Optimization of Imperative Functional Parallel Programs with Non-local Program Transformations

Alexei Adamovich

PSI RAS, Pereslavl-Zalessky

META 2012

Plan of presentation

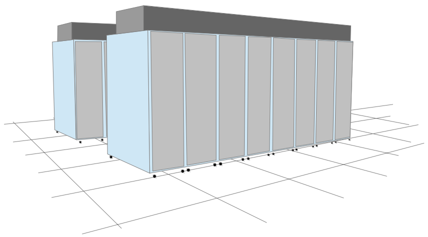
- Introduction (background)
- The toolchain for optimization of imperative functional programs
- Sample algorithm of program transformation
- Final part: conclusions, thanks etc

Plan of presentation

- ***Introduction (background)***

Parallel Revolution

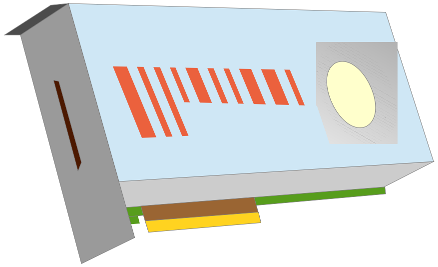
They are already here:



- **high-performance parallel computers (clusters)**

- **multicore desktops**

- **many-core accelerators**



Are we prepared?

T-System Approach

- Is under development since early 90-th in the Research Centre for Multiprocessor Systems of Program Systems Institute in Pereslavl-Zalessky under the leadership of Sergei Mikhailovich Abramov
- Can be viewed as a particular implementation of the Parallel Functional Programming paradigm
- Several different successful implementations were available for the computational clusters and SMP servers/desktops

T-System Approach II

- Capitalizes on the inherent properties of the functional programming: the independent calls of pure functions (the T-functions in case of T-System) can be computed in parallel, e.g.:

`F(G(x),H(x))`

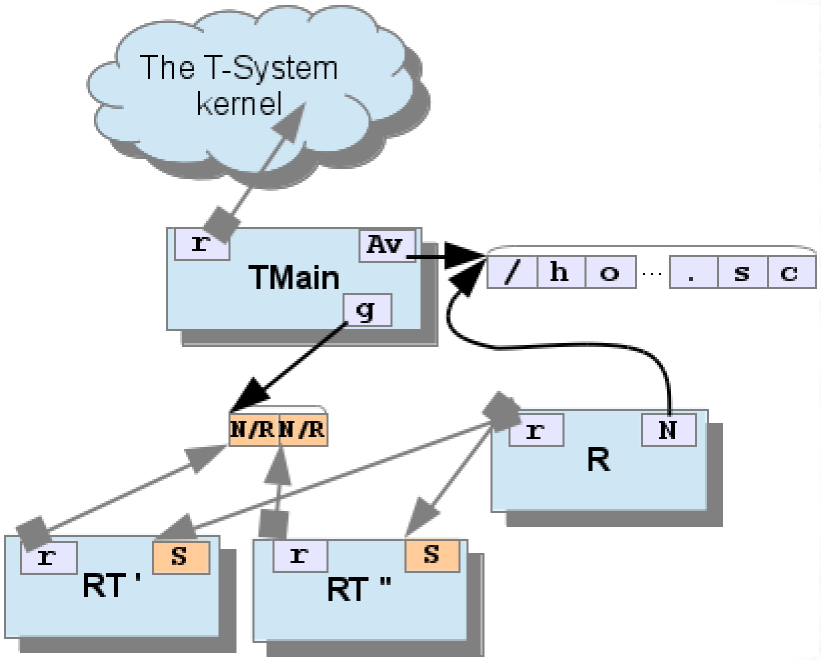
- Uses non-standard operational semantics for the calls of T-functions:
  after the call of a T-function all results of this call are assigned with non-ready (non-evaluated) values and computation of the initial T-function (caller) can be continued. So in the example above functions F, G, H and the caller of the function F (4 functions total) can be under the process of computation in parallel up to the moment when the value of any variable that is still non-ready will be really needed for the computations (not for assignments)

T-System Approach III

- Allows the bodies of T-functions be programmed in traditional *imperative* style (in particular, calls to a usual C functions are can be used). But several restrictions are applied to not permit the side effects get out of the T-function borders to influence the other calls
- The information is supplied to a call of a T-Function only via its arguments
- Each of the results can be returned by a T-Function call only via special primitive (SEND)
- Special flavor of variables (the T-variables) are introduced to hold non-ready values

T-System Approach IV

- In the course of parallel execution of a program each of the performed calls of a T-function becomes a lightweight thread of control (so called T-process)
- T-processes and the T-System data together form a network that is being self-transformed during the execution of T-processes
- The process of execution of an application in the whole starts with the T-function named TMain



A sample network of a T-System processes and data

Plan of presentation

- Introduction (background)
- **The toolchain for optimization of imperative functional programs**

ACCT

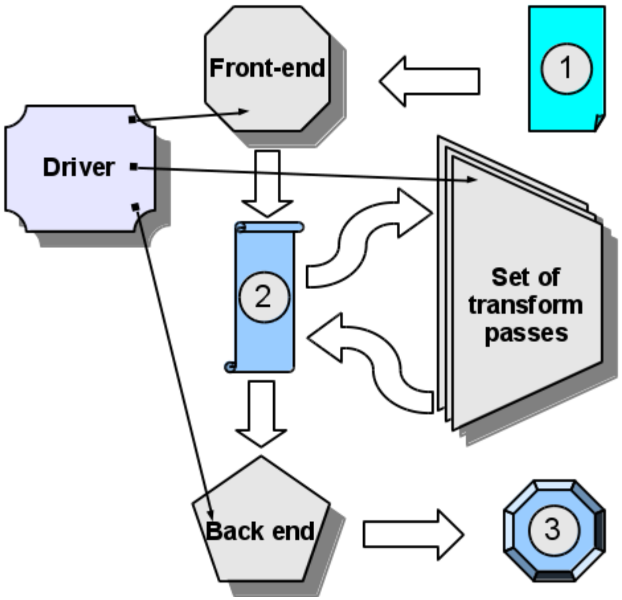
The compilation and transformation framework for the T-System programs

- Is intended for to allow to analyze the T-System programs and execute optimizing transformations of their intermediate representation
- The input language is an extended restriction of the C programming language (the cT)
- The development is still not completed

ACCT II

Architecture

- Main components: *front end*, a set of *transform passes*, and *back end*
- ***Front end*** transforms the program module from an input language into an intermediate representation (IR). After the transformation is complete, the IR obtained as a result is stored in a separate file or special program library.
- Each ***transform pass*** is able to transfer IR from the file or program library into RAM and somehow modify it. After that, a new version of IR is stored back on the external storage. Since IR of all application modules is potentially available to the transform pass, the performed transformations have a possibility to rely on the use of complete information about the application code as a whole



1. **Source code**
2. **The library of intermediate representations**
3. **The output (assembler, C etc.) file**

(continued on the next slide ...)

