_MINUS, sub1, t3));
         /* Now put expression into subscript list in
             tree and prune list. */
         sub1 = walk (tree, ARG2, PT_SUBLIST);
         sub1->arg[ARG1] = t1;
         sub1->arg[ARG2] = PT_NIL;
         return (tree);               /* Prune recursion. */
...
```

Fig. 7. C code to replace window relative indexes with image relative indexes.

```
FixWindowsIndexBorder :
 WindowAccessElement(x,row,col,rowrange,colrange,Border(t,e),
          looptype) ->
   IfElseExp( |[ defined(x( ~row, ~col)) ]| ,
     Subscript(x, |[ ~row * width + app_index + ~col ]| ), e)
FixWindowsIndex :
 WindowAccessElement(x,row,col,rowrange,colrange,type,looptype) ->
   Subscript(x, |[ ~row * width + app_index + ~col ]| )
FixWindowsNoIndex : WindowAccessScalar(x,type,looptype) ->
   Subscript (x, |[ row * width + column ]| )
FixWindowsNoIndexBorder :
 WindowAccessElement(x,r,c,rowrange,colrange,Border(t,e),
          looptype) ->
   IfElseExp( |[ defined(x( ~r, ~c)) ]| ,
     Subscript(x, |[ (row + ~r) * width + column + ~c ]| ), e)
FixWindowsNoIndex :
 WindowAccessElement(x,r,c,rowrange,colrange,type,looptype) ->
   Subscript(x, |[ (row + ~r) * width + column + ~c ]| )
```

Fig. 8. Stratego/XT rewrite rule: window relative indexes to image relative indexes.

Phase	Original Compiler		Current Compiler	
	LOCC	% Comments	LOCC	% Comments
lexical & parsing	1023	27%	280	25%
analysis & transformation	1351	36%	1897	27%
constant folding	77 (8)	32%	76 (15)	38%
(number of operators)				
format target language	306	34%	144	9%

Fig. 9. Physical commented LOC with % of comments and blank lines to total lines.

identities.

Needless to say learning a new tool set involves a learning curve. The log documents many incidents where the first attempt at achieving a desired outcome using Stratego did not work and further reading, learning and trial and error testing was needed. Stratego's terse syntax was the cause of confusion on a number of occasions: everything from strategy combinators to parameterised rules and matching terms. Using Stratego effectively often requires an unfamiliar functional programming style of problem solving. For example, to compute the extreme dimensions of all windows given as arguments to an *Apply* function the log records requiring a different way of thinking:

"Rather than thinking about it as passing through the tree and accumulating the most extreme dimensions, I need to think about reducing the tree to the most extreme dimensions by rewriting it. Then, I can use `where` to assign the results to a temporary variable inside a strategy."

The reverse production format of the Stratego Syntax Definition Formalism (SDF) posed difficulties at first, as did identifying sources of ambiguities in the SDF grammar definition and correcting them. Being forced to re-specify the grammar represented a barrier to entry, but it facilitated the integration between concrete and abstract syntax. Overall productivity was greater than before, primarily due to the domain specific nature of Stratego over C.

Debugging was difficult during both implementations as the only method was to dump the parse tree or insert debugging statements in the code. With Stratego we can separate transformation stages into separate executables, to test isolated stages on simple AST fragments, but it's not easy to see which of Stratego's dynamic rules are active except by their effect when executed.

6 Performance

We found it easy to implement optimisations in Stratego. Is it better to optimise the generated source rather than leave optimisation to the target C compiler? After constant folding, constant propagation and unreachable code elimination the C code was succinct, more readable and thus easier to verify by inspection than un-optimised source code.

We compared the performance of the Sobel operator written in *Apply* (see Figure 4) with the Sobel operator written by hand similar to that shown in Figure 3, using 512 x 512 pixel images. The *Apply* generated code was specialised with a constant image width as discussed in section 4. We also compared the performance of the C code generated by the old compiler with that of the current compiler. The tests were run with five combinations of CPU and C compiler. For each C compiler a range of common optimisation options were tested with the best option used for comparison for each compiler on each platform. See [3] for more details. Each measurement is the median time over 7 runs each involving 30 seconds of execution time with randomised array locations to avoid the impact of cache and paging.

The results in Figure 10 show that the Apply generated code ran up to 37% faster than the hand-written code. The results vary greatly between hardware platforms and compilers. Tests were also performed with operators that had simpler code. The speed-up achieved by the Apply generated code depended on the complexity of the Apply program, but there was no consistent trend across the platforms. We conclude that source-to-source optimisation, using transformation techniques including generating specialised code, is worthwhile for readability and verifiability and sometimes for performance reasons.

	Core-gcc Core-gcc	PC-gcc PC-gcc	SPARC -gcc	Core -MSVC	PC -MSVC
Hand-written	5.232	3.608	4.668	5.190	4.070
Old compiler	4.429	3.438	6.366	4.450	4.6
Speedup	15%	5%	-36%	14%	-13%
New *Apply* compiler	3.283	2.462	4.596	3.940	3.97
Speedup	37%	32%	2%	24%	2%

Fig. 10. Summary of execution times and speedup for a Sobel operator.

7 Conclusion

We described the experience of implementing a Domain-Specific Language for image processing and low-level computer vision, comparing our experience of using Stratego/XT with the experience of using traditional techniques. The Stratego/XT toolset enabled easy implementation and provided opportunities to enhance the language and improve the performance of generated code. The use of concrete syntax in the transformations facilitated rapid retargeting to exploit available APIs and target environments. Our experience demonstrates that implementing a compiler by transformation to a GPL is a practical way of achieving a non-embedded DSL.

References

[1] Gonzalez, R. C. and R. E. Woods, "Digital Image Processing," Addison-Wesley, New York, 1992.

[2] Hamey, L., *Re-implementation of Apply: Stratego/XT experience notes* (2006), unpublished.

[3] Hamey, L. G. C. and S. N. Goldrei, *Implementing the Apply compiler using Stratego/XT*, Technical Report C/TR07-01, Department of Computing, Macquarie University, NSW, Australia (February 2007).

[4] Hamey, L. G. C., J. A. Webb and I.-C. Wu, *Low-level vision on Warp and the Apply programming model*, Parallel computation and computers for artificial intelligence (1988), pp. 185–199.

[5] Ledgard, H., "Reference Manual for the ADA Programming Language," Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[6] Mernik, M., J. Heering and A. M. Sloane, *When and how to develop domain-specific languages*, ACM Computing Surveys **37** (2005), pp. 316–344.

[7] Olmos, K. and E. Visser, *Strategies for source-to-source constant propagation*, in: B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, Electronic Notes in Theoretical Computer Science **70** (2002), p. 20.

[8] Visser, E., *Meta-programming with concrete object syntax*, in: D. Batory, C. Consel and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, Lecture Notes in Computer Science **2487** (2002), pp. 299–315.

[9] Visser, E., *Program transformation with Stratego/XT*, Technical Report UU-CS-2004-011, Institute of ICS Utrecht University (2004).
URL http://www.stratego-language.org/Stratego/ProgramTransformationWithStrategoXT