Tempo Documentation - Interacting with a C Program Specializer

Renaud Marlet

Tempo Documentation – Interacting with a C Program Specializer

Renaud Marlet

N° 0390

September 7th, 2010

Distributed Systems and Services

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--0390--FR+ENG

Tempo Documentation - Interacting with a C Program Specializer

Renaud Marlet[*]

Theme: Distributed Systems and Services
Networks, Systems and Services, Distributed Computing

Project-Team Phoenix

Technical Report n° 0390 — September 7th, 2010 — 149 pages

Abstract: *Tempo* is a program specializer for C programs. It has been developed at IRISA / INRIA - University of Rennes 1 (1994-2000), and then at LaBRI / INRIA - University of Bordeaux 1 (since 2000).

This technical report puts together a cleaned-up and reformatted version of the various on-line manuals and other useful documents that have been written on Tempo for its distribution, but that used to exist only as separate and sometimes mobile HTML pages. Grouping them and giving them a technical report number make it easy to reference them in a publication.

Although it is not developed and maintained anymore, Tempo is still distributed. It can be downloaded from the Phoenix project-team web pages (http://phoenix.inria.fr/). Publications concerning Tempo as well as tutorial slides are also available on this web site.

Technical information in this report is (theoretically) up to date with respect to the last official release of Tempo, dated February 11th, 2003.

Keywords: program specialization, partial evaluation, compile-time specialization, runtime specialization, user's guide, reference manual, C language

[*] Currently at École des Ponts ParisTech – renaud.marlet@enpc.fr

Documentation de Tempo –
Interagir avec un spécialiseur de programmes C

Résumé : *Tempo* est un spécialiseur de programmes C. Il a été développé à l'IRISA / INRIA - University of Rennes 1 (1994-2000), puis au LaBRI / INRIA - University of Bordeaux 1 (à partir de 2000).

Ce rapport technique rassemble des versions « nettoyées » et remise en forme des divers manuels et autres documents pratiques qui ont été écrits pour la distribution de Tempo, mais qui n'existaient que sous forme de pages HTML séparées et parfois mobiles. Les regrouper et leur donner un numéro de rapport technique permet d'y faire proprement référence dans des publications.

Bien qu'il ne soit désormais plus développé et maintenu, Tempo est toujours distribué. Il peut être téléchargé sur le site web de l'équipe-projet (http://phoenix.inria.fr/). Les publications de l'équipe sur Tempo ainsi que les transparents d'un tutoriel sont également disponibles sur ce site.

Les informations techniques dans ce rapport sont (en théorie) à jour par rapport à la dernière version officielle de Tempo, qui date du 11 février 2003.

Mots-clés : spécialisation de programmes, évaluation partielle, spécialisation à la compilation, spécialisation à l'exécution, guide de l'utilisateur, manuel de référence, langage C

Tempo Specializer Documentation

.....

1 > Tempo — Tutorial

Partial evaluation is program transformation that automates a *specialization* process. *Tempo Specializer* (or *Tempo* for short) is a partial evaluator for C programs. It has been applied in various domains such as operating systems and networking, computer graphics, scientific computation, software engineering and domain specific languages.

1.1 Program Specialization

Let us consider a program P, taking two arguments S and D, and producing a result R:

P(S,D) = R

If S is *known* before the execution of the program (S is said to be *static* whereas D is *dynamic*), we may form a new program <P,S> that waits until D is available and then calls the original P program on (S,D) to produce the same result R:

<P,S>(D) = P(S,D) = R

However, now that S is "known to P", computations relying on S can be performed before D is actually available. Therefore, we can form a new program P_S equivalent to <P,S> where computations depending on S have already been exploited. We thus have:

$$P_S(D) = \langle P,S\rangle(D) = P(S,D) = R$$

The program P_S is called a *specialization* of P with respect to the *invariant* S.

More generally, specialization exploits any invariant present in the code, not only input values. The idea is to factor out computations from the specialized program. A specialized program generally runs faster and in some cases may also be smaller that the original program.

1.2 Partial Evaluation

Partial evaluation is the process that automates program specialization. A *partial evaluator* (or *specializer*) is a program M that takes two arguments, a program P and a known (static) subset of the input S, and produces the result P_S:

$$M(P,S) = P_S$$

(Very) roughly speaking, common partial evaluation can be thought of as a combination of aggressive constant folding, inlining, loop unrolling and *inter-procedural* constant propagation applied to *all* data types (including pointers, structures and arrays) instead of just scalars.

Partial evaluation has been applied in various domains such as operating systems, computer graphics, numerical computation, circuit simulation, program understanding, compiling and compiler generation.

1.3 What is Tempo ?

Tempo Specializer is a partial evaluator for C programs. It has first been developed in the Compose project-team at IRISA / INRIA - University of Rennes 1 (1994-2000), in France. The group then moved to Bordeaux to form the Phoenix project-team at LaBRI / INRIA - University of Bordeaux 1 (since 2000).

Features

Tempo is an *off-line* specializer: the specialization process is divided into two steps. First, a program analysis propagates information about known and unknown values throughout the code. The output of this process may be visualized in order to assess the degree of specialization of the program. Then, the user may provide actual specialization values (*i.e.*, values of invariants) and the source code of a specialized program is automatically produced; this is called compile time specialization. Tempo can specialize programs at run time as well, for cases when the invariants are known only after execution starts.

Distribution

Although it is not developed and maintained anymore, Tempo is still distributed. It can be downloaded from the Phoenix project-team web pages (http://phoenix.inria.fr/). Publications concerning Tempo as well as tutorial slides are also available on this web site. See also the Installation Manual.

A Prototype, Not A Product

Note that Tempo is a *prototype*, not a product. It comes with *no warranty*.

1.4 Guided Tour of Tempo

Let us consider a C file `/home/jake/spec/power.c` with the following content.

```
int pow(int base, int expon)
{
  int accum = 1;

  while (expon > 0) {
    accum *= base;
    expon--;
  }
  return accum;
}
```

We want to specialize the `pow()` function with respect to a known exponent. We want to specialize it at compile time as well as at run time. The following sections list the basic interaction with Tempo for achieving that.

Further reading:

- Modular Specialization (User's Manual)

1.5 Running Tempo

The user interacts with the Tempo Specializer system through an interactive top level that encapsulates all functionalities. It is actually a Standard ML top level (Tempo is mainly built on top of SML/NJ) where the Tempo functionalities have been pre-loaded.

To run the Tempo system, invoke the shell command `tempo`. You should see something that looks like this (the output of the system is in bold face, roman is for what you type):

```
mingus% tempo
TEMPO Version 1.191, 03/24/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes
val it = () : unit
-
```

The "`-`" sign is the prompt character of this top level.

Further reading:

- The SML Top Level (User's Manual)

1.6 Configuring The Analysis

Before specializing a program, Tempo must first analyze it. This not only prepares the specialization process (it makes it more efficient) but also lets you visualize the possible impact of specialization before actually providing specialization value.

Further reading:

- The Analyses and Their Precision (User's Manual)

1.6.1 Working Directory

The first thing that you have to do is to tell Tempo where the program that you want to specialize is or, more precisely, where the *working directory is*. Type the following command in the Tempo top level.

```
- cd "/home/jake/spec";
val it = "/home/jake/spec" : string
```

You could also have run `tempo` directly from the `/home/jake/spec` directory, as the current directory is considered the working directory by default.

1.6.2 Function To Specialize and Specialization Context

Then, you must specify what is the function in `power.c` that you are interested in (there could be several of them, calling each others) and with respect to what known information you want to specialize it. This function is called the *entry point*. In our case, the entry point is `pow()`.

To specify the entry point, you have to create a configuration file next to your `power.c` file. It has the same prefix but extension `config.sml`, *i.e.* `power.config.sml`. In this file, write the following

```
entry_point := "pow(D,S)";
```

Setting the `entry_point` variable tells Tempo two things: (i) that the function to specialize is `pow()`, and (ii) that its first argument is to be considered unknown ("`D`" is for dynamic) while the second can be assumed known ("`S`" is for static).

Note that, in the general case, if there are several functions that call each other in the C file, specialization is inter-procedural: each function is analyzed and specialized in turn starting from the specified entry point, with an automatic propagation of known quantities and specialization values.

Further reading:

- Configuring the Analyses (User's Manual)

1.6.3 Visualizer

You are now almost ready to run the analysis phase of Tempo. This phase prepares the actual specialization process and lets you assess the degree of specialization of your function. This assessment is provided through generated files that represent the results of the analysis. Most of those files are colored files. There exists two formats, that let you use different visualizers.

MIME text/enriched format.
Those file are suffixed by "`.color`". You can view them using the emacs (or xemacs) editor.

HTML format.
Those file are suffixed by "`.html`". You can view them using any HTML viewer.

By default, Tempo generates MIME text/enriched color files. If you prefer to use HTML files, do the following in the top level:

```
- viewer := html;
val it = () : unit
```

Further reading:

- Visualization of Colored files (User's Manual)

1.7 Running The Analysis

You now can run the analysis phase of Tempo. As said above, this phase prepares the actual specialization process and lets you assess the degree of specialization of your program. The top-level command to do so is named `an`. It takes as an argument the root name of the program file, *i.e.* `"power"`.

```
- an "power";
Starting from /home/jake/spec/power.c
Generating Suif file
Generating Suif tree
Generating abstract syntax tree
Eliminating gotos
Analyzing aliases
iterating analysis
Eliminating function pointers
Analyzing side effects
Generating MONO binding time information
Generating MONO evaluation time information (phase 1)
Generating MONO evaluation time information (phase 2)
Generating flattened program (phase 1 - Return sensitivity)
Generating flattened program (phase 2 - S&D)
Generating action tree
Produced file : power.at
val it = true : bool
```

Tempo lists all the sub-phases of the analysis. In the working directory, a set of files have been generated, among which are:

- `power.bta.color`: Color-annotated result of the binding-time analysis
- `power.eta2.color`: Color-annotated result of the evaluation-time analysis
- `power.at.color`: Color-annotated result of the action analysis
- `power.at`: Final result the analysis phases

If you did `viewer := html` then you see "`.html`" files rather than "`.color`" files.

1.7.1 Binding-Time Analysis

In the working directory, a set of files have been generated. Try to visualize the file named `power.bta.color` in an emacs editor (or file `power.bta.html` if you decided to switch to HTML files generation). You should see something like this.

```
/* TEMPO Version 1.193, 04/18/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes */

/* LEGEND:  STATIC  DYNAMIC  SD_FUNC  STRUCTURE  BOTTOM
 */

extern int power(int, int);

extern int power/*0*/(int base, int expon)  {
    int accum;

    accum = 1;
    if (0 < expon)
      {
        do
          {
            accum = accum * base;
            expon = expon - 1;
          }
        while (0 < expon);
      }
    return accum;
  }
```

As you can see, your original program as been transformed a little. Some of those transformations are performed inside the SUIF system, which is used as a front end for parsing C files. Others are done by Tempo during abstract syntax generation, to work on a smaller C subset.

- `int accum = 1` has been split into a declaration and an explicit assignment.
- The `while` loop has been turned into a `do` ... `while` loop, duplicating the condition.
- `accum *= base` and `expon--` have been made explicit.

Moreover, colors express binding times, as computed by the binding-time analysis (BTA). For example, assuming `expon` is known (shown with blue "`STATIC`" color, as indicated in the legend), the expression `0 < expon` is also known, *i.e.* static. Similarly, the assignment `expon = expon - 1` inside the `do` ... `while` loop is static because it only involves static expressions and it is performed a static number of times. (This reasoning requires induction, which translates into a fix-point iteration in the analysis.)

Although assigned to `1` in the first statement (which makes it static), the `accum` variable becomes dynamic in the `do` ... `while` loop: it becomes dynamic as soon as the first iteration as it is assigned an expression that depends on the unknown (shown with red "`DYNAMIC`" color, as indicated in the legend) `base` variable: `accum * base`. The second iteration reaches the fix-point.

Further reading:

- Colors For Binding-Time Information (User's Manual)

1.7.2 Evaluation-Time Analysis

Note that, in the above BTA file, the left-hand side of assignments appears as static because it expresses the binding time of the address, not of the memory content. This is not true in the `power.eta2.color`, which is the next interesting file to visualize.

```
/* TEMPO Version 1.193, 04/18/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes */

/* LEGEND:  STATIC  DYNAMIC  STAT&DYN  STRUCTURE  BOTTOM
 */

extern int power(int, int);

extern int power/*0*/(int base, int expon)  {
    int accum;

    accum = 1;
    if (0 < expon)
      {
        do
          {
            accum = accum * base;
            expon = expon - 1;
          }
        while (0 < expon);
      }
    return accum;
  }
```

It shows the result of running the evaluation-time analysis (ETA2). At this point, the color of an assignment to a variable expresses the evaluation time of the assignment, and thus the binding time of the variable after the assignment. This is why the left-hand side of the `accum = accum * base` assignment has become totally dynamic.