ACCT III

Architecture (continued)

- ***Back end*** reads IR from the file or program library and forms the resulting assembly (or C) code for further transformation into an executable program
- There also exists a ***compiler driver*** – a control program, which is needed for to call all the passes described above in the proper order and with the proper arguments

A similar structure of compiling systems is used in a number of program transformation systems, such as ***SUIF***, ***LLVM***, ***OPS***, etc.

The ACCT implementation is heavily based on the C front end of the ***GCC*** compiler.

(continued on the next slide ...)

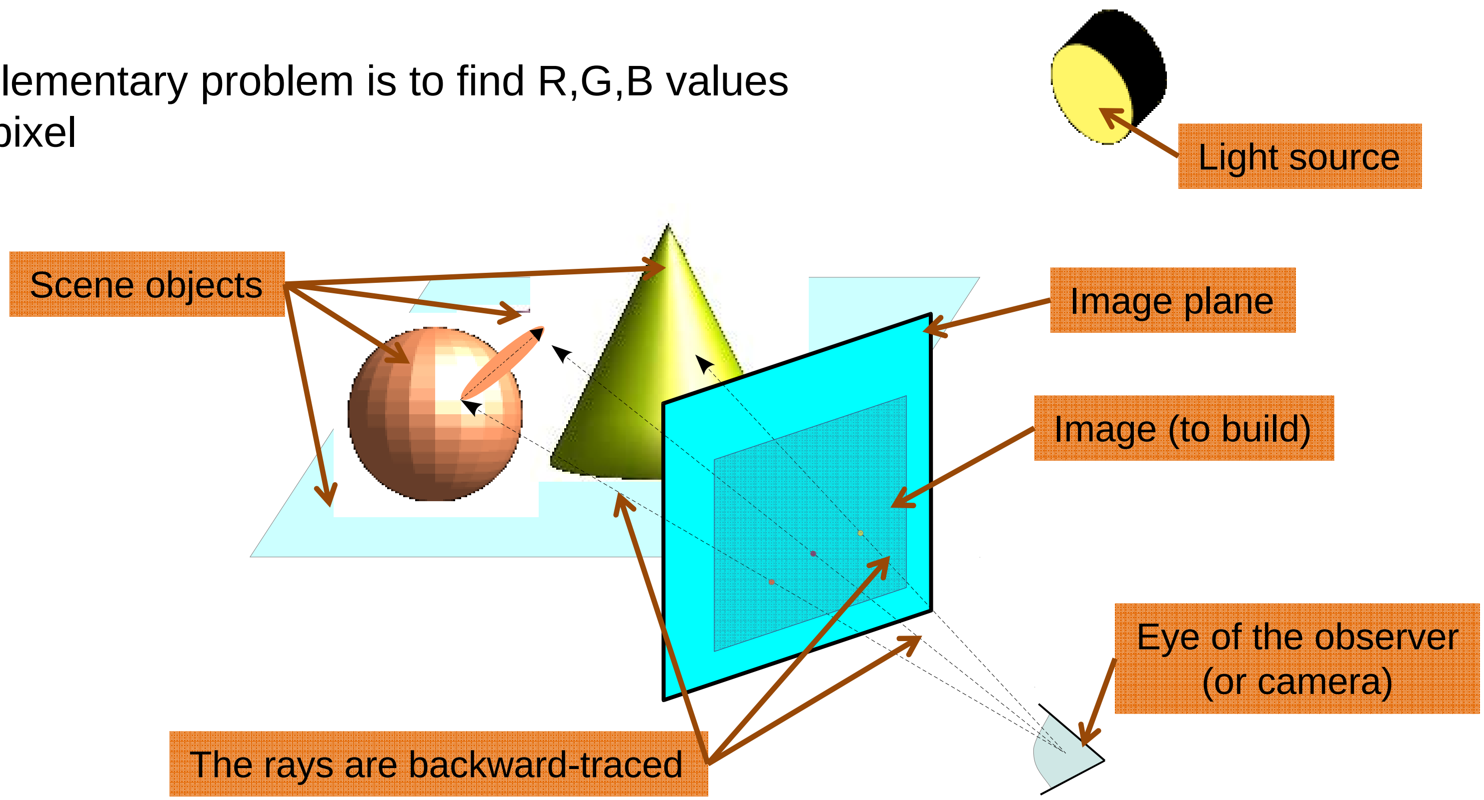
Plan of presentation

- Introduction (background)
- The toolchain for optimization of imperative functional programs
- ***Sample algorithm of program transformation***

Ray Tracing

Sample massively parallel problem

- The elementary problem is to find R,G,B values for a pixel



- The massively parallel problem is to find the R, G, B values for all the pixels of the image
- The elementary problems in the massively parallel one differs only in coordinates of the pixels on the image plane

Ray Tracing: Initial Program

The implementation may be represented as the following three functions:

•The **`render_scene`** function (***which is a C function***) is destined for filling small rectangles with the RGB intensity values for each point of the fragment contained within such a rectangle

•The **`render_scene_ut`** ***T-function*** recursively bisects the rendering area. It also calls the **`render_scene`** function – in case the size limit of the area is reached (that is the base case)

•The **`TMain`**. The launch of the T-process of the **`TMain`** function starts the execution of any application written in cT. **`TMain`** reads the scene description from the file and then launches the T process with the first call to **`render_scene_ut`**. After that, **`TMain`** solves the problem of breadth-first traversal of the binary tree built by **`render_scene_ut`** and assembles a composite image from the fragments located inside the leaves of the tree, in parallel with the computation of individual fragments performed by **`render_scene_ut/render_scene`** calls

Ray Tracing: Initial Program II

`render_scene_ut` function header & initial part

```
[void safe * sh]
render_scene_ut (double f_ulx,f_uly,f_stepx,f_stepy,
                 int nx, ny,
                 void safe * sh_scene) {
  void safe * utsh_res;

```

Recursive branch

```
  if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
    int ny1, ny2;

    ny1 = ny / 2;
    ny2 = ny - ny1;
    utsh_res = tnew (void safe * [2]);
    utsh_res [0] =
      render_scene_ut (f_ulx,f_uly,f_stepx,f_stepy,
                       nx, ny1, sh_scene);
    utsh_res [1] =
      render_scene_ut (f_ulx,f_uly + f_stepy * ny1,
                       f_stepx, f_stepy, nx, ny2,
                       sh_scene);
    sh <== utsh_res;
```

Basis of recursion

```
  } else {
    utsh_res =
      tnew (char[sizeof (frag_dsc) +
                 CHAR_PER_POINT * nx * ny] outer);
    render_scene
     (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
      ((char *) &(utsh_res->C)) + sizeof(frag_dsc));
    sh <== utsh_res;
  }
}
```