The evaluation-time analysis actually propagates backwards uses of dynamic and static variables. Even though computed "static" by the BTA, the `accum = 1` assignment must have a dynamic facet as there is a dynamic use of the variable `accum` (inside the loop): we need the preceding definition(s) of `accum` to be residualized, *i.e.* to appear in the specialized program. The assignment should thus be both static and dynamic, *i.e.* `accum = 1`. However, there is actually not any use of a static `accum` after this assignment, hence the static facet of the assignment is useless: the static computed value is never exploited. Consequently, the static facet of the assignment can be removed, yielding `accum = 1`.

Further reading:

- Colors For Evaluation-Time Information (User's Manual)

1.7.3 Action Analysis

The final output of the analysis phase is the `power.at.color` file, which reflects the action analysis.

```
/* TEMPO Version 1.193, 04/18/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes */

/* LEGEND:  EVAL  REDUCE  REBUILD  IDENTITY  STRUCTURE  EV&RES
 */

extern int power_1/*0*/(int base, int expon)  {
    int accum;

    accum = 1;
    if (0 < expon)
      {
        do
          {
            accum = accum * base;
            expon = expon - 1;
          }
          while (0 < expon);
      }

    return accum;
  }
```

Here colors express specialization actions, *i.e.* transformations to perform at specialization time. Blue expressions are totally evaluated at specialization time. Green expressions contain a totally evaluated condition; they are reduced: an `if` is rewritten into one of its branches, a loop is unrolled. Orange expressions (here, blocks) contains subcomponents that can be further specialized but that, themselves, cannot be reduced: during specialization, they are rewritten as they are, with specialized sub-terms. Finally, red expressions are just copied verbatim to form the specialized program.

Knowing this, we can see that the `if` and the `while` will be simplified (*i.e.* removed) by the specialization process. All computation depending on the `expon` variable disappears.

We can even see the form of any specialized program: it will always start with the `accum = 1` assignment, it will be followed by several `accum = accum * base` (actually, the line will be repeated `expon` times), and it will be always ended by `return accum`.

As long as your original program does not change, you do not have to re-run the analysis; the generated `power.at` file will be used as the starting point for all subsequent phases.

Further reading:

- Colors For Actions (User's Manual)

1.8 Building a Compile-Time Specializer

Once the analysis has been performed successfully, *i.e.* if you are satisfied with the computed binding times, you can move on to building a compile-time specializer. The top-level command to do so is named `cs`. It takes as an argument the root name of the program file, *i.e.* `"power"`, just like the `an` command.

```
- cs "power";
Starting from /home/jake/spec/power.at
Generating specializer file
Produced file : power.ctspec.C
val it = true : bool
```

If you list the files in your `/home/jake/spec` directory, you will see that three new files have been created (you do not have to understand at all their content):

- `power.sctx.h`: header file for specifying actual specialization values.
- `power.ctspec.C`: the specializer for `pow()`
- `power.ev.c`: the evaluator of static expressions in `pow()`

As long as your original program does not changes, you do not have to re-run the construction of the compile-time specializer; the above files can be used as the starting point of the following actual specialization phase.

1.9 Running a Compile-Time Specializer

Now we need to provide Tempo with actual specialization values. Because the initialization of those values can be arbitrarily complex (including library calls), the values are given using a C file, that is linked to the compile-specializer for performing actual specialization. This C file is named `power.sctx.c`. It should

- include the file `power.sctx.h`,
- define a function named `set_specialization_context()` whose arguments are pointers to the actual arguments of your entry point, so that you can set them by an indirect assignment.

In our case, we create the following `power.sctx.c` file.

```
#include "power.sctx.h"

void set_specialization_context(int *expon)
{
  *expon = 4;
}
```

Note that the `expon` variable here is an `*int`, not an `int` as in the `pow()` entry point: the value is written using a pointer dereferencing. (Writing `expon` as opposed to `*expon` just sets the local variable and leaves the corresponding argument unaffected, *i.e.* uninitialized.)

Note also that you must provide a value for the static arguments only. The arguments of `set_specialization_context()` should be in the same order as those of the entry point. You can use the name that you want for the arguments as it is a normal C function.

When that is done, run the specializer by invoking the `sp` top-level command:

```
- sp "power";
Starting from /home/jake/spec/power.ctspec.C
Generating specializer
gcc   -c -I/usr/local/lib/tempo/bin/../ctcg
  /home/jake/spec/power.ev.c
  -o /home/jake/spec/SunOS-5/power.ev.o
gcc   -c -I/usr/local/lib/tempo/bin/../ctcg
```

```
    /home/jake/spec/power.ctspec.C
    -o /home/jake/spec/SunOS-5/power.ctspec.o
  gcc   -c /home/jake/spec/power.sctx.c
    -o /home/jake/spec/SunOS-5/power.sctx.o
  gcc    -L/usr/local/lib/tempo/bin/../ctcg/SunOS-5
    /home/jake/spec/SunOS-5/power.ev.o
    /home/jake/spec/SunOS-5/power.ctspec.o
    /home/jake/spec/SunOS-5/power.sctx.o
    -lctcg -lbsdmalloc
    -o /home/jake/spec/SunOS-5/power.ctspec
  Specializing
  TEMPO specializer: version 1.18, 98/03/03, Copyright Irisa
  Postprocessing
  Translating abstract syntax into C text
  Produced file : power.cts.c
  val it = true : bool
```

The resulting `power.cts.c` file is:

```
/* TEMPO Version 1.193, 04/18/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes */

/*
call signature of entry point pow:
binding times of formals: base D
 */
extern int _Gpow_1_0_0(int);

extern int _Gpow_1_0_0/*0*/(int base)
/*
binding times of formals: base D
evaluation time of body: D
evaluation time of return: D
evaluation times of pairings: D
 */
  {
    int accum;

    accum = base * base * base * base;
    return accum;
  }
```

Though correct and nice, this result may seem a bit strange as we said in the Action Analysis section that any specialized program would always start with the `accum = 1` assignment, followed by `expon` times copies of `accum = accum * base` , and ended by `return accum`, *i.e.*

```
extern int _Gpow_1_0_0/*0*/(int base)
  {
    int accum;

    accum = 1;
    accum = accum * base;
    accum = accum * base;
    accum = accum * base;
    accum = accum * base;
    return accum;
  }
```

The reason is that specialization is followed by a post-processing phase which performs some algebraic transformations. It is not crucial that Tempo performs those transformations as they are done by most compilers nowadays. But the post-processing also performs some cleaning up of the specialized code which makes it more pleasant to read.

Each time that you want to specialize your program with respect to new specialization values, you do just have to edit the `file.sctx.c` file and re-run the `sp` top-level command.

Further reading:

- Invocation of a Compile-Time Specializer (User's Manual)
- Post-Processing (Reference Manual)

1.10 Building a Run-Time Specializer

Starting from the "`.at`" action file after the analysis has been performed, you may also generate a run-time specializer. There is nothing like an `.sctx.c` file to provide at this stage since the static parameters will be given at run time, when calling the specializer. In order to build the run-time specializer, run the `rs` command.

```
- rs "power";
Starting from /home/jake/spec/power.at
Generating binary run-time specializer
gcc -O0   -c /home/jake/spec/power.temp.c
  -o /home/jake/spec/SunOS-5/power.temp.o
/usr/local/lib/tempo/bin/../rtcg/tcc/SunOS-5/tcc
  /home/jake/spec/power.temp
  /home/jake/spec/SunOS-5/power.temp
tcc: ret in template temp_5 offset 0x0010
gcc -O0   -c  -DSUNOS5 /home/jake/spec/power.rtspec.c
  -o /home/jake/spec/SunOS-5/power.rtspec.o
ld -r -o /home/jake/spec/SunOS-5/power.rts.o
  /home/jake/spec/SunOS-5/power.rtspec.o
  /home/jake/spec/SunOS-5/power.temp.o
  /usr/local/lib/tempo/bin/../rtcg/rts/SunOS-5/functions.o
  /usr/local/lib/tempo/bin/../rtcg/rts/SunOS-5/flushcopy.o
Produced file : power.rts.o
val it = true : bool
```

The result of the specialization may be found in `SunOS-5/power.rts.o`. (A directory named after the architecture and system of the current machine is generated.) A file named `power.rts.h` is also generated in the working directory; it contains the signature of the run-time specialization function.

Further reading:

- Construction of a Run-Time Specializer (User's Manual)

1.11 Using a Run-Time Specializer

The `power.rts.h` file contains the (partial) prototype of the run-time specializer:

```
/* TEMPO Version 1.193, 04/18/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes */

extern void *rts_pow_1(int);
```

The run-time specializer `rts_pow_1` is a function that takes an integer argument (the static value of `expon`) and returns a function pointer. Indirect call of this function pointer with an integer argument (some value for `base`) returns the same result as the original `int pow(int,int)` function.

Here is a example of use. Write the following `test.c` file.

```
#include <stdio.h>
#include <stdlib.h>

#include "power.rts.h"

main(int argc, char **argv)
{
  int i, base, expon;
  int (*spec_pow)(int);  // Important to give proper type

  base = atoi(argv[1]);  // First argument on the command line
  expon = atoi(argv[2]); // Second argument on the command line

  spec_pow = rts_pow_1(expon);       // Invoke run-time specialization

  printf("pow(%d,%d) = %d\n",
    base, expon, (*spec_pow)(base)); // Use specialized pow()
}
```

Now compile it and link it with the run-time specializer, *i.e.* `SunOS-5/power.rts.o`.

```
mingus% gcc test.c SunOS-5/power.rts.o -o testpower
mingus% ./testpower 2 3
pow(2,3) = 8
```

Congratulations!, you have achieved your first run-time code specialization.

Further reading:

- Invocation of a Run-Time Specializer (User's Manual)

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

2 > Tempo — User's Manual

2.1 Running Tempo

The user interacts with the Tempo Specializer system through an interactive top level that encapsulates all functionalities. It is actually a Standard ML top level (Tempo is mainly built on top of SML/NJ) where the Tempo functionalities have been pre-loaded.

To run the Tempo system, invoke the shell command `tempo`. You should see something that looks like this (the output of the system is in bold face, roman is for what you type):

```
mingus% tempo
TEMPO Version 1.191, 03/24/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes
val it = () : unit
-
```

The "`-`" sign is the prompt character of this SML top level.

To exit the Tempo system, just type `^D`, *i.e.* control-D, *i.e.* end-of-file character.

2.1.1 Running the Analysis

Before specializing a program, Tempo must first analyze it. This not only prepares the specialization process (it makes it more efficient) but also lets you assess the degree of specialization before actually providing specialization value.

Starting from compulsory files *`file.c`* and *`file.config.sml`* (the *`file.actx.c`* file is optional), the analyses produce as the final result a *`file.at`* file. By default, it also produces color files that show the intermediate results of the analyses (see Visualization.)

The SML command that runs the whole analysis phase is `an`. Alternatively, you may use the `tempo` SML command (not the `tempo` shell-level command) to run sub-phases individually or up to a certain phase. *E.g.*,

```
- an "power";
Starting from /home/jake/spec/power.c
Generating Suif file
Generating Suif tree
Generating abstract syntax tree
Eliminating gotos
Analyzing aliases
iterating analysis
Eliminating function pointers
Analyzing side effects
Generating MONO binding time information
Generating MONO evaluation time information (phase 1)
Generating MONO evaluation time information (phase 2)
Generating flattened program (phase 1 - Return sensitivity)
Generating flattened program (phase 2 - S&D)
Generating action tree
Produced file : power.at
val it = true : bool
```

Note that if the program to specialize does not lie in the directory where the `tempo` shell-level command was run, you have to specify the working directory explicitly (see Specifying the Program to Specialize below).

See also:

- The Analyses and Their Precision
- Configuring the Analyses
- Visualization

2.1.2 Building a Compile-Time Specializer

When the "`.at`" action file has been generated and assessed, you can move on to building a compile-time specializer, using the top-level `cs` command. The result is split into two files: `.ctspec.C` and `.ev.c`. *E.g.*,

```
- cs "power";
Starting from /home/jake/spec/power.at
Generating specializer file
Produced file : power.ctspec.C
val it = true : bool
```

An additional `.sctx.h` is also produced, to be used in the next phase, *i.e.* actual specialization.

See also:

- Compile-Time Specializer Construction

2.1.3 Running a Compile-Time Specializer

The last step is the generation of the source of a specialized program, given actual values for the static parameters of the entry point. For this, you must provide a `.sctx.c` file, that expresses parameters' initialization as C code, and run the `sp` command.

```
- sp "power";
Starting from /home/jake/spec/power.ctspec.C
Generating specializer
gcc   -c -I/usr/local/lib/tempo/bin/../ctcg
  /home/jake/spec/power.ev.c
  -o /home/jake/spec/SunOS-5/power.ev.o
gcc   -c -I/usr/local/lib/tempo/bin/../ctcg
  /home/jake/spec/power.ctspec.C
  -o /home/jake/spec/SunOS-5/power.ctspec.o
gcc   -c /home/jake/spec/power.sctx.c
  -o /home/jake/spec/SunOS-5/power.sctx.o
gcc    -L/usr/local/lib/tempo/bin/../ctcg/SunOS-5
  /home/jake/spec/SunOS-5/power.ev.o
  /home/jake/spec/SunOS-5/power.ctspec.o
  /home/jake/spec/SunOS-5/power.sctx.o
  -lctcg -lbsdmalloc
  -o /home/jake/spec/SunOS-5/power.ctspec
Specializing
TEMPO specializer: version 1.18, 98/03/03, Copyright Irisa
Postprocessing
Translating abstract syntax into C text
Produced file : power.cts.c
val it = true : bool
```

The result of the specialization may be found in the generated `.cts.c` file. It is up to you to install it in your original application. Make sure you call the specialized entry point when appropriate, *i.e.* when the specialization hypothesis (actual values of static parameters) are valid.

See also:

- Compile-Time Specialization

2.1.4 Building a Run-Time Specializer

Starting from the "`.at`" action file after the analysis has been performed, you may also generate a run-time specializer. There is nothing like an `.sctx.c` file to provide at this stage as the static parameters will be given at run time, when calling the specializer.

In other to build the run-time specializer, run the `rs` command.

```
- rs "power";
Starting from /home/jake/spec/power.at
Generating binary run-time specializer
gcc -O0   -c /home/jake/spec/power.temp.c
  -o /home/jake/spec/SunOS-5/power.temp.o
/usr/local/lib/tempo/bin/../rtcg/tcc/SunOS-5/tcc
  /home/jake/spec/power.temp
  /home/jake/spec/SunOS-5/power.temp
tcc: ret in template temp_5 offset 0x0010
```

```
gcc -O0   -c  -DSUNOS5 /home/jake/spec/power.rtspec.c
  -o /home/jake/spec/SunOS-5/power.rtspec.o
ld -r -o /home/jake/spec/SunOS-5/power.rts.o
  /home/jake/spec/SunOS-5/power.rtspec.o
  /home/jake/spec/SunOS-5/power.temp.o
  /usr/local/lib/tempo/bin/../rtcg/rts/SunOS-5/functions.o
  /usr/local/lib/tempo/bin/../rtcg/rts/SunOS-5/flushcopy.o
Produced file : power.rts.o
val it = true : bool
```

The result of the specialization may be found in the generated `.rts.o` file, that is generated in a directory named after the architecture and system of the current machine (see variable `arch_dep_dir`). A `.rts.h` file is also generated in the working directory; it contains the signature of the run-time specialization function.

It is up to you to install the run-time specializer in your original application, including the management of run-time specialized functions: when to specialize, caches for already specialized functions, etc. Make sure you call the run-time specialized entry points when appropriate, *i.e.* when the specialization hypothesis (actual values of static parameters) are valid.

See also:

- Run-Time Specialization
- Using a Run-Time Specializer (Tutorial)

2.1.5 SML Top Level

The Tempo Specializer top level provides a standard SML top-level interaction. You may type any SML code, *i.e.* definitions as well as expressions to evaluate, separated by semicolons.

```
- 1+2;
val it = 3 : int
- "Te" ^ substring("compose",2,3);
val it = "Tempo" : string
- val foo = ["bar","gee"];
val foo = ["bar","gee"] : string list
- (hd foo, tl foo, fact 6);
val it = ("bar",["gee"]) : string * string list
- hd [];
uncaught exception Hd
- fun fact n = if n=0 then 1 else n*fact(n-1);
val fact = fn : int -> int
- fact 6;
val it = 720 : int
```

Using SML Variables

In particular, from this top level the user may consult or set Tempo configuration variables. Most of those variables are reference variables, *i.e.* pointers to a value. This can be seen from the type: it ends with "`ref`". *E.g.*

```
- explicit_cts_bufsize;
val it = ref NONE : int option ref
```

```
- max_decls_size;
val it = ref 100 : int ref
- max_decls_size := 3 * (!max_decls_size);
val it = () : unit
- max_decls_size;
val it = ref 300 : int ref
```

Note the use of:

`!variable`
: to return the value pointed by a existing reference variable, *i.e.* to dereference it.

`variable := value`
: to assign a value to an existing reference variable, *i.e.* to overwrite its previous value.

`val variable = value`
: to define a top-level variable. (This is not needed unless having very complex interactions with Tempo.)

Using SML Commands

The user may also run Tempo commands, *e.g.*

```
- cd "/home/jake/spec";
val it = "/home/jake/spec" : string
- cs "power";
Starting from /home/jake/spec/power.at
Generating specializer file
Produced file : power.ctspec.C
val it = true : bool
```

Note that all of the SML commands provided by Tempo are curried *i.e.* arguments are provided separated by spaces. For example, the SML-level `tempo` command (not the `tempo` shell-level command) is used as follows:

```
- tempo "power" "alias" "bta";
Starting from /home/jake/spec/power.alias.as
Eliminating function pointers
Analyzing side effects
Generating MONO binding time information
Produced file : power.bta.as
val it = true : bool
```

If you forget an argument to this command, you will not get a type error; you will be returned a partially instantiated closure without any action being performed, *e.g.*

```
- tempo "alias" "bta";
val it = fn : string -> bool
```

However, if you provide the arguments as a tuple (separated by commas and surrounded by parentheses), you do get a type error:

```
- tempo("power","alias","bta");
```

```
std_in:26.1-26.29 Error: operator and operand don't agree
(tycon mismatch)
  operator domain: string
  operand:         string * string * string
  in expression:
    tempo ("power","alias","bta")
```

Using SML Files

Besides the user's interaction at the SML top level, there are two SML files that are used in Tempo: the `file.config.sml` per-program configuration file and the `.tempo.sml` per-session configuration file. Those are standard SML files. In particular, you may write (nestable) comments in them as follows.

```
(* Specialization of pow(base,expon)
   Assume expon is known
 *)
entry_point := "pow(D,S)";

post_inlining := true;
output_mode := COLOR;
external_functions := RESIDUALIZE ["printf","scanf"];
```

Note that each variable setting must be ended by a semicolon. If you wish to load explicitly an SML file, type:

```
use "some_file.sml";
```

2.1.6 Running Tempo In Batch Mode

If you wish to run Tempo in batch mode, just pipe SML commands into the shell command `tempo`, *e.g.*

```
mingus% echo 'wd := "/home/jake/spec"; cs "power"' | tempo
Running Tempo on SunOS-5
TEMPO Version 1.191, 03/24/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes
val it = () : unit
- val it = () : unit
Starting from /home/jake/spec/power.at
Generating specializer file
Produced file : power.ctspec.C
val it = true : bool
- mingus%
```

The end of the input marks the end of the job.

2.1.7 Running Tempo Under Emacs

Here is a suggestion for running Tempo under emacs (courtesy of Olivier Danvy). Put the following lines in your `.emacs` file and run `run-tempo` to get a convenient shell buffer with a running Tempo.

```
(setq tempo-path "/usr/local/lib/tempo/bin/tempo")
```

```
(defun run-tempo ()
  "Run an inferior Tempo process,
     input and output via buffer *Tempo*."
  (interactive)
  (require 'comint)
  (pop-to-buffer
    (make-comint "Tempo"
                 "/bin/sh"
                 nil
                 "-c"
                 tempo-path))
  ;(inferior-sml-mode)
  (make-local-variable 'comint-prompt-regexp)
  (setq comint-prompt-regexp "- ")
  (setq mode-name "Inferior Tempo"))
```

2.2 Modular Specialization

It is not usually practical or even desirable to apply specialization to a complete application, *i.e.* from the `main()` function down to all leaf functions. Instead, specialization is usually applied to part of an application (without altering the rest of the application) or to library functions.

2.2.1 Concepts

Modular specialization supports for specializing a fragment of an application. It is explained through the following definitions.

Application.
An *application* is some code that makes up a full executable, *i.e.* that contains a `main()` function. Part of it may be available only in binary form; you only need to have the source code of the parts that you wish to specialize (the module), and to know how those parts are called (the execution context).

Entry Point.
The *entry point* is (the name of) the function of your application that you wish to specialize. In practice, it generally is not the `main()` function.

Formals, Actuals and Parameters.
We use the terms *formals* to refer to the formal arguments of a function, *actuals* to refer to their actual value and *parameters* to designate the formals as well as the global variables that are visible from this function.

Execution Context.
An *execution context before* a function is called (or *execution pre-context*) is some knowledge about the current memory store as well as the actual arguments of the function. An *execution context after* a function is called (or *execution post-context*) is some knowledge about the uses of the results of the function, not only the returned value but all uses of the memory affected by the function call. (The term "knowledge" is intentionally vague as in practice we actually refer to analysis and specialization contexts; see below.)

Analysis Context.

The *analysis context* is a set of properties that are true at some point of the execution flow. The analysis context *before* a function is called (or *analysis pre-context*, or *initial analysis context*) is the execution pre-context that consists of alias relations and binding time properties concerning its parameters. The analysis context *after* calling a function (or *analysis post-context*), consists of the live uses of memory locations occurring after the function has returned.

Specialization Context.

The *specialization context* of the entry point (and more generally of any function) consists of actual values for the parameters (including content values for pointers).

Program (a.k.a. Module).

The *program* (or *module*) is the set of functions of your application that you wish to specialize. It at least includes the entry-point function. It may also include other functions called by the entry point, however it does not have to include all such functions. The program is provided to Tempo as a single C file.

External Function.

External functions are functions that are external to the program, *i.e.* functions of your application that are called by the program but that you do not want to specialize. The definitions of external functions should not be provided in the program file. Such functions may include library functions such as `printf()` or `strcpy()`.
Making a function external can be a choice: it may not interesting to specialize, not actually specializable (always called with only static or only dynamic arguments), too big, etc. When a function is part of a library whose source code is inaccessible, the function of course must be external.

Abstract Function.

An *abstract function* is a function written to model the behavior of another function (usually an external function) with respect to the analyses. In other words, it may not respect the execution semantics of the function (*i.e.* return the same results) but it should respect the analysis semantics (*i.e.* return the same analysis results). Most often, an abstract function is useful in order to give an abstract definition to an existing function that is uninteresting to specialize or that is available only as binary code. See Behavior of External Functions below for more details.

Context Abstract Function.

Tempo needs to know the execution context in which the entry point is called from the application. This analysis context can be specified using a *context abstract function*. It may also be necessary to specify the behavior of the application *after* the entry-point function is called, especially uses of locations that are manipulated by the entry point (because otherwise they may be specialized away). This may also be specified using a context abstract function.

Abstract Function File.

Abstract functions (for external functions as well as context abstract functions) are provided to Tempo in a separate file suffixed with `actx.c` (*e.g.* `fft.actx.c`). This *abstract function file* is optional.

More practical details are given in the sections Behavior of External Functions and Specifying Complex Analysis Contexts below.

2.2.2 Installation of a Specialized Function

It is up to the user to integrate in the application a specialized function or, more generally, various instances of specialized functions. (Whether the specialized functions are produced at compile time or generated at run time is a separate issue.) In doing so, you must make sure that you call the specialized functions when appropriate, *i.e.* when the specialization hypotheses (actual values of static parameters) are valid. There are several strategies for that. In the general case, you must implement some switching mechanism between the original function and specialized functions.

Specialized Application

In some cases, it is possible to restrict the domain of your application so that your are sure that the specialized functionality will always be called in the appropriate context, unconditionally. In those cases, you can just replace the original entry point with the specialized one. This is often the case when you specialize interpreters. Another, less common case is when you specialize your whole application.

Dynamic Testing

If switching functions is needed, one approach is to test the execution context. If it matches a context for which you have a specialized function (either constructed at compile time or generated at run time), then call it. Otherwise call the original, generic, entry point function.

The cost is in performing the tests for each invocation of the function. This strategy is interesting when the test is negligible with respect to the execution time of the specialized function.