Ray Tracing: Initial Program III

`render_scene_ut` function header & initial part

List of results

Holders – special kind of pointers

Tree node holder

```
[void safe * sh]
render_scene_ut (double f_ulx,f_uly,f_stepx,f_stepy,
                        int nx, ny,
                        void safe * sh_scene) {
    void safe * utsh_res;

```

Ray Tracing: Initial Program IV

Recursive branch

Condition

Starting T-processes for subbranches

Returning the result (SEND)

Allocation of a tree node

```
if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
  int ny1, ny2;

  ny1 = ny / 2;
  ny2 = ny - ny1;
  utsh_res = tnew (void safe * [2]);
  utsh_res [0] =
    render_scene_ut (f_ulx,f_uly,f_stepx,f_stepy,
                     nx, ny1, sh_scene);
  utsh_res [1] =
    render_scene_ut (f_ulx,f_uly + f_stepy * ny1,
                     f_stepx, f_stepy, nx, ny2,
                     sh_scene);
  sh <== utsh_res;
```

Ray Tracing: initial program V
Basis of recursion

- The branch is entered in case the power of the set of jobs (i.e. size of image fragment) is reasonably small

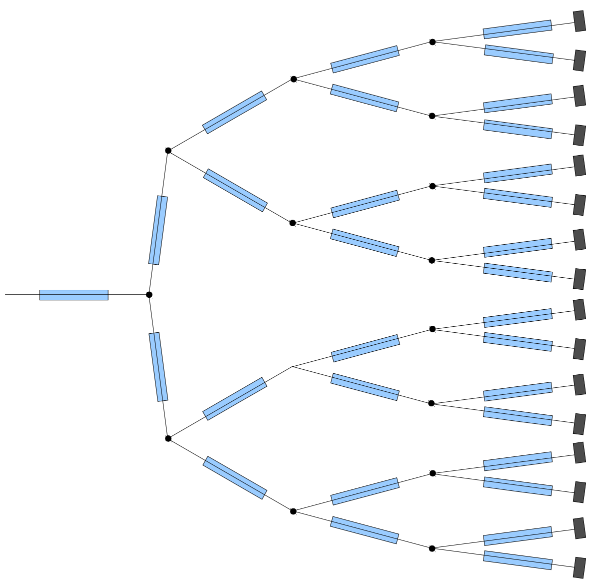
Allocation of a tree leaf

Returning the result (SEND)

Direct computation of a tree leaf

```
    } else {
      utsh_res =
        tnew (char[sizeof (frag_dsc) +
                   CHAR_PER_POINT * nx * ny] outer);
      render_scene
       (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
        ((char *) &(utsh_res->C)) + sizeof(frag_dsc));
      sh <== utsh_res;
    }
}
```

Ray Tracing: Initial Execution Pattern

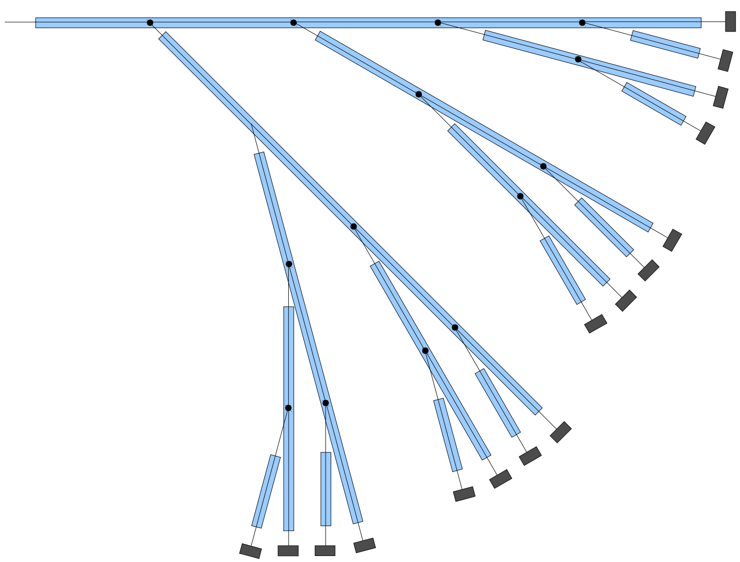


- `One T-process for each branch ==`
  `one for each node`
  `+ one for each leaf`
- ***Problem***:
  a lot of T-processes – *approximately a half* – are launched only to allocate a tree node and to start another (two) T-processes for subbranches , so they are *too lightweight*.
- ***Inefficiency*** is especially seen in case of running the program on a *distributed-memory multiprocessor (e. g. cluster)*

Legend:

•	tree node		tree branch
	tree leaf		T-process (call of T-function)

Ray Tracing: More Optimal Execution Pattern



- `One T-process for each leaf`
- The number of T-processes reduced *approximately in a half* .
- ***No inefficiency***: all processes have reasonable weight.

Legend:	•	tree node	───	tree branch
	▬	tree leaf	▭	T-process (call of T-function)

Ray Tracing: Transformation

Initial part

Building the root of subtree

```
...
if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
 int ny1, ny2;
     void safe * utsh_w;

     ny1 = ny / 2;
     ny2 = ny - ny1;
     utsh_res = tnew (void safe * [2]);
     utsh_w = utsh_res;
```

Loop through subtree nodes

Loop exit

Former recursive branch

```
     for (;;) {
       utsh_w [0] =
         render_scene_ut
          (f_ulx, f_uly, f_stepx, f_stepy,
           nx, ny1, sh_scene);
       f_uly = f_uly + f_stepy * ny1;
       if (nx * ny2 <= MIN_POINTS_PER_FRAG
           || ny2 < 2)
         break;
       ny1 = ny2 / 2;
       ny2 = ny2 - ny1;
       utsh_w [1] = tnew (void safe * [2]);
       utsh_w = utsh_w [1];
     }
```

Building the final leaf

Returning the root of subtree

```
     utsh_w [1] =
       tnew (char[sizeof (frag_dsc) +
                  CHAR_PER_POINT * nx * ny2] outer);
     render_scene
       (f_ulx, f_uly, f_stepx, f_stepy, nx, ny2,
        ((char *) (utsh_w[1].C))+sizeof(frag_dsc));
     sh <== utsh_res;
   } else {
```

Recursive Branch Rewritten : Loop Through Subtree Nodes

(see next slide for the legend)

```
    for (;;) {
      utsh_w [0] =
        render_scene_ut
          (f_ulx, f_uly, f_stepx, f_stepy,
           nx, ny1, sh_scene);
      f_uly = f_uly + f_stepy * ny1;
      if (nx * ny2 <= MIN_POINTS_PER_FRAG
          || ny2 < 2)
        break;
      ny1 = ny2 / 2;
      ny2 = ny2 - ny1;
      utsh_w [1] = tnew (void safe * [2]);
      utsh_w = utsh_w [1];
    }
```

Recursive branch rewritten : loop through subtree nodes II

Legend

(for previous slide)

- **Starting a T-process for a left branch**
- **Loop exit**
- **Reinitialization**
- ***Reinitialization: directly allocating the space for the subbranches of the right branch***

General Massively Parallel Task

Initial program wire-frame

`compute_it_ut` function header & initial part

Recursive branch

Basis of recursion

```
[void safe * sh]
compute_it_ut (ARGLIST) {
  void safe * utsh_res;
  Env A

  Prep A
  ...
  Prep Z
```

```
  if (Cond) {
    Env B

    Prep BA
    ...
    utsh_res = tnew (void safe * [2]);
    ...
    Prep BM
    utsh_res [0] = compute_it_ut (ARGS0(ARGLIST, Env A, Env
B));
    Prep BN
    ...
    Prep BZ
    utsh_res [1] = compute_it_ut (ARGS1(ARGLIST, Env A, Env
B));
```

```
  } else {
    Env C

    Prep CA
    ...
    utsh_res =
      tnew (char[sizeof (frag_dsc) + SIZE_of_SUBSET] outer);
    ...
    Prep CZ
    compute_it (ARGS(ARGLIST, Env A, Env C, & (utsh_res ->
C)));
    Postp C
    sh <== utsh_res;
  }
}
```

General Massively Parallel Task II

The “wire-frame” is a program skeleton. To write an application on it’s basis the application programmer should provide some meat:

- The C function that solves the problem for some ***volume*** of variants of initial data;
- The way to compute the size of result data (for memory allocation)
- The algorithm to compute arguments to this C function
- The way to break the ***volume*** into two equal parts
- The condition when to stop breaking and proceed to the recursion base

General Massively Parallel Task III

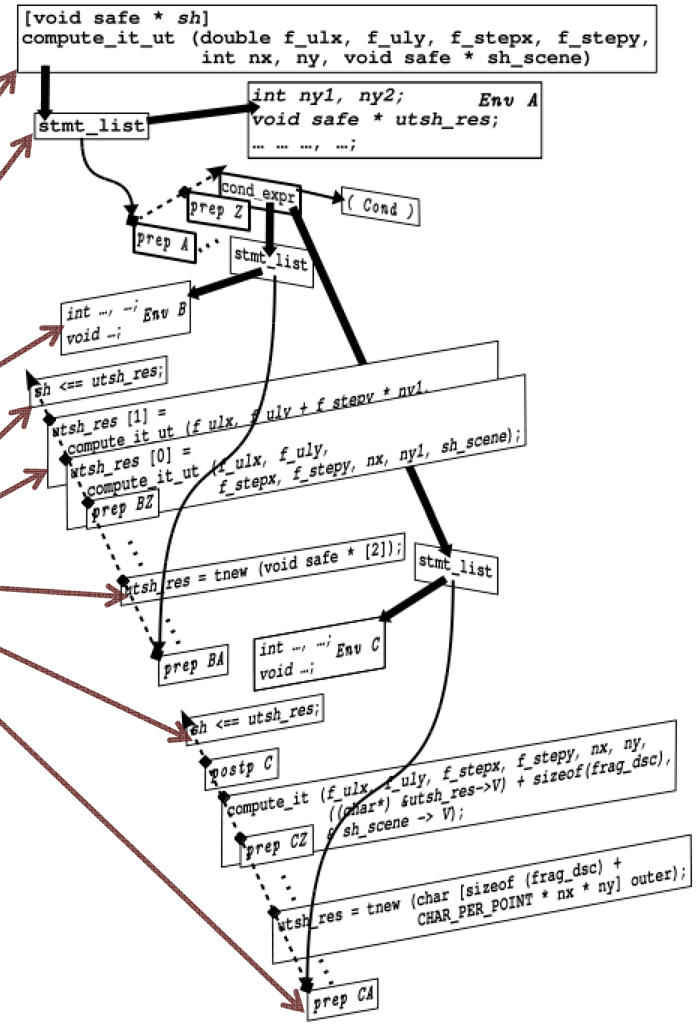
The initial wire-frame as it is (simplified)

A member of a list of top-level definitions

List of statements

Set of definitions of names (environment)

Statements (consists of elementary operation nodes)



We should remember that we are working with the internal representation

This time this one is the ***form of AST*** (Abstract Syntax Trees). ASTs are more or less equivalent to source code (up to parenthesis etc.)

General Case Transformation

The original version

- Tail recursion wasn't revealed
- The best attempt: the "***tail recursion modulo cons***" approach (by David H. D. Warren)
- A problem when trying to apply "tail recursion modulo cons" approach:
  need to use a side effect to assign the values to the externally allocated variables. This is incorrect and *explicitly prohibited* in the cT: assignments to variables that are "non-owned" by the function in the cT are possible only via value return (SEND) statements

General Case Transformation II

The idea of the approach used:

try to apply a sequence of transformations to reshape initial wire-frame of the general-case program in the same way as the ray-tracing program wire-frame was transformed

General Case Transformation III

Transformation implemented as a sequence of stages:

- Substitution
- Looping
- Final cleaning of variables and

assignments

General Case Transformation IV

- **Substitution**

The body of compute_it_ut function – realizing the recursion step – is substituted (inlined) instead of the second recursive call.

The return of the result (the SEND statements) in the inlined code is substituted with the usual assignments.

```
[void safe * sh]
compute_it_ut (ARGLIST) {
  void safe * utsh_res;
  Env A

  Prep A
  ...
  Prep Z
  if (Cond) {
    Env B

    Prep BA
    ...
    utsh_res = tnew (void safe * [2]);
    ...
    Prep BM
    utsh_res [0] = compute_it_ut (ARGS0(ARGLIST, Env A, Env
B));
    Prep BN
    ...
    Prep BZ
    utsh_res [1] = compute_it_ut (ARGS1(ARGLIST, Env A, Env
B));
  } else {
    Env C

    Prep CA
    ...
    utsh_res =
      tnew (char[sizeof (frag_dsc) + SIZE_of_SUBSET] outer);
    ...
    Prep CZ
    compute_it (ARGS(ARGLIST, Env A, Env C, & (utsh_res ->
C)));
    Postp C
    sh <== utsh_res;
  }
}
```