Guards

Another strategy to switching functions is to invoke the function indirectly via a pointer variable and to put guards at each program point where the execution context may change (with respect to specialization hypotheses). Each time the context is changed in the course of the execution, the corresponding guard checks if the new execution context matches a context for which you have a specialized function (either constructed at compile time or generated at run time). If so, the function pointer variable is set to the specialized function. If not, the function pointer variable is set to the original, generic, entry point function. Invocations should be indirect, through the function pointer variable.

The cost here is in performing the tests each time the context may change. This second strategy is interesting when there are many function invocations between each context change. Otherwise, you might end up performing tests, installing new specialized functions, but never actually calling the function.

Specialized Functions Cache

Another issue concerns specialized functions cache when doing run-time specialization: When is it interesting to make a new specialized function as opposed to use the original function? When is it interesting to keep in a cache an installed specialized function when the context changes, in the hope that it will eventually switch back to the same context? The answers to those questions are highly dependent on your application...

2.3 The Analyses and Their Precision

Tempo is an off-line partial evaluator: specialization is preceded by an analysis phase that prepares the specialization process (and makes it more efficient). Moreover, it lets you assess the degree of specialization of your program. The main analyses compute:

- Alias analysis: the set of possible targets for pointer locations and expressions.
- Side-effect analysis: the set of non-locals variables that are read or written in functions.
- Binding-time analysis: binding-time properties for each statement and expression.
- Evaluation-time analysis: evaluation-time properties for each statement and expression.

A large part of the power of an off-line partial evaluator lies in the precision of its analyses. All Tempo analyses are *inter-procedural*, *i.e.* properties at the various call sites of a function are exploited when analyzing the function. In the following, we list other features, that are specific to each analysis. But before that, we need to present the memory model used by Tempo.

2.3.1 Memory Model

Before explaining the different ways to parameterize and to read the results of Tempo's analyses, the concept of "location" must first be understood: all initial and deduced properties are specified or computed in terms of locations. Here are some definitions that will be used afterwards.

Scalar and Structured Types.

A *scalar type* is an integer, floating point or pointer type (including array). A *structured type* is a structure or union.

Named Memory Cells.

All C objects are stored in *memory cells*. Each memory cell has an address. *Scalar memory cells* store integers, floating point numbers or pointers (including pointers to arrays). *Composite memory cells* store structures, unions and arrays (i.e. the sequence of cells, not the pointer). A *named memory cell* is a memory cell that can be given a name, i.e.

- a variable (whether local or global),
- a function (for indirect function calls),
- an array (i.e. the name of the constant pointer to the first array cell),
- a structured data type (structure or union),
- a field of a structured data type.

This is in contrast with for example `malloc()`, that does not return a named memory cell. In other words, named memory cells include static and stack allocated cell, but exclude heap allocated cells.

Locations.

A *location* is a named scalar memory cell. The fact that a location is scalar and that Tempo's analyses work on locations means that there is no property directly attached to a structure or a union; properties only apply to (scalar) fields of those structured types. Consequently, when providing information about a structured data type, each of its (scalar) field must be referred to individually. Similarly, visualization of computed properties attaches no explicit information to structured data type as a whole; however, each occurrence of individual fields does reflect the properties.

Composite Locations.

A *composite location* is a location that corresponds to possibly many actual memory cells. Composite locations are

- arrays: a single name for all cells of the array, *i.e.* there is not an explicit location for each cell (see Array Monovariance below),
- structure and union fields: a single name for all instances of the same structure or union type (see Structure Mono/polyvariance below).

Since properties are attached to locations, this means in particular that properties of arrays are properties that are true for all cells of the array (i.e. index does not matter).

Globals, Locals and Non Locals.

Globals are location that are static (in the C sense, *i.e.* not on the stack nor on the heap) and visible from everywhere in the program file. *Locals* of a function are stack allocated locations, *i.e.* local variables. *Non locals* of a function are all locations but locals that can be manipulated from the function, *i.e.* globals and, possibly, locals of other functions up in the call graph that are passed as pointer arguments or via globals.

Structure Mono/Polyvariance.

In this Tempo version, our alias and binding-time analyses are by default *monovariant* with respect to structured types (structure and union). This means that for the purposes of the analyses, all instances of a given field are represented by a *single* location. Thus, properties of fields are properties that are true for all instances of the structured type. In Tempo, we use the name of the type to refer to structured type locations, as opposed to using names of instances.

However, there is an experimental feature that allows structure *polyvariance*. Please see variables `poly_structs` and `struct_version` .

Array Monovariance.

Similarly, our alias and binding-time analyses are *monovariant* with respect to array cells. This means that for the purposes of the analyses, all cells of an array are represented by a *single* location. Thus, properties of arrays are properties that are true for all cells of the array. As a matter of fact, in Tempo we use the name of the array to refer to any (pointed) array cell.

Composite Locations and Binding Times.

Because of monovariance, once a field has become dynamic at some point in the control flow, it cannot be made static again afterwards, neither by an assignment nor by any other mechanism. This is because making the field of one instance static does not guarantee that the corresponding field of all other instances may be considered static, whereas it is always safe to leave it dynamic. The same applies to array cells: putting a dynamic value in an array cell makes the whole array dynamic for the rest of the control flow. E.g.,

```
struct str {int a; ...} s1, s2;
int u[5], x;
// Assume everything is static initially

x = dynexp;      // x becomes dynamic
u[0] = dynexp;   // u[i] becomes dynamic for all i
s1.a = dynexp;   // Both s1.a and s2.a become dynamic

x = 42;          // x is now static
u[0] = 42;       // u[i] stays dynamic for all i
s1.a = 42;       // Both s1.a and s2.a stay dynamic
```

Note that this happens even if there is *only one* instance of a given structured type or if the array index is fully known.

Composite Locations and Aliases.

The same applies to the alias property: once a field may point to some location at some point of the control flow, there is no way to "kill" this afterwards. This is because assigning the field of an instance with a pointer to some other location does not guarantee that the corresponding field of all other instances are still not able to point to the original location. However, it is always safe to assume that the field (of all instances) *may* point to both the old and the new location.

```
struct str {int *a; ...} s1, s2;
int u[5], *p, a, b;
// Assume all pointers may point to nothing initially

p = &x;      // p may point to x only
u[0] = &x;   // u[i] may point to x for all i
s1.a = &x;   // Both s1.a and s2.a may point to x

p = &y;      // p may point to y only
u[0] = &y;   // u[i] may point to x or y for all i
s1.a = &y;   // Both s1.a and s2.a may point to x or y
```

Note that this happens even if there is *only one* instance of a given structured type or if the array index is fully known.

2.3.2 Alias Analysis

The alias analysis computes the set of possible targets for pointer locations and expressions. This analysis is:

flow-sensitive

The set of possible targets for a given pointer location may differ from one statement to another.

context-insensitive

There is a single alias analysis for each function, *i.e.* a single alias description for each function, not one per call site.

2.3.3 Side-Effect Analysis

The side-effect analysis computes the set of non-locals variables that are read or written in each function. This analysis is:

flow-sensitive

The set of possible effects depends on the control flow.

context-insensitive

There is a single side-effect analysis for each function, *i.e.* no particular context depending on the call site is exploited. It relies only on alias information, which is computed on a context-insensitive basis as well.

2.3.4 Binding-Time Analysis

The binding-time analysis annotates the program with binding-time information. This analysis is:

flow-sensitive
: The binding time of a given location may differ from one statement to another.

context-sensitive
: There is a separate binding-time analysis for each call site, depending on its binding-time context. This is also called (function) *polyvariance*.

return-sensitive
: The BTA distinguishes between the binding time of the body of a function (static if the whole body is static, dynamic otherwise) and the binding time of its return value. With this approach, if a function performs some dynamic side effects but returns a static value, the static value can be exploited at the call site. The ability of the BTA to make this distinction is called *return sensitivity*.

sensitive to partially-static structures
: A structure may have both static and dynamic fields at the same time.

use-sensitive
: The binding time of a location is affected not only by the binding times of locations that it depends on, but also by the uses of the location. This information is actually computed by the evaluation-time analysis.

monovariant with respect to arrays, possibly polyvariant w.r.t structures and unions
: There is a single location for all cells of an array; see Array Monovariance. There is a single location of all instances of a given structured type (structure or union), or possibly many; see Structure Mono/polyvariance.

2.3.5 Evaluation-Time Analysis

The evaluation-time analysis make the binding-time analysis use-sensitive:

- It makes dynamic all static expressions that appear in a dynamic context and are not representable as C program text (*i.e.* pointers, structures and arrays): ETA1.
- It makes dynamic all the static definitions of variables with dynamic uses: ETA2.
- If there is no static use of a definition, the definition is turned totally dynamic as the static facet is not needed: ETA2.

Precision for evaluation times in the ETA is the same as the precision for binding times in the BTA, *i.e.* the ETA is flow-sensitive, context-sensitive, return-sensitive, sensitive to partially static structures and monovariant with respect to arrays, structures and unions.

2.4 Configuring the Analyses

Tempo is an *off-line* specializer. This means that the specialization is preceded by an pre-processing phase. This pre-processing phase is divided into a sequence of program analyses that compute properties about locations and resulting specialization transformations:

- the alias relation between locations,
- the binding times of program locations,
- the specialization actions, *i.e.* the transformations to perform at specialization time.

The result of the analyses can be visualized in order to assess the degree of specialization of the program.

In order to parameterize the analyses, the user must provide Tempo with the following compulsory information.

1. **A program**, i.e. a *single* legal ANSI C file.
2. **An entry point**, i.e. the root of the call-graph, i.e. the name of the function to specialize.
3. **Initial binding times for the arguments of the entry point**.

In addition, the user *may* provide the following information.

1. **Initial binding times of global variables**. By default, all uninitialized global variables are assumed dynamic.
2. **Initial binding times of arrays**. This is necessary because of the array monovariance of the analyses. By default, all arrays are assumed dynamic.
3. **Initial binding times of structures and unions**. This is necessary because of the default structure (and union) monovariance of the analyses. By default, all fields are assumed dynamic.
4. **The binding time of external functions calls**, *i.e.* the permission or prohibition to evaluate external functions (if any) at specialization time if all the arguments are static,
5. **The binding time of unknown indirect function calls**, *i.e.* the permission or prohibition to evaluate external indirect function calls (if any) at specialization time if all their arguments are static,
6. **The behavior of external functions** with respect to aliases and binding times. By default, external functions are assumed not to have any effect on aliases or binding times *at all*.
7. **The binding time of parameters passed by reference** (for the entry-point function). The standard binding-time specification for the arguments describes just the arguments themselves, not the values they point to. By default, pointed scalar values have the same binding time as the pointer and pointed composite values are dynamic (*i.e.* default binding time rules for structures, unions and arrays apply).
8. **Initial alias relations** among pointer parameters (actual arguments and global variables), if any. By default, it is assumed that all locations are unrelated.
9. **Live locations after the entry point** is called in the original application. By default, no location is assumed to be live at the end of the program.

2.4.1 Specifying the Program to Specialize

The program to specialize is provided to Tempo as a *single* legal ANSI C file. It is passed through `cpp` (the C pre-processor) before parsing. Consequently, it may contain statements like `#include`, `#define` and `#ifdef`. When visualizing annotated source code with Tempo, you will see the result of the C pre-processor expansion, not the original code.

The specification of the program to specialize is done in two steps.

1. The user must specifying the *working directory*, which is the directory where the program lies. In order to do so, the user may use either the `wd` variable, as in

   ```
   wd := "/home/jake/spec/misc";
   ```

   or the `cd` command, as in

   ```
   cd "/home/jake/spec/misc";
   ```

   which also sets the variable `wd`. By default, the working directory is the directory in which Tempo is run, unless the shell variable `TEMPOWORK` is defined.

2. Then, in order to run any phase, the name of the program must be used each time, as in

   ```
   an "fft";
   cs "fft";
   sp "fft";
   ```

   This runs the analyses and specialization of the program `fft.c`, located in the current working directory. Note that you must specify the file prefix only, not the suffix (*i.e.*, `"fft"`, not `"fft.c"`).

   Note also that several programs may reside in the same working directory. Thus, you may run:

   ```
   sp "fft";
   an "power";
   ```

   if both `fft.c` and `power.c` lie in the current working directory.

In practice, step 1 is done once, as long as you work on the same program file, and step 2 is performed many times.

2.4.2 Entry Point and Binding Times For Its Arguments

The entry point and the binding-time information of its arguments are defined by the SML variable `entry_point`. This variable must be defined in the `config.sml` file (*e.g.* `fft.config.sml`), in the same directory as the `c` file (*e.g.* `fft.c`). Assigning it at the SML top level has no effect and the default value ("") is not meaningful. In the following example

```
entry_point := "foo(S,D)";
```

the function `foo` is the entry point. It has two arguments: the first one is initially static, the second one is initially dynamic.