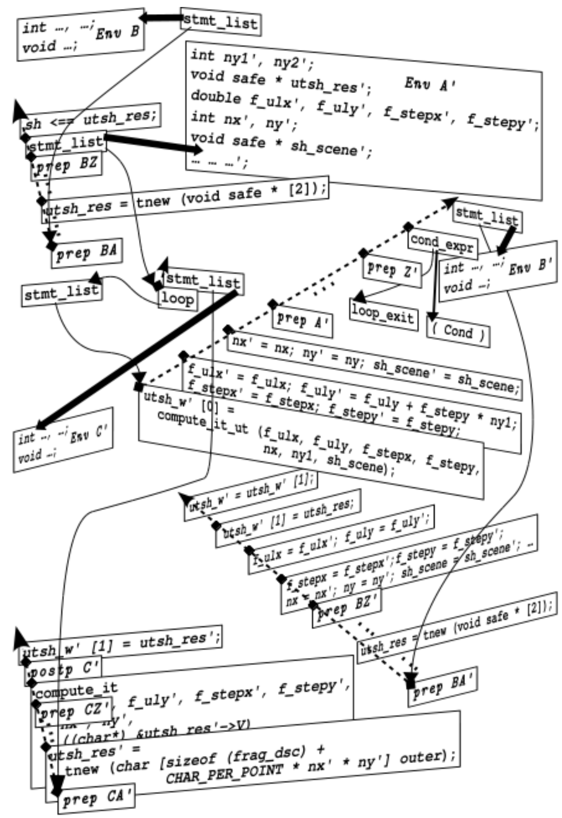
here

General Case Transformation V

- **Looping (introducing the iteration)**

The stage is executed in several steps. The execution of all the steps allows to considerably reduce the number of lightweight parallelism granules.

The two (of three) last recursive calls are completely eliminated at the looping stage. As a substitution to the eliminated recursive calls, the loop structure and the recursion base — with the call to the compute_it C function — is inserted into the recursive branch of the compute_it_ut function.



The resultant internal representation of former “recursive branch” after the looping stage

General Case Transformation VI

- **Cleaning**

After mechanically implemented transformations, the recursive branch has a number of odd assignments and T-variables. We should “optimize” them.

Example. If we apply the above transformation steps to the **`render_scene_ut`** function, the result will contain the following definition:

```
void safe * sh_scene';
```

and a pair of assignments, such as

`sh_scene' = sh_scene;` and **`sh_scene = sh_scene';`**

with no other assignments to these variables are performed (so we can substitute the **`sh_scene'`** with the **`sh_scene`** one and remove the second assignment).

General Case Transformation ???

Remarks

- ***The algorithm was based on “intuitive correctness”***
- ***The were no room prepared for the “formally verifiable correctness”***

- **Great thanks to the reviewers**

Ray Tracing: Initial Program

(revisited for reminder)

`render_scene_ut` function header & initial part

```
[void safe * sh]
render_scene_ut (double f_ulx,f_uly,f_stepx,f_stepy,
                 int nx, ny,
                 void safe * sh_scene) {
   void safe * utsh_res;

```

Recursive branch

```
   if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
     int ny1, ny2;

     ny1 = ny / 2;
     ny2 = ny - ny1;
     utsh_res = tnew (void safe * [2]);
     utsh_res [0] =
       render_scene_ut (f_ulx,f_uly,f_stepx,f_stepy,
                        nx, ny1, sh_scene);
     utsh_res [1] =
       render_scene_ut (f_ulx,f_uly + f_stepy * ny1,
                        f_stepx, f_stepy, nx, ny2,
                        sh_scene);
     sh <== utsh_res;
```

Basis of recursion

```
   } else {
     utsh_res =
       tnew (char[sizeof (frag_dsc) +
                  CHAR_PER_POINT * nx * ny] outer);
     render_scene
      (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
       ((char *) &(utsh_res->C)) + sizeof(frag_dsc));
     sh <== utsh_res;
   }
}
```

Ray Tracing: transformation'

Stage 1: compile the initial cT program to the C one

- The meaning of the program is preserved:

$$(f_{comp}([\![RT]\!]^{cT}\ (cT\ data))) = [\![f_{comp}(RT)]\!]^{C}(f_{comp}(cT\ data))$$

- Mapping to the C programming language with standard sequential operational semantics of cT as a functional language is implemented by f_{comp}.

Ray Tracing: transformation'

render_scene_ut_int
C function header
& initial part

```
void
render_scene_ut_int (double f_ulx, double f_uly,
                     double f_stepx, double f_stepy,
                     int nx, int ny,
                     tholder sh_scene,
                     tholder * sh_int_ret) {
   tholder utsh_res_int;

```

Recursive branch

```
   if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
     int ny1, ny2;

     ny1 = ny / 2;
     ny2 = ny - ny1;
     utsh_res_int -> ptr = calloc (tholder [2]);
     utsh_res_int -> cnt = 2;
     render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
                          nx, ny1, sh_scene,
                          utsh_res_int -> ptr);
     render_scene_ut_int (f_ulx,f_uly + f_stepy * ny1,
                          f_stepx, f_stepy, nx, ny2,
                          sh_scene,
                          (tholder *) utsh_res_int -> ptr + 1);
     * sh_int_ret =  utsh_res_int;
     return;
```

Basis of recursion

```
  } else {
    utsh_res_int -> ptr =
    calloc (TSIZE (sizeof (frag_dsc)
                  + CHAR_PER_POINT*nx*ny)), 1);
    utsh_res_int -> cnt = 1;
    ((tholder *) utsh_res_int -> ptr) -> tag = TTAG_LCELL;
    ((tholder *) utsh_res_int -> ptr) -> len =
      TSIZE (sizeof (frag_dsc) + CHAR_PER_POINT*nx*ny);
    render_scene
     (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
      ((char *) utsh_res_int -> ptr) + SZ_THEAD
      + sizeof(frag_dsc));
    * sh_int_ret = utsh_res_int;
    return;
}
}
```

Ray Tracing: transformation'

`render_scene_ut_int`

C function

header & initial part

void

result

tholder: a pair (ptr, cnt);

```
void
render_scene_ut_int (double f_ulx, double f_uly,
                     double f_stepx,double f_stepy,
                     int nx, int ny,
                     tholder sh_scene_int,
                     tholder * sh_int_ret) {
  tholder utsh_res_int;

```

Ray Tracing: transformation'
Recursive branch

Allocation of the node

Return of the result

Recursive calls

```
07   if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
08     int ny1, ny2;
09
10     ny1 = ny / 2;
11     ny2 = ny - ny1;
12     utsh_res_int -> ptr = calloc (tholder [2]);
12b    utsh_res_int -> cnt = 2;
14     render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                          nx, ny1, sh_scene,
13                          utsh_res_int -> ptr);
17     render_scene_ut_int (f_ulx,f_uly + f_stepy * ny1,
18                          f_stepx, f_stepy, nx, ny2,
19                          sh_scene,
16                          (tholder *) utsh_res_int -> ptr + 1);
20     * sh_int_ret =  utsh_res_int;
20b    return;
```

Ray Tracing: transformation'

Recursion basis branch

Allocation of the leaf

Calling the rendering function

Return of the result

```
} else {
  utsh_res_int -> ptr =
    calloc (TSIZE (sizeof (frag_dsc)
                         + CHAR_PER_POINT*nx*ny)), 1);
  utsh_res_int -> cnt = 1;
  ((tholder *) utsh_res_int -> ptr) -> tag = TTAG_LCELL;
  ((tholder *) utsh_res_int -> ptr) -> len =
    TSIZE (sizeof (frag_dsc) + CHAR_PER_POINT*nx*ny);
  render_scene
    (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
     ((char *) utsh_res_int -> ptr) + SZ_THEAD
     + sizeof(frag_dsc));
  * sh_int_ret = utsh_res_int;
  return;
}
```