Valid binding-time specifications for the entry-point parameters are:

`S` : scalar static argument
`D` : scalar dynamic argument
`_` : uninitialized or non-scalar argument

If an argument is a structured type (union or structure), the specified binding time must be `_` and the actual binding time must be provided separately (see Binding Time of Structures and Unions below). If the argument is a pointer (hence a scalar), the specified binding time is for the pointer, not the pointed value. See below for specifying binding time of pointed values.

Specifying Several Entry Points

For the time being, Tempo is restricted to a *single* entry point and binding-time information definition. If you wish to perform specialization for *multiple* entry points (e.g., a whole library), you have to explicitly (i.e. manually)

1. **Build a dummy entry point that calls all sub entry points.** This is typically in a switch or a cascade of ifs so that the call to one entry point doesn't influence the calls to the others. It is up to you to propagate relevant binding time information from the dummy entry point to each sub entry points.
2. **Extract specialized entry points.** There is some additional work to do on the resulting specialized file.
   i. At the moment, in a specialized program file, only the specialization(s) of the entry point is/are declared `extern`; all other specialized functions are made `static` (in the C sense). You have to manually turn those specifiers into `extern`.
   ii. Furthermore, it is only possible to specify the name of the specialization of the actual entry point (using the variable `specialized_entry_point_name`). The sub entry points have to be renamed by hand, if desired.

   You must also make sure that sub entry points are not inlined into the dummy entry point.

2.4.3 Binding Time of Global Variables

The initial binding time for global variables is specified using the SML variable `static_locations`. The default is that no global locations are static, *i.e.* `static_locations` is the empty list. As an example, the following definition specifies that the global variables `x` and `y` are static; all other global variables are considered dynamic.

```
static_locations := ["x", "y"];
```

It is not necessary to specify the binding times of local variables (including formals), as these are determined automatically by their use.

2.4.4 Binding Time of Arrays

Because of array monovariance, there is only one location associated with an array. This location can be declared static using the SML variable `static_locations` as well, meaning that all cells of the array are static. *E.g.*, an array `int a[5]` can be declared static as follows.

```
static_locations := ["a"];
```

Note that only the name of the array is used, without the brackets. Note also that the pointer to the array (*i.e.* `a` itself, meaning `&a[0]`) is a constant, and hence is always static.

2.4.5 Binding Time of Structures and Unions

Unlike binding times for scalar variables which are "per variable", binding times for a given structured type are the same for *any* instance: it is a "per type" binding time. (See Structure Mono/polyvariance above.)

For example, if the C program contains the following declaration,

```
struct pt {
  int x;
  int y;
} point1, point2;
```

then specifying

```
static_locations := ["pt.x"];
```

with the following C declaration in the program file

makes both `point1.x` and `point2.x` static. An error will be reported if you try to specify

```
static_locations := ["point1.x"];
```

In other words, you *must* provide the name of the type for the fields of a structure or a union, *not* the name of an instance.

Structures and Unions Fields

If a whole structure or union is static, each field must be declared as such individually. For example, specifying

```
static_locations := ["pt.x","pt.y"];
```

with the following C declaration in the program file

```
struct pt {
  int x;
  int y;
} point1, point2;
```

will make both `point1` and `point2` totally static. An error will be reported if you try to specify

```
static_locations := ["pt"];
```

This is because there is no location associated to a structured type; the only locations are for each field.

Nested Structures and Unions

Let us consider the following global declarations:

```
struct pt2d {
  int x;
  int y;
};

struct pt3d {
  struct pt2d pt;
  int z;
};

struct pt2d u;
struct pt3d v;
```

Because the `pt` field of a `pt3d` structure has a structure type, trying to directly declare `pt3d.pt` or `pt3d.pt.x` to be static will raise an error. The only locations that can be declared static are `pt2d.x`, `pt2d.y` and `pt3d.z`. For example, the following makes `u` and `v.pt` static:

```
static_locations := ["pt2d.x", "pt2d.y"];
```

The following makes `u` and `v` static:

```
static_locations := ["pt2d.x", "pt2d.y", "pt3d.z"];
```

Anonymous Structures or Unions

If the definition of your original structure or union is anonymous (this is common with nested structures) like this

```
struct /*no name here*/ {
  int x;
  int y;
} point;
```

then SUIF will automatically generate a dummy name:

```
struct __tmp_struct1 {
  int x;
  int y;
} point;
```

As the name is generated automatically with an unknown suffix, it is not safe to rely on SUIF to pick a particular name for you. When is possible, it is better to provide an explicit name in the original file.

Restrictions on Typedef

Names of structured types defined using `typedef` cannot be used to specify a location. For example, with a type definition like this

```
typedef struct pt {
  int x;
  int y;
} point;
point point1, point2;
```

You cannot specify

```
static_locations := ["point.x"];
```

You have to write

```
static_locations := ["pt.x"];
```

2.4.6 Binding Time of External Function Calls

The SML variable `external_functions` is used to specify the binding times of external function calls. For each function, you may either force all calls to be residualized (*i.e.* dynamic) or to be evaluated (*i.e.* static) provided the arguments allow it (*i.e.* if they are all static).

Evaluation is often allowed for library functions, whereas residualization can be required to prevent I/O side-effects at specialization time (such as printing).

E.g., the following declaration means that all calls to the external functions `foo` and `bar` are residualized, regardless of the arguments or the possible abstract definitions are; all calls to other external functions are evaluated if all their arguments are static.

```
external_functions := RESIDUALIZE["foo","bar"];
```

This declaration has the following impact on the binding-time analysis.

```
foo(stat,dyn);   // Natural residualization of the call
foo(stat,stat);  // Forced residualization of the call
gee(stat,dyn);   // Natural residualization of the call
gee(stat,stat);  // Allowed evaluation of the call
```

If you have more external functions to residualize than to evaluate, you may want to use this other form, instead:

```
external_functions := EVALUATE["foo","bar"];
```

The above declaration means that all calls to the external functions `foo` and `bar` should be evaluated if their arguments are static; all calls to other external functions are always residualized. *I.e.*

```
foo(stat,dyn);   // Natural residualization of the call
foo(stat,stat);  // Allowed evaluation of the call
gee(stat,dyn);   // Natural residualization of the call
gee(stat,stat);  // Forced residualization of the call
```

The default treatment is to residualize all calls, *i.e.* to evaluate none of the external functions: `EVALUATE[]`.

2.4.7 Binding Time of External Indirect Function Calls

The boolean variable `residualize_all_icalls` is used to specify the binding time of indirect external function calls. If it is `true`, indirect calls to unknown external functions are always residualized (independently of their arguments). If it is `false`, they are evaluated if possible (*i.e.* function pointer and arguments are static) at specialization time.

This variable is only useful when the phase to eliminate indirect function calls cannot totally transform some indirect calls. This usually happens when not enough information is given to the alias analysis: some unknown function pointer (provided as a global variable or as an argument to the entry-point function, or returned by an external call) may be used for an indirect call.

In the following program, `fp` may point to `f1`, `f2`, or `*gfp`, for which alias information is not provided.

```
int f1(int, int), f2(int, int), (*gfp)(int, int);

int entry()
{
  int (*fp)(int,int);

  if (c == 1)
    fp = f1;
  else if (c == 2)
    fp = f2;
  else
    fp = gfp;

  // fp may point to f1, f2 and ... *gfp

  return (*fp)(a,b);
}
```

The phase that eliminates external function pointers rewrites this program as follows:

```
int entry()
{
  if (c == 1)
    fp = f1;
  else if (c == 2)
    fp = f2;
  else
    fp = gfp;

  return _apply_1(fp,a,b);
}

int _apply_1(int (*_q)(int,int)), int _a1, int _a2)
{
  if (_q == f1)
    return f1(_a1,_a2);
  else if (_q == f2)
    return f2(_a1,_a2);
  else
    return (*_q)(_a1,_a2);
}
```

This rewriting enables possible specializations of `f1` and `f2`, even though they are used through function pointers. However, the global variable `gfp` might point to some unknown function `f3`. That is why a default case `(*_q)(_a1,_a2)` remains in `_apply_1`. If the alias analysis can determine that the only possible targets are known functions (as here, `f1` and `f2`), there is no such default case.

The remaining indirect external call

```
(*_q)(_a1,_a2);
```

can be forced to be dynamic or allowed to be static (providing all arguments are static), using the `residualize_all_icalls` variable. With

```
residualize_all_icalls := true;
```

the binding-time analysis yields only dynamic indirect calls, even if `_q`, `_a1` and `_a2` all are static

```
(*_q)(_a1,_a2);   // Note: static pointer _q later
(*_q)(_a1,_a2);   //       turned dynamic by eta1
(*_q)(_a1,_a2);   // Ditto
```

whereas with

```
residualize_all_icalls := false;
```

the binding-time analysis may yield a static (*i.e.* evaluated at specialization time) indirect call if both the function pointer and the arguments are static:

```
(*_q)(_a1,_a2);   // Rediidualized because _a1 is dynamic
(*_q)(_a1,_a2);   // Rediidualized because _q is dynamic
(*_q)(_a1,_a2);   // Evaluated
```

There is no way to specify an imposed or allowed binding time per indirect call site, not even per generated `_apply_`*n* function.

2.4.8 Behavior of External Functions

By default, external functions are supposed not to have any action. Consider for example a pointer argument to an external function call that has not been given an abstract definition:

```
ch = 'X';          // 'X' is static
read(fd,&ch,1);    // Call to read does not affect ch
t = ch;            // Considered static instead of dynamic
```

The pointer `&ch` is read at the call site, but the content `ch` is considered unaffected, not even read. This is not specific to binding times: a similar case can be constructed regarding aliases. The problem is that the default void behavior for external functions is not safe.

In Tempo, the behavior of external functions with respect to aliases and binding times is expressed using abstract functions, *i.e.* some C code that mimics the actual implementation of external functions. The code of those abstract functions is used during the analyses to simulate the real behavior and yield the proper analysis results. However, at specialization time, only the actual values and implementations are used. In other words, abstract function are not specialized; calls to them may either be residualized or evaluated using the real implementation.

Example

Consider for example this implementation of the `strcpy()` function.

```
char *strcpy(char *dest, const char *src)
{
  char *orig_dest;
  orig_dest = dest;
  do
  {
    *dest++ = *src;
  }
  while (*src++ != '\0');
  return orig_dest;
}
```

This function copies the value of the `src` array into the `dest` array. No aliases are created. The alias and binding-time behavior of this function may be modeled by the following abstract function.

```
char *strcpy(char *dest, const char *src)
{
  *dest = *src;
  return dest;
}
```

Because of the monovariance of arrays, we achieve the effect of copying the contents of an array into another by copying a single cell. We also model the return behavior of `strcpy` by returning the `dest` pointer.

As a result, the binding-time analysis of the program file yields, for example, the following propagation of dynamic values (assuming `q` points to an array with dynamic content):

```
x = p[0];     // Static pointer to static array
strcpy(p,q); // Copy from static pointer to dynamic array
y = p[0];     // Static pointer to dynamic array
```

or the preservation of static computation (assuming `q` points to an array with static content):

```
x = p[0];     // Static pointer to static array
strcpy(p,q); // Copy from static pointer to static array
y = p[0];     // Static pointer to static array
```

Similarly, the alias analysis exploits the fact that the returned value of `strcpy` is a pointer to the same targets as the first argument.

```
char a[5], b[5], c[5], d[5], *p, *q, *r;
p = c1 ? a : b;
q = c2 ? c : d;
x = p/* a[], b[] */[0];   // p may point to arrays a or b
y = q/* c[], d[] */[0];   // q may point to arrays c or d
r = strcpy(p,q);
z = r/* a[], b[] */[0];   // r may point to arrays a or b
```

This model is correct with respect to an analysis that does not make any difference for different array cells (as is the case for Tempo). This is why the loop that is present in the original `strcpy()` function does not need to be represented in the abstract function. If the analysis was finer, *e.g.* interval analysis and separate attributes for array segments, a loop would be needed in order to express the (potentially partial) overwriting of `dest` with `src`.

Providing Abstract Functions

In order to provide abstract functions definitions to Tempo, just write them in a separate abstract function file, *i.e.* suffixed with `actx.c`. Thus, if the program file is `foo.c`, the optional abstract function file should be `foo.actx.c`.

When an abstract function is provided, Tempo automatically uses the abstract function for analysis and the actual function for specialization. Note that, additionally, you still may have to use variable `external_functions` in order to specify whether actual functions may be called at specialization time (see Binding Time of External Function Calls above).