Ray Tracing: transformation' II

Stage 2: reveal the tail recursion

- Take the recursive branch out of conditional statement to the final part of the function
- Make the recursive call be placed at the very end of the function

Ray Tracing: transformation' II

1. Take the recursive branch out of the conditional statement

render_scene_ut_int
C function header
& initial part

```
01 void
02 render_scene_ut_int (double f_ulx, double f_uly,
02b                      double f_stepx, double f_stepy,
03                       int nx, int ny,
04                       tholder sh_scene,
04b                      tholder * sh_int_ret) {
05   tholder utsh_res_int;
06
```

Basis of recursion

```
07  if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
22    utsh_res_int -> ptr =
23    calloc (TSIZE (sizeof (frag_dsc)
24               + CHAR_PER_POINT*nx*ny)), 1);
24b   utsh_res_int -> cnt = 1;
24c   ((tholder *) utsh_res_int -> ptr) -> tag = TTAG_LCELL;
24d   ((tholder *) utsh_res_int -> ptr) -> len =
24d     TSIZE (sizeof (frag_dsc) + CHAR_PER_POINT*nx*ny);
25    render_scene
26      (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27       ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b      + sizeof(frag_dsc));
28    * sh_int_ret = utsh_res_int;
28    return;
29  }
```

Former recursive branch

```
08  int ny1, ny2;
09
10  ny1 = ny / 2;
11  ny2 = ny - ny1;
12  utsh_res_int -> ptr = calloc (tholder [2]);
12b utsh_res_int -> cnt = 2;
14  render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                       nx, ny1, sh_scene,
13                       utsh_res_int -> ptr);
17  render_scene_ut_int (f_ulx,f_uly + f_stepy * ny1,
18                       f_stepx, f_stepy, nx, ny2,
19                       sh_scene,
16                       (tholder *) utsh_res_int -> ptr + 1);
20  * sh_int_ret =  utsh_res_int;
20b return;
30 }
```

Ray Tracing: transformation' II

1. Take the recursive branch out of the conditional statement …

Basis of recursion

Inverted condition changes the order of branches

```
if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
  utsh_res_int -> ptr =
...
  * sh_int_ret = utsh_res_int;
  return;
}
```

The branch ends with the return statement, so configuration of the control is preserved

Ray Tracing: transformation' II

1. Take the recursive branch out of the conditional statement to the final part of the function

```
int ny1, ny2;

...
render_scene_ut_int (f_ulx,f_uly + f_stepy * ny1,
                     f_stepx, f_stepy, nx, ny2,
                     sh_scene,
                     (tholder *) utsh_res_int -> ptr + 1);
* sh_int_ret =  utsh_res_int;
return;
}
```

Still recursive call isn't the last statement of the function

Ray Tracing: transformation' II

2. Make the recursive call be placed at the very end of the function

This statement moving is clear since the side effect (the assignment to the external memory) will became visible only after returning out of the function

```
08  int ny1, ny2;
09
...
20  * sh_int_ret = utsh_res_int;
17  render_scene_ut_int (f_ulx,f_uly + f_stepy * ny1,
18                       f_stepx, f_stepy, nx, ny2,
19                       sh_scene,
16                       (tholder *) utsh_res_int -> ptr + 1);
20b return;
30 }
```

Return statement not needed at the very end of function that returns void

Ray Tracing: transformation' III

Stage 3: convert the tail recursion into iteration

- All the function body statements go into the body of "***`for(;;)`***" loop
- The last recursive call is substituted with recalculation of the values stored in function arguments – according to the argument list of the call statement

Ray Tracing: transformation' III

Conversion of the tail recursion into iteration

Loop statement

Recalculation of the arguments

```
01 void
02 render_scene_ut_int (double f_ulx, double f_uly,
02b                      double f_stepx, double f_stepy,
03                       int nx, int ny,
04                       tholder sh_scene,
04b                      tholder * sh_int_ret) {
0x1    for (;;) {
05       tholder utsh_res_int;
06
07       if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25         render_scene
26           (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27            ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b           + sizeof(frag_dsc));
28         * sh_int_ret = utsh_res_int;
28         return;
29       }
08       int ny1, ny2;
09
10       ny1 = ny / 2;
11       ny2 = ny - ny1;
12       utsh_res_int -> ptr = calloc (tholder [2]);
12b      utsh_res_int -> cnt = 2;
14       render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                            nx, ny1, sh_scene,
13                            utsh_res_int -> ptr);
20       * sh_int_ret =  utsh_res_int;
0y1      ny = ny2;
0y2      sh_int_ret = (tholder *) utsh_res_int -> ptr + 1;
0x2    }
30 }
```

Ray Tracing: transformation' III

Conversion of the tail recursion into iteration

Recalculation of the (different) arguments

```
0y1        ny = ny2;
0y2        sh_int_ret = (tholder *) utsh_res_int -> ptr + 1;
```

- The place where the sample program can differ from the general case
- Only the arguments should be recalculated that differs from upper-level call
- Recalculation order is regulated with the rules of calculation of the arguments
- In the general case we can create a temporary variable for each value to be recalculated.

Ray Tracing: transformation' +

The problem:

- ***<u>We have</u>*** **a pure sequential function written in C with some side effects**

- ***<u>We need</u>*** **a parallel-style function written in cT without any side effect**

Ray Tracing: transformation' +

The route:

- **Remove the side effects out of the loop**
- **Integrate the side effects to the structures of that style into which the cT- function value return was mapped**
- **Convert the C function back into the cT one (“parallelize” it)**

Ray Tracing: transformation' IV

Stage 4: remove the side effects out of the loop

- All the side effects in the function body looks like that:

```
* sh_int_ret = utsh_res_int;
```

- After recalculation **`sh_int_ret`** points to the newly allocated “own” data:

```
sh_int_ret = (tholder *) utsh_res_int -> ptr + 1;
```

- That means that “side effects” exists only on the first iteration, when **`sh_int_ret`** points to the external data

Ray Tracing: transformation' IV

The code motions that ***evidently*** doesn't change the semantics

The rest of former "recursive branch"

```
08        int ny1, ny2;
09
10        ny1 = ny / 2;
11        ny2 = ny - ny1;
12        utsh_res_int -> ptr = calloc (tholder [2]);
12b       utsh_res_int -> cnt = 2;
14        render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                             nx, ny1, sh_scene,
13                             utsh_res_int -> ptr);
20        * sh_int_ret =  utsh_res_int;
```