In practice, the content of the program file and the abstract function file (`c` and `actx.c`) are merged and analyzed together. (There is a special treatment for functions defined in the abstract function file though.) When visualizing the result of the analyses, you can see abstract functions as well. However, all abstract functions are removed during action analysis, *i.e.* the last analysis phase, which yields an action-annotated program without any trace of abstract code.

Modeling Dynamic Memory Allocation

As explained above, the memory model of Tempo does not handle memory cells that have no name, such as cells generated by dynamic memory allocation. Another typical use of external functions is to bestow (predefined) names on such memory cells by providing a function that models memory allocation. *E.g.*,

```
char dummy_location[1];

char *malloc(size_t size)
{
  return dummy_location;
}
```

The above abstract definition returns the same named memory cell, *i.e.* `dummy_location[]`, for any call to `malloc()`. At specialization time, this abstract definition is removed; the actual `malloc()` function is used if static, or else residualized (see Binding Time of External Function Calls above).

Returning the same location for each call to `malloc()` will merge all effects concerning that location. For a finer analysis, you may want to define and use a different memory allocation routine per call site, each one returning locations with a different name.

Moreover, calls to `malloc()` are often followed by a cast. Because it is not totally safe to use casts in Tempo (especially for structured types), it is safer to define at least a "per type" allocation routine, *e.g.*

```
struct foo dummy_struct_foo;

struct foo *malloc_struct_foo(size_t size)
{
  return &dummy_struct_foo;
}
```

Note that it is useless to have several memory allocation routines for the same structured type with monovariance: all named structures (or unions) will share the same analysis properties anyway.

Safety of Abstract Functions

You must be careful in writing an abstract function as Tempo can do nothing but assume it models the original function correctly.

Moreover, you must keep in mind the precision of the analyses: there does not exists a single good abstraction for a function that is independent of the analyses. Sometimes an abstract definition might be too precise with respect to present analyses.

Consider for example the above abstract definition for the function `malloc()`. In this definition, it is necessary for `dummy_location` to be an array for safety reasons. If it is a scalar, as in the following abstract definition,

```
char dummy_location;

char *malloc(size_t size)
{
  return &dummy_location;
}
```

this is not be safe as one could write (pink color is for a static & dynamic binding time) the following program.

```
p = malloc(1);   // p points to dummy_location
*p = dynexp;     // dummy_location becomes dynamic
q = malloc(1);   // q points to dummy_location
*q = 's';        // dummy_location becomes static
ch = *p;         // Considered static instead of dynamic
```

Note that it is always safe to consider an expression dynamic instead of static. (After all, the BTA, as most analyses, just computes an approximation.) However, it is not safe to consider static an expression that depends on a dynamic value.

The problem above comes from the fact that the assignment `*q = 's'` acts as a killing definition for the binding time of `dummy_location`. This situation does not arise with an array definition for `dummy_location` as, according to the memory model, the analysis conservatively merges binding times of all cells (*i.e.* array monovariance). Hence, with this definition,

```
char dummy_location[1];

char *malloc(size_t size)
{
  return dummy_location;
}
```

the program is analyzed as follows

```
p = malloc(1);   // p points to dummy_location
*p = dynexp;     // dummy_location becomes dynamic
q = malloc(1);   // q points to dummy_location
*q = 's';        // dummy_location stays dynamic
ch = *p;         // Considered dynamic
```

Now `*p` is safely considered dynamic.

Standard Tricks for Specifying Aliases and Binding Times

To finish with generalities concerning abstract functions, here are some standard tricks that are commonly used to express various binding-time or alias behaviors in abstract function files. The section Specifying Complex Analysis Contexts below contains other typical uses.

Dummy Dynamic Location

To make a dynamic location, you can declare a global location in the `actx.c` abstract function file; global locations are dynamic by default (see the variable `static_locations`). Alternatively, calling a dummy external function returns a dynamic value by default (see the variable `external_functions`). *E.g.*

```
int dyn;
foo()
{
  x = dyn;     // Globals are dynamic by default
  y = fdyn(); // So are external calls
}
```

Dummy Static Location

To make a static location, you can declare a location in the abstract function file and initialize it to a constant value (or declare it static in the `config.sml` configuration file). *E.g.*

```
int stat = 0;  // Or add stat in static_locations
foo()
{
  x = stat;
}
```

Turn a Location Dynamic

A location may be turned dynamic either by assigning a dynamic value to it or by making an assignment under dynamic control, *e.g.*

```
if (dyn)   // Any dynamic condition
  x = x;   // Even if x is static,
use(x);    // now x is dynamic
```

Pointer to a Set of Locations

To make a dynamic (resp. static) pointer to a set of locations, you can assign the pointer to one of the required targets depending on a dynamic (resp. static) condition, *e.g.*

```
{
  if (cond)   // p gets the same binding time as cond
    p = &x;   //   p may point to x
```

```
      else        // or
        p = &y;   //   p may point to y
    }
```

Now `p` points to either `x` or `y`. Furthermore, the variable `p` has the same binding time as the expression `cond`.

Note that `cond` should not be a constant because SUIF unconditionally eliminates dead code when parsing code such as `if(0)` or `if(1)`. A simple variable, as used in `if(cond)` above, does the trick.

Note also that the ifs are necessary because assignments to pointers may kill previous targets. *E.g.*, after

```
    {
      p = &x;   // p may point to x only
      p = &y;   // p now may point to y only
    }
```

variable `p` may point to location `y` only.

2.4.9 Specifying Complex Analysis Contexts

When only a module of a larger application is to be specialized (modular specialization), the user must specify the exact analysis context of the module. This description includes information about the execution context both before and after the entry point is called from the larger application.

If the parameters (formals of the entry point and globals) are simple scalars, the variables `entry_point` and `static_locations` are enough to provide Tempo with the relevant initial binding time context (see Entry Point and Binding Times For Its Arguments and Binding Time of Global Variables above). This covers a lot of common cases.

For specifying more complex contexts, *i.e.* initial alias relations, binding times of parameters passed by reference, or live locations after calling the entry point, another (less declarative) mechanism is provided, named context abstract functions.

In addition to abstract external functions, the abstract function file (*i.e.* suffixed with `actx.c`) *may* contain:

- a function `void set_analysis_context()`, whose arguments are pointers to the entry point's arguments, to be used for setting the initial analysis context, *i.e.* the execution context just *before* calling the entry point in the original application.
- a function `void set_post_analysis_context()`, whose arguments are pointers to the entry point's arguments, to be used for setting the uses of non-locals of the entry point, *i.e.* the execution context *after* calling the entry point in the original application.

If a function `set_analysis_context()` or `set_post_analysis_context()` is defined in the abstract context file, Tempo will automatically create a new (dummy) program entry point. This new entry point has the same interface as the previous one and does three things:

1. It calls the `set_analysis_context()` function (if any) with arguments that are pointers to the original entry-point arguments.
2. It calls the original entry point.

3. It calls the `set_post_analysis_context()` function (if any) with arguments that are pointers to the original entry-point arguments.

For example, starting from a program file `bar.c` with content

```
char foo(int a, struct baz *b) { ... }
```

and an abstract context file `bar.actx.c` with content

```
void set_analysis_context(int *p_a, struct baz **p_b)
{ ... }
void set_post_analysis_context(int *p_a, struct baz **p_b)
{ ... }
```

Tempo actually analyses merge the two files and considered as the new entry point the following generated function.

```
char dummy_entry_point(int a, struct bar *b)
{
  char retval;

  set_analysis_context(&a,&b);
  retval = foo(a,b);
  set_post_analysis_context(&a,&b);
  return retval;
}
```

Do note that abstract context functions `set_analysis_context()` and `set_post_analysis_context()` take as arguments pointers to the real entry-point arguments. The body of those functions may contains any C code in order to model the proper execution context (see Behavior of External Functions).

Note also that the dummy entry-point function, as well as `set_analysis_context()` and `set_post_analysis_context()`, are only used in the main analysis stages. They are removed at the beginning of the phase that performs the flattening of static returns (return sensitivity processing) and replaced by the original entry point. In particular, there remains no trace of them in the action-annotated program.

The following sections give various examples showing uses of context abstract functions.

Binding Times of Parameters Passed By Reference

When declaring a pointer static or dynamic, the binding time concerns the address only, not the content. By default, pointed scalar values have the same binding time as the pointer and pointed composite values are dynamic (*i.e.* default binding-time rules for structures, unions and arrays apply). The exact binding time of the content can be specified as follows.

Pointers to Structures or Unions

Because of monovariance, the binding time of structures and unions are declared for all instances and must be stated separately (see Binding Time of Structures and Unions above). This is the case even when the structure or a union is first given as a pointer. With polyvariance enabled, the binding time of structure instances may still differ though.

Pointers to Scalars

Declaring a pointer to static or dynamic scalar can be specified using the context abstract function `set_analysis_context()`. Consider for example the following program file `foo.c`

```
int a, b, *q1, *q2, *q3, *q4;

entry(int *q, int x) { ... }
```

and the `foo.actx.c` file.

```
int dyn1, dyn2; // Dummy dynamic locs
int stat = 0;   // Dummy static (thanks to the init) loc

                     // Declarations from foo.c are needed
extern int *q1,*q2; // only for the variables that are
                     // actually used in foo.actx.c

void set_analysis_context(int **p_q, int *p_x)
{
  // p_q is a pointer to the real (pointer) argument q
  // p_x must be present though unused

  if (dyn1)
    q1 = &dyn1; // q1 dynamic ptr to some dynamic loc
  if (dyn1)     // q1 may also point to some other
    q1 = &dyn2; //   dynamic loc
  q2 = &dyn1;   // q2 static ptr to some dynamic loc
  q4 = &dyn2;   // q4 static ptr to some other dynamic loc
  *p_q = &stat; // q static ptr to some static loc
}
```

Note that, in the above example, `q1` may point to two locations whereas `q2` and `q4` may point only to a single (different) locations. It is important here to provide a faithful information (*i.e.*, same or different location) because it may have an impact on the analysis. For example, if `q2` and `q4` were pointing to the same location `dyn1`, then assigning a static value to `*q2` would make `*q4` static as well. (`*q1` would still not be static though, because it could point to `dyn2` as well.)

Note also that the above specification concerning `q1` is not equivalent to the following one:

```
  if (dyn1) {
    q1 = &dyn1; // q1 points to dyn1 only
    q1 = &dyn2; // q1 now points to dyn2 only
  }
```

Indeed, because the assignment act as a killing definition, only the latest one is taken into account in the final result. (See Behavior of External Functions: pointer to a set of locations.)

Pointers to Strings and Arrays

As is the case for structures and unions, the binding time of arrays (*e.g.* strings) must be declared explicitly because of array monovariance. However, the binding time is not "per type" but "per actual array". Hence the above scheme can be applied to specify the binding time of pointed arrays.

There is a trick though, because SUIF unconditionally turns array arguments into pointers. As a result, type information is lost, which can make the analysis unsafe. Let us consider an entry point `f` defined as follows:

```
int f(int a[])
{
  a[0] = 3;
  return a[1];
}
```

This program is translated by SUIF into:

```
int f(int *a)
{
  *a = 3;
  return a[1];
}
```

As a result, Tempo cannot tell any longer whether `a` points to an scalar integer or to an array. Now assume the array `a` is dynamic. In the first case, if `a` points to an scalar integer, the statement `*a = 3` acts as a killing definition and the location `a` is made static. As a result, `a[1]` is considered static as well, which is wrong (*i.e.* unsafe). In the second case, if `a` is considered to point to an array, the statement `*a = 3` does not act as a killing definition and the location `a` is stays dynamic because of array monovariance. In order to have that second, correct behavior, you can define the following context abstract function.

```
int dummy_array[1];

void set_analysis_context(int **p_a)
{
  *p_a = dummy_array;
}
```

This actually defines argument `a` of `f` as a static pointer to a dynamic array. If the array is static, just add `dummy_array` in the variable `static_locations`. Note also that the above context abstract function forces the binding time of the argument of `f` to static, even if the variable `entry_point` specifies `f(D)`. This is because `dummy_array` is a constant (hence static) pointer. If the pointer to the array is dynamic, you may define:

```
int dyn, dummy_array[1];

void set_analysis_context(int **p_a)
{
  if (dyn)                // Dynamic condition makes
    *p_a = dummy_array;   // pointer p_a dynamic
}
```

Initial Alias Relation

Initial alias relations between locations can be specified using `set_analysis_context()`. Suppose we want to specialize the following procedure, which has been extracted from a larger program. Let `dyn` be some dynamic expression and assume arguments of `f` are all static pointers to static locations.

```
int a;              // Assume a is static initially
f(int *x,int *y)    // Static ptrs to static locations
{
  if (dyn)            // The dynamic test makes *x
    *x = *y - a;      //   dynamic after the condition
  c = (*y + a) / *x;  // Is (*y + a) static ?
}
```

Without more information, Tempo will consider that expression `*y + a` is static, so that it can be replaced by a constant during specialization. This is unsafe as function `f` could be called in the larger program (*i.e.* the application) with one of the following contexts:

```
f(&a, &b);
f(&b, &b);
```

In the first case, making `*x` dynamic also makes `a` dynamic because of the aliasing relation. In the second case, making `*x` dynamic makes `*y` dynamic as well.