Grouping the "wireframe" sentences together

The result of the conversion of the rest of former "recursive branch"

```
12        utsh_res_int -> ptr = calloc (tholder [2]);
12b       utsh_res_int -> cnt = 2;
20        * sh_int_ret =  utsh_res_int;
```

```
08        int ny1, ny2;
09
10        ny1 = ny / 2;
11        ny2 = ny - ny1;
14        render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                             nx, ny1, sh_scene,
13                             utsh_res_int -> ptr);
```

Ray Tracing: transformation' IV

Partial unwinding of the loop

A

B

The loop body after the code motions

C

D

E

```
0x1    for (;;) {
05         tholder utsh_res_int;
06
07         if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25           render_scene
26             (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27              ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b             + sizeof(frag_dsc));
28           * sh_int_ret = utsh_res_int;
28           return;
29         }
12         utsh_res_int -> ptr = calloc (tholder [2]);
12b        utsh_res_int -> cnt = 2;
20         * sh_int_ret =  utsh_res_int;
08         int ny1, ny2;
09
10         ny1 = ny / 2;
11         ny2 = ny - ny1;
14         render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                              nx, ny1, sh_scene,
13                              utsh_res_int -> ptr);
0y1        ny = ny2;
0y2        sh_int_ret = (tholder *) utsh_res_int -> ptr + 1;
0x2    }
```

In the absence of the **`break`** statements and **`goto`**s

```
for (;;) {A; B; C; D; E;} == A; B; C; for (;;) {D; E; B; C;}
```

Ray Tracing: transformation' IV

Partial unwinding of the loop

The loop preamble after the partial unwinding

```
05        tholder utsh_res_int;
06

07        if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25          render_scene
26            (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27             ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b            + sizeof(frag_dsc));
28          * sh_int_ret = utsh_res_int;
28          return;
29        }

12        utsh_res_int -> ptr = calloc (tholder [2]);
12b       utsh_res_int -> cnt = 2;
20        * sh_int_ret =  utsh_res_int;
```

The loop after the partial unwinding

```
0x1     for (;;) {

08        int ny1, ny2;
09
10        ny1 = ny / 2;
11        ny2 = ny - ny1;
14        render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                             nx, ny1, sh_scene,
13                             utsh_res_int -> ptr);

0y1       ny = ny2;
0y2       sh_int_ret = (tholder *) utsh_res_int -> ptr + 1;

07        if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25          render_scene
26            (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27             ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b            + sizeof(frag_dsc));
28          * sh_int_ret = utsh_res_int;
28b          return;
29        }

12        utsh_res_int -> ptr = calloc (tholder [2]);
12b       utsh_res_int -> cnt = 2;
20        * sh_int_ret =  utsh_res_int;

0x2     }
```

Ray Tracing: transformation' IV

Renaming of the "wireframe" pointers in the *loop body*

The loop preamble after the renaming

```
05     tholder utsh_res_int;
0z1    tholder sh_u_int, * sh_w_intp;
06
```

```
07     if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25       render_scene
26         (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27          ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b         + sizeof(frag_dsc));
28       * sh_int_ret = utsh_res_int;
28b      return;
29     }
```

```
12     utsh_res_int -> ptr = calloc (tholder [2]);
12b    utsh_res_int -> cnt = 2;
20    * sh_int_ret =  utsh_res_int;
0zz    sh_u_int = utsh_res_int;
```

utsh_res_int → sh_u_int

sh_int_ret → sh_w_int

The loop after the renaming

```
0x1    for (;;) {
```

```
08         int ny1, ny2;
09
10         ny1 = ny / 2;
11         ny2 = ny - ny1;
14         render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                              nx, ny1, sh_scene,
13                              sh_u_int -> ptr);
```

```
0y1        ny = ny2;
0y2        sh_w_intp = (tholder *) sh_u_int -> ptr + 1;
```

```
07         if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25           render_scene
26             (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27              ((char *) sh_u_int -> ptr) + SZ_THEAD
27b             + sizeof(frag_dsc));
28           * sh_w_intp = sh_u_int;
28b          return;
29         }
```

```
12         sh_u_int -> ptr = calloc (tholder [2]);
12b        sh_u_int -> cnt = 2;
20         * sh_w_intp =  sh_u_int;
```

```
0x2    }
```

Ray Tracing: transformation' IV

Stage 4: remove the side effects out of the loop

- **Here it is:**
  there are no more references to the externally allocated memory in the loop body

Ray Tracing: transformation' V

Stage 5: Integrate the side effects to the structures of that style into which the cT-function value return was mapped

Preparing to the “back conversion” to the cT

in the cT

```
sh <== utsh_res
```

```
* sh_int_ret = utsh_res_int;
return;
```

in the C

Ray Tracing: transformation' V

Preparing to the “back conversion” to the cT

```
01 void
02 render_scene_ut_int (double f_ulx, double f_uly,
02b                     double f_stepx, double f_stepy,
03                      int nx, int ny,
04                      tholder sh_scene,
04b                     tholder * sh_int_ret) {

05     tholder utsh_res_int;
0z1    tholder sh_u_int, * sh_w_intp;
06

07     if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25       render_scene
26         (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27          ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b         + sizeof(frag_dsc));
28       * sh_int_ret = utsh_res_int;
28b      return;
29     }

12     utsh_res_int -> ptr = calloc (tholder [2]);
12b    utsh_res_int -> cnt = 2;
20     * sh_int_ret = utsh_res_int;
0zz    sh_u_int = utsh_res_int;
```

```
0x1    for (;;) {

08       int ny1, ny2;
09
10       ny1 = ny / 2;
11       ny2 = ny - ny1;
14       render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                            nx, ny1, sh_scene,
13                            sh_u_int -> ptr);

0y1      ny = ny2;
0y2      sh_w_intp = (tholder *) sh_u_int -> ptr + 1;

07       if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25         render_scene
26           (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27            ((char *) sh_u_int -> ptr) + SZ_THEAD
27b           + sizeof(frag_dsc));
28         * sh_w_intp = sh_u_int;
20         * sh_int_ret =  utsh_res_int;
28b        return;
29       }

12       sh_u_int -> ptr = calloc (tholder [2]);
12b      sh_u_int -> cnt = 2;
20       * sh_w_intp =  sh_u_int;

0x2    }

30 }
```

Again, the code motion is legal, since:

- The sentence is executed only once
- The values were not accessed between the source and destination code positions
- The side effect will in both cases will be seen only after the return out of the function

Ray Tracing: transformation' V

Ready to the “back conversion” to the cT

```
01 void
02 render_scene_ut_int (double f_ulx, double f_uly,
02b                      double f_stepx, double f_stepy,
03                       int nx, int ny,
04                       tholder sh_scene,
04b                      tholder * sh_int_ret) {
```

```
05      tholder utsh_res_int;
0z1     tholder sh_u_int, * sh_w_intp;
06
```

```
07      if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25        render_scene
26          (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27           ((char *) utsh_res_int -> ptr) + SZ_THEAD
27b          + sizeof(frag_dsc));
28        * sh_int_ret = utsh_res_int;
28b       return;
29      }
```

```
12      utsh_res_int -> ptr = calloc (tholder [2]);
12b     utsh_res_int -> cnt = 2;
0zz     sh_u_int = utsh_res_int;
```