We can specify these properties using the context abstract functions and thus yield a safe specialization. For example, the second case above can be specified by the following `set_analysis_context()` which states that the arguments `x` and `y` of `f` point to the same static cell.

```
int dummy = 0;       // Any static value

void set_analysis_context(int **p_x, int **p_y)
{
  *p_x = *p_y = &dummy; // *p_x and *p_y refer to the same thing
}
```

Live Locations After The Entry Point

Live locations refer to locations that are still live at the end of the program being specialized. This can happen during modular specialization, when the program fragment is extracted from a larger program and this fragment is reinserted after specialization. Consider for example this function

```
void foo(int a, int b)
{
  x = x + y;
  y = x + b;
  z = x + a;
}
```

used in the following context in the larger application.

```
x = 1;
y = 2;
z = dyn;
foo(4,z);
bar(x,y,z);   // Called with arguments (3,3+dyn,7)
```

In this context, function `foo` can be specialized with a binding-time context where: `a`, `x` and `z` are static; `b` and `y` are dynamic. The binding time analysis then yields:

```
void foo(int a, int b)
{
  x = x + y;     // Static assignment
  y = x + b;     // Dynamic assignment
```

```
    z = x + a;     // Static assignment
  }
```

which leads to the following specialization:

```
void foo_1(int b)
{
  y = 3 + b;
}
```

However, it is not correct to replug this function in the original application.

```
x = 1;
y = 2;
z = dyn;
foo_1(z);
bar(x,y,z);  // Called with arguments (1,3+dyn,dyn)
```

The problem comes from the fact that Tempo has not been instructed that locations `x`, `y` and `z` are used after the entry point is called. Tempo has assumed that they were *not* used and has specialized away all the static locations (that is why only `y` remains).

The variable `live_locations` is used to indicate to Tempo that some locations are live after the entry point is called. (Syntax and constraints on locations are the same as for variable `static_locations`.)

```
live_locations := ["x", "y", "z"];
```

Using this information, Tempo now produces the following binding times.

```
void foo(int a, int b)
{
  x = x + y;     // Static & dynamic assignment
  y = x + b;     // Dynamic assignment
  z = x + a;     // Dynamic assignment
}
```

which leads to the following specialization:

```
void foo_2(int b)
{
  x = 3;
  y = 3 + b;
  z = 7;
}
```

It is now correct to replug this function in the original application.

```
x = 1;
y = 2;
z = dyn;
foo_2(z);
bar(x,y,z);  // Called with arguments (3,3+dyn,7)
```

Live locations may be thought of as dynamic uses after calling the entry point. It is actually implemented using the backward propagation of dynamic uses performed by the evaluation-time analysis phase, *i.e.* ETA2.

It is equivalent to specify live locations using variable `live_locations` or to model a use of those locations in the `set_post_analysis_context()` function. *E.g.*, in the above example, live locations could also have been specified using this definition in the abstract context file:

```
int dyn;                // Dynamic by default

void set_post_analysis_context(int *a, int *b)
{
  do { x=x; y=y; z=z; }  // Turns them dynamic
  while (dyn);           // and make dynamic uses
}
```

In practice, variable `live_locations` covers most common cases. The function `set_post_analysis_context()` is used for more complex cases, especially when aliases are involved, as it is easier to rely on Tempo's analyses to simulate the exact dynamic uses. In this case, some C code from the application can be copy/pasted (and possibly abstracted) into the function `set_post_analysis_context()`.

New alias relations occurring after the entry point is called do not actually have an impact on analysis and modular specialization. However, they may imply different uses of locations, which does have an impact on specialization.

2.5 Visualization

Files containing the result of intermediate stages of the analysis have extensions ".as" and ".at", *e.g.* `fft.bta.as`, `fft.at`. They are not kept by default, except the ".at" file which is the final result of all the analysis phases.

The results are also stored as commented colored C files (generated by default) that can be used to visualize the outputs of the analysis, *e.g.* `fft.bta.color`, `fft.at.html`. In those colored files, the main items for understanding the result of the analyses are:

- Annotations for alias information: possible targets for each pointer expression.
- Colors for binding-time information: expressions that are static, dynamic, etc.
- Colors for evaluation-time information: expressions that should be static, dynamic, etc.
- Colors for actions: transformations to perform at specialization time.
- Function polyvariance: analysis of the same function in different contexts.
- Function call site: binding times and polyvariance index.
- Function definition: binding times and polyvariance index.
- Function signature: side effects (read and written non-locals) and live uses.
- Entry-point signature: summarized signature of the entry point.

2.5.1 Colored Files

Color files show the result the analyses. Each color file has a legend near the top describing the colors used in the file. The colors have similar meanings (*i.e.* binding-time and evaluation-time information) in most of the files, except the `at.color` file, showing the results of the action analysis. We describe the colors used in this file separately.

For visualizing those files, you can choose between two colored files formats:

- the HTML format, that can be displayed using any any web browser,
- the MIME text/enriched format, than can be displayed using GNU emacs or xemacs.

The Tempo `viewer` variable let you choose between those two formats. It may be set to either `html` or `emacs` triggering the generation of HTML files (suffixed with `.html`) or MIME text/enriched format files (suffixed with `.color`).

Be careful: if you choose, say, `html`, then the ".color" files that might be present in the directory will not be removed when the ".html" files are generated. This may lead to inconsistencies and confusion if later on you forget about html and read the ".color" files (that will not be up-to-date).

In the following explanation, we assume that `viewer` is set to `emacs`. If `viewer` is set to `html`, the HTML files generated are colored similarly.

If you have troubles visualizing ".color" files with emacs, consider reading the Emacs Installation Issues in the Installation Manual.

See also:

- Variable: `max_width` (width of displayed code)
- Variable: `max_decls_size` (threshold for declarations ellipsis)
- Variable: `print_dir_in_html_title` (display path in HTML title)

2.5.2 Text For Alias information

Recall that the analyses operate on the output of SUIF. Hence the color files contain the results of transformations performed by SUIF. In particular, note that SUIF unconditionally performs the following translations:

```
ptr->field     =>   (*ptr).field

                    suif_tmp42   =  ptrexp;
... *ptrexp ...  => 
                    ... *suif_tmp42 ...
```

where *`ptrexp`* is some pointer expression (not just a variable). Hence any dereference is explicit and applies only to a pointer variable.

Alias information appears in the C output files as comments after each pointer dereference:

```
*ptr/* loc1, ..., locN */
```

where *loc1, ..., locN* is the list of locations where "`ptr`" *may* point to, *i.e.* it is a superset (safe approximation) of possible targets. Those locations may be variable (including pointers) locations, array (contents) locations, structure locations, or (structure) member locations (remember that all instances of a structure type have the same binding time.) Here is how you can read alias information:

`*ptr/* var */`
    `ptr` may point to variable `var`,
`*ptr/* a[] */`
    `ptr` may point to any cell of array `a`,
`*ptr/* {s} */`
    `ptr` may point to any structure of type `struct s`,

```
*ptr/* s.f */
```

`ptr` may point to field `f` of any structure typed `struct s`.

If the pointer variable has not been initialized (it may point to nothing yet), we consider that it points to... what it points to!:

```
*ptr/* *ptr */
```

Thus, the alias information is never empty.

See also:

- Variable: `verbose_aliases` (display alias information)
- Variable: `verbose_aliases_in_specializations`
- Variable: `verbose_callsig` (display polyvariance index)
- Variable: `verbose_headers` (display signature information)

2.5.3 Colors For Binding-Time and Evaluation-Time Information

The legend in ".bta.color" files, that express the results of the BTA phase, is:

```
/* LEGEND:  STATIC  DYNAMIC  SD_FUNC  STRUCTURE  BOTTOM
 */
```

In the subsequent phases (*i.e.*, ".color" files for "eta1", "eta2", "flsret", "flsd" and "preproc2"), the legend is:

```
/* LEGEND:  STATIC  DYNAMIC  STAT&DYN  STRUCTURE  BOTTOM
 */
```

The only difference concerns the interpretation of the pink color, *i.e.* `SD_FUNC` *vs.* `STAT&DYN`.

Blue color: Static

Blue is always used for static expressions. The color of the left-hand side of an assignment statement deserves some explanation. After the BTA (*i.e.*, in the ".bta.color" file), when the L-value is known at specialization time, the left-hand side of an assignment is considered static, regardless of the binding time of the variable before or after the assignment statement. Beginning with the ETA1 (*i.e.*, in the ".eta1.color" and subsequent files), the color reflects the evaluation time of the assignment, and thus the binding time of the variable after the assignment.

Red color: Dynamic

Red is always used for dynamic expressions and statements. The color of a function identifier (and parentheses) in a function call reflects the binding time of the body of the called function. Regardless of this binding time, if the function definition is part of the program, it will be specialized. Thus even when the call is dynamic, some specialization can occur.

Pink color: Static & Dynamic

There two meanings for the pink color, according to the phase

BTA
: The pink color in the BTA *only* refers to calls to functions that contain some dynamic code in the body and have a static return value. This property is indicated in the legend as `SD_FUNC` (static and dynamic function). Such calls will eventually be flattened in a subsequent phase.

ETA (and subsequent phases)
: The pink color in the ETA and subsequently refers to definitions (and their subcomponents) that are both evaluated and residualized. This property is indicated in the legend as `STAT&DYN` (static and dynamic expression). The use of pink in ETA files is actually ambiguous, since it is also used for calls to functions that contain a dynamic body and a static return value (as in the BTA).

Pale blue color: Structure

In the ".color" files (except the ".at.color" file), structures are considered to simply have a generic "structure" binding-time. This "structure" binding-time does not specify if the structure is totally static, partially static, or totally dynamic. It only appears if the structure is manipulated as a whole, *e.g.* assigned to another structure or passed as an argument. When a field is selected (with the `"."` dot operator), the field will be annotated with its corresponding binding time.

This color may appear as pale green, rather than pale blue.

Black color: Bottom

Black indicates a term for which no binding time can be determined. Black usually appears when local variables are used before they are assigned. If the code is not dead code, this situation may represent an error in the source program. When combined with other binding times during the BTA, the "bottom" binding time is treated as a "static" binding time, *i.e.* a static (resp. dynamic) expression combined with a bottom expression make a static (resp. dynamic) expression.

2.5.4 Colors For Actions

The legend in ".at.color" files, that express the results of the Action Analysis phase, is:

```
/* LEGEND:  EVAL  REDUCE  REBUILD  IDENTITY  STRUCTURE  EV&RES
 */
```

Blue color: Evaluate

Blue indicates a term that can be completely evaluated during specialization. Variable declarations that can be eliminated during specialization are also colored blue.

Green color: Reduce

Green indicates a term that can be simplified during specialization, but contains some dynamic components. For example, an `if` statement with a static test but dynamic code in the branches would be colored greern: after specialization, the `if` statement will be rewritten into (the specialization of) one of the two branches.

Orange color: Rebuild

Orange indicates that a term cannot be simplified during specialization, but that the term does contain some subexpressions that can be simplified. For example, in an addition expression with a static operand and a dynamic operand, the addition operator would be colored orange.

Red color: Identity

Red indicates a term that must be completely residualized. Such terms are essentially copied unmodified into the residual program. Variable declarations that are not used during specialization and will be part of the residual program are also colored red.

Pale blue color: Structure

Pale blue is used for values of structure type, as in the other analyses. (It actually hides one of two possible actions: "identity" or "evaluate & residualize"; the case "evaluate" is printed out properly.)

Pink color: Evaluate & Residualize

Pink is used for terms and variable declarations that are both evaluated and residualized during specialization. Because of the function flattening phase, there is not the second meaning of pink (in the BTA or ETA files) to indicate a function that has a static return value but some dynamic expressions in the function body.

2.5.5 Function Polyvariance (a.k.a. Context Sensitivity)

After the BTA, a function definition may appear several times in the colored file. Because the BTA is polyvariant, each occurrence of the definition corresponds to a different binding-time contexts, *e.g.* once with a static argument, once with a dynamic argument. The binding-time context includes the binding times of the read variables, as well.