“Surprisingly”, all the side effects are now in the correct environments

```
0x1   for (;;) {
```

```
08      int ny1, ny2;
09
10      ny1 = ny / 2;
11      ny2 = ny - ny1;
14      render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                           nx, ny1, sh_scene,
13                           sh_u_int -> ptr);
```

```
0y1     ny = ny2;
0y2     sh_w_intp = (tholder *) sh_u_int -> ptr + 1;
```

```
07      if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
...
25        render_scene
26          (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27           ((char *) sh_u_int -> ptr) + SZ_THEAD
27b          + sizeof(frag_dsc));
28        * sh_w_intp = sh_u_int;
20        * sh_int_ret =  utsh_res_int;
28b       return;
29      }
```

```
12      sh_u_int -> ptr = calloc (tholder [2]);
12b     sh_u_int -> cnt = 2;
20      * sh_w_intp =  sh_u_int;
```

```
0x2   }
```

```
30 }
```

Ray Tracing: transformation' VI

Stage 6: converting the C function back into the cT one ("parallelization")

- The ***`tholder`*** C type to the "***`void safe *`***" cT one
- The header of the C function returning the "***`void`***" to the cT-style header
- The "correctly-shaped" side effects to the cT "SEND" sentences
- The recursive calls to the cT –style ones
- The C-style memory allocations to the cT-style ones

Ray Tracing: transformation' VI

Back to the cT (parallelization)

Conversion of the function header

in the C

```
void
render_scene_ut_int (double f_ulx, double f_uly,
                     double f_stepx, double f_stepy,
                     int nx, int ny,
                     tholder sh_scene,
                     tholder * sh_int_ret) {
```

```
[void safe * sh]
render_scene_ut (double f_ulx, double f_uly,
                 double f_stepx, double f_stepy,
                 int nx, int ny,
                 void safe * sh_scene) {
```

in the cT

Ray Tracing: transformation' VI

Back to the cT (parallelization)

Conversion of the "`tholder`" type and the side effects

in the C

```
tholder
```

```
safe void *
```

```
* sh_int_ret = utsh_res_int;
return;
```

```
sh <== utsh_res
```

in the cT

Ray Tracing: transformation' VI

Back to the cT (parallelization)

Conversion of the recursive function calls

in the C

```
14  render_scene_ut_int (f_ulx,f_uly,f_stepx,f_stepy,
15                       nx, ny1, sh_scene,
13                       sh_u_int -> ptr);
```

```
13   sh =
14     render_scene_ut (f_ulx,f_uly,f_stepx,f_stepy,
15                      nx, ny1, sh_scene);
```

in the cT

Ray Tracing: transformation' VI

Back to the cT (parallelization)

Conversion of the memory allocations

in the C

```
utsh_res_int =
  calloc (TSIZE (sizeof (frag_dsc)
                 + CHAR_PER_POINT * nx * ny)), 1);
 utsh_res_int -> cnt = 1;
 ((tholder *) utsh_res_int -> ptr) -> tag = TTAG_LCELL;
 ((tholder *) utsh_res_int -> ptr) -> len =
   TSIZE (sizeof (frag_dsc) + CHAR_PER_POINT*nx*ny);
```

```
utsh_res =
  tnew (char[sizeof (frag_dsc) +
                   CHAR_PER_POINT * nx * ny] outer);
```

in the cT

Ray Tracing: transformation' VI

Back to the cT (parallelization): ENJOY

```
01 [ void safe * sh]
02 render_scene_ut_int (double f_ulx, double f_uly,
02b                      double f_stepx, double f_stepy,
03                       int nx, int ny,
04                       void safe * sh_scene) {

05      void safe * utsh_res_int;
0z1     void safe * sh_u, sh_w;
06

07      if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
22        utsh_res =
23          tnew (char[sizeof (frag_dsc) +
24                CHAR_PER_POINT * nx * ny] outer);
25        render_scene
26          (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27           ((char *) &(utsh_res->C))+sizeof(frag_dsc));
28        sh <==  utsh_res;
29      }

12      utsh_res = tnew (void safe * [2]);
0zz     sh_u_int = utsh_res_int;
```

```
0x1     for (;;) {

08        int ny1, ny2;
09
10        ny1 = ny / 2;
11        ny2 = ny - ny1;
13        sh =
14         render_scene_ut (f_ulx,f_uly, f_stepx,f _stepy,
15                          nx, ny1, sh_scene);

0y1       ny = ny2;
0y2       sh_w = sh_u + 1;

07        if (!(nx * ny > MIN_POINTS_PER_FRAG && ny >= 2)) {
22          sh_u =
23            tnew (char[sizeof (frag_dsc) +
24                  CHAR_PER_POINT * nx * ny] outer);
25          render_scene
26            (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27             ((char *) &(utsh_res->C))+sizeof(frag_dsc));
28          * sh_w = sh_u;
20          sh <==  utsh_res;
29        }

12        sh_u = tnew (void safe * [2]);
20        * sh_w =  sh_u;

0x2     }

30 }
```

Ray Tracing: transformation' Lessons

- The transformation is formally correct at each step, so the result program is formally equivalent to the initial one
- The transformation is not nearly the same as initially proposed
- Result program is the same as hand-crafted (modulo simple insignificant changes)
- Mostly wire-frame control and data structures are concerned. When application-specific structures are touched (as the recalculation of the arguments when converting recursive call to the iteration), the correctness is provided for the general case. So transformation can be applied for the solving of any massively-parallel problem
- “Tail recursion modulo cons” rule has worked again, but now in a less usual way

Toolchain: the other possibilities

(i.e. in addition to equalizing the granulation of the parallelism)

- Various local optimizations oriented on the T-System specific properties (e.g. code motion to the region before the possible T-process suspending due to non-readiness of a T-variable)
- Code instrumentation for rapid in-line checking of the incoming messages (to improve reactivity of run-time support on distributed-memory multiprocessors)
- ***The opportunity to use the specialization techniques (including some forms of program supercompilation or distillation)***

The last entry was (and still is) the most inspiring in the course of the development.

Summary and Conclusiones

- Background (the T-system)
- Review of the ACCT compiling and transformation toolchain
- Algorithm for optimization of general massively-parallel tasks (after all, obviously correct)

- Only a first effort in the direction of cT programs optimization
- Still a lot of things to do to complete the first version of the system
- Wary hope the toolset will be useful in the context of metacomputation technologies applications

Thanks

Questions

? Internal representation suitable for meta-computations

? “Parallel computing with help of metacomputations” or “metacomputations vs. parallel computing”

E-mail: “Alexei Adamovich <lexa@adam.botik.ru>”

? Your questions