In order to differentiate each variant of the function at the call site, the name of each function is labeled with an index (shown as a comment just after the function name) at the function definition as well as at the call site. Note that the index is global: each function, variant or not, gets a different index (*e.g.* f_1, f_2, g_3, g_4), as opposed to a "per-function" index (*e.g.* f_1, f_2, g_1, g_2).

Note that there exist also a degree of polyvariance in the ETA as each BTA variant may be further split into several ETA variants.

Note also that the polyvariance index is global to the current file: there is no relation between some index in the ".bta.color" file and the same index in the ".eta2.color" file.

2.5.6 Function Call Site

A function call looks like this:

```
bar/*7*/(statexp, dynexp, (sdexp))
```

The corresponding binding-time information can be read as follows.

Function name and actuals' parentheses: `bar(...)`
: Binding time of the both the function body and the return value: blue means that both are static, red means that both are dynamic, pink (as above) means that the body is dynamic while the return value is static. If the function has a void return type, the binding time color is that of the body (thus it cannot be pink).

Number next to function name: `/*7*/`
: Polyvariance index, in order to find the corresponding definition (see Function Definition below). Note that the index numbers for a particular function vary from analysis to analysis. In the HTML colored files, the index number at each call site is a hyperlink to the corresponding definition.

Actuals: `statexp, dynexp, (sdexp)`
: Binding time of each actual argument. The possible extra parenthesis level means that the binding time of the actual argument (given by the color of the expression inside the parentheses) is different from the binding time of the formal argument (given by the color of the parentheses) in the analyzed function; this may happen only after ETA2 as the analysis may turn a static expression into a static&dynamic or a totally dynamic expression.

2.5.7 Function Definition

Let us consider the following colored binding-time annotations computed for a function definition.

```
// BT of return, index, BT of body, BT of formals
extern int foo/*3*/(int stat, int dyn)
  {                  // BT of body: dynamic
    gdyn = dyn;      // BT of stmt: dynamic
    return stat;     // BT of return: static
  }                  // BT of body: dynamic
```

The corresponding binding-time information can be read as follows.

Type and function name: `int foo`
: The color of the function name and type (here blue, *i.e.* static) indicates the binding time of the return value. If the function has a void return type, the function name and type are displayed in black.

Number next to function name: `/*3*/`
: Polyvariance index. Because of polyvariance, there can be several definitions of the same function corresponding to different call site (see Function Definition above), each with different binding-time properties. While all variants have the same name, each has a unique index number. Note that the index numbers for a particular function vary from analysis to analysis.

Formals' parentheses: `(...)`
    Binding time of the body of the function: red, *i.e.* dynamic, in the above example.

Formals: `int stat, int dyn`
    Binding time of each formal argument.

Body's braces: `{...}`
    Binding time of the body of the function: red, *i.e.* dynamic, in the above example.

Return statement: `return` ...
    Binding time of the return value: blue, *i.e.* static, in the above example. If the function has a void return type, the binding-time color (static or dynamic) is irrelevant.

2.5.8 Function Signature

A signature is generated for each function when the variables `verbose_headers` is true (the default). In this case, the signature is shown in a comment at the beginning of each function. It provides information about the non-locals of a function. It looks like this.

```
extern int foo/*42*/(int u, int v)
/*
binding times of read non_locals: a, b, tab, str.x
binding times of written non_locals: a, str.y
residual non-locals top: a, str.x
eval non-locals top: str.x
residual non-locals bottom: b
 */
  {
    ...
  }
```

Names listed are location names: scalar variables, arrays (without brackets), fields of structures and unions (with the name of the type, not of an instance).

This signature contains up to six parts. Only the non-empty parts are shown. The following signature components may appear in all color files.

`binding times of read non_locals:`
    This list gives the binding times of non-local variables and structure fields that are read in either this function or in any function called by this function.

`binding times of written non_locals:`
    This list gives the binding times of non-local variables and structure fields that are written by either this function or by any function called by this function.

The ETA2 propagates information backwards from the point where variables are used to the points where they are defined. (There are possibly several of them.) In the ".eta2.color" file (as well as colored file of subsequent phases), a function signature contains in addition information about non-local variables that are evaluated or residualized within the function (or some function it calls) and about non-local variables that are evaluated or residualized after some call site of the function:

`residual non-locals top:`
: There are residual occurrences of the non-local variables in this list somewhere in the control flow path after entering this function.

`eval non-locals top:`
: There are evaluated occurrences of the non-local variables in this list somewhere in the control flow path after entering this function.

`residual non-locals bottom:`
: There are residual occurrences of the non-local variables in this list somewhere in the control flow path after leaving this function.

`eval non-locals bottom:`
: There are evaluated occurrences of the non-local variables in this list somewhere in the control flow path after leaving this function.

If a variable is in a "top" list, but not in the corresponding "bottom" list (not to be confused with the black "bottom" binding time), then there is an occurrence of the variable within the function or some function it calls. If a variable is in a "bottom" list, but not in the corresponding "top" list, then the function or some function it calls provides the definition required for the subsequent occurrence of the variable.

2.5.9 Entry-Point Signature

If `verbose_headers` is true (the default), there is also a comment near the beginning of each color file indicating the call signature of the entry point. This signature may contains the following information:

`binding times of formals:`
: This list gives the binding times of the formals.

`binding times of read non_locals:`
: This list gives the binding times of the global variables that are used by the program.

`residual non-locals bottom:`
: This amounts to a list of the live variables after the program is called in the larger application, as specified by the `live_locations` Tempo variable, or residualized via the `set_post_analysis_context()` function.

2.6 Compile-Time Specialization

Once the analysis has been performed successfully, *i.e.* if you are satisfied with the computed binding times, you can move on to building a compile-time specializer. The compile-time specialization is done in two steps:

1. Construction of a Compile-Time Specializer, given a successful analysis (*i.e.* an `.at` file).
2. Invocation of a Compile-Time Specializer, given actual values for static parameters of the entry point, yielding the source of a specialized program.

The first step is generally done once; you do not have to rebuild a new compile-time specializer as long as your program does not changes. The second step can be performed many times, for each set of specialization values.

See also:

- Installation of a Specialized Function

2.6.1 Construction of a Compile-Time Specializer

There is no special parameterization for the construction of a compile-time specializer; just run the `cs` top-level command on your program. (See Building a Compile-Time Specializer.)

You do have to rebuild the specializer each time that you provide new specialization values; just edit the `.sctx.c` file (see below) and re-run the `sp` top-level command.

2.6.2 Invocation of a Compile-Time Specializer

Before specializing a program, using the `sp` top-level command (see Running a Compile-Time Specializer), a value must be provided for each piece of data which has previously been declared as static. Recall that static parameters can be globals as well as arguments of the entry point. Because the initialization of those values can be arbitrarily complex, including library calls, the values are given using a C file that is linked to the compile-specializer for performing actual specialization. This C file is suffixed "`.sctx.c`". It should at least:

- Include the file suffixed `.sctx.h`, that has been generated by the construction of the compile-time specializer phase. The `.sctx.h` file provides the exact prototype of the `set_specialization_context()` function (see just below). It also contains the type definitions and the declarations of the global variables of your program.
- Define a function named `set_specialization_context()` whose arguments are pointers to the actual static arguments of your entry point, in the same order. Unlike the `set_analysis_context()` function, the dynamic arguments of the entry point are not (and must not be) present.

The `set_specialization_context()` function is called to set the static values before specialization so that you can set them by an indirect assignment. Because all declarations are provided through the `.sctx.h` file, static global variables can be set by ordinary assignment. *E.g.*

```
#include "foo.sctx.h"

void set_specialization_context(int *expon)
{
  *expon = 4;
  glob = 3;
}
```

You can use the name that you want for the arguments as it is a normal C function. In practice, it is convenient to reuse the argument names of the entry point, *i.e.* to start from the list of formal arguments in the definition of the entry point function, remove the dynamic ones and "add a star" in front of each remaining formal.

The general rule for not forgetting to provide a static value to a static memory cell is to check all locations listed in the `static_locations` variable, as well as the static entry point arguments listed in the `entry_point` variable. If you forget to initialize a memory cell, its value is undefined, as in any C program.

Since the `.sctx.c` file is a standard C file, you may also call functions, open files, read from the terminal, etc. *E.g.*

```
#include <math.h>
#include <stdio.h>
#include "foo.sctx.h"

void set_specialization_context(int *n, struct bar **x)
{
  printf("Enter number of iterations: ");
  scanf("%d",n);

  phase = 4.0 * atan(1.0);

  *x = (struct bar *) malloc(sizeof(struct bar));
  (*x)->a = *n / 2;
  (*x)->b = &phase;
}
```

As you can see in the above example, there is no need to declare the global variable `phase` nor the strcture type `bar`: they are declared in the `foo.sctx.h` file.

Note that structure `bar` may have more fields than `a` and `b`. During specialization, the structure that are partially static are not split into separate static and dynamic parts. You may thus call a function that initializes all fields of a structure even though only some of them are static. The initialization of dynamic fields is just useless. However, you cannot initialize, say, a dynamic global variable as it is not declared in the `.sctx.h` file.

Common Errors in Setting Specialization Contexts

A common error is to set the local variables of `set_specialization_context()`, not their pointer values (*i.e.* not the static arguments of the entry point), *e.g.*

```
void set_specialization_context(int *expon)
{
  expon = 4;  // Should be *expon
}
```

Another common error in `.sctx.c` files is not to provide/allocate memory cells. This generally leads to segmentation faults or bus erros. Consider for example the following implementation of an inner product.

```
int dot_product(int size, int *u, int *v)
{
  int i, sum = 0;
  for (i = 0; i < n; i++)
    sum += u[i] * v[i];
  return sum;
}
```

Assuming vector u and integer size are static, it is an error to define this:

```
#include "dotprod.sctx.h"

void set_specialization_context(int *size, int **u)
{
  *size = 3;
  (*u)[0] = 1;
  (*u)[1] = 0;
  (*u)[2] = 5;
}
```

The reason is that no memory cell has been allocated for vector u. What the entry point expects is an initialized integer size and a initialized pointer to an initialized vector. What is written above does initialize the size but does not initialize the pointer; it assumes the pointer is initialized and only initializes the vector. When specializing the program, you will most likely get a segmentation fault or a bus error but in any case not the expect behavior. A pointer and a vector may be initialized as follows.

```
#include "dotprod.sctx.h"

int init_u[3] = {1,0,5};

void set_specialization_context(int **u, size *n)
{
  *size = 3;
  *u = init_u;
}
```

You may also use malloc() to allocate the memory. But you should not define a local variable in set_specialization_context() and initialize a static location with a pointer to it because the corresponding cell is lost (it is stack allocated) as soon as function set_specialization_context() returns, *e.g.*

```
#include "dotprod.sctx.h"

void set_specialization_context(int **u, size *n)
{
  int init_u[3];

  *size = 3;
  *u = init_u; // Will put you into trouble
  (*u)[0] = 1;
  (*u)[1] = 0;
  (*u)[2] = 5;
}
```

See also:

- Running a Compile-Time Specializer
- Compilation
  - Variable: ctcg_cflags (flags for the C/C++ compiler)
  - Variable: ctcg_ldflags (flags for the link editor)
  - Variable: ctcg_ldflags (additional libraries)
  - Variable: sh_command (shell setting)
- Specializer

    - Constant: `default_cts_bufsize` (default buffer size for the specializer)
    - Variable: `explicit_cts_bufsize` (buffer size for the specializer)
- Post-processing phase (Reference Manual)
- Pretty-printing & extra code transformations phase (Reference Manual)

2.7 Run-Time Specialization

A compile-time off-line specializer first takes the source code of a program (*i.e.*, a set of functions and an entry point) and a description of the invariants. Analyses partition the program into fragments of code that can be evaluated knowing the invariants, and fragments of code that cannot be evaluated until run time. Then, given specialization values, the specializer yields the source code of the specialized program. This result may then be compiled to obtain a specialized executable code.

Unfortunately, this approach is not well suited to cases where the specialization values are only known at run-time. For those cases, specialization must be performed at run time. Directly reusing the traditional source-to-source program specialization would involve the use of a compiler at run-time. In most cases, because compilation is an expensive operation, this solution yields no speed up, but rather a slow down...

Run-time specialization is an extension of traditional off-line specialization. Our system uses the concept of *generating extension*. A traditional generating extension for a given program is a dedicated specializer that, given specialization values as input, yields the source code of the specialized program. Our generating extensions, called *run-time* specializers, directly produces the executable code of the specializations. A run-time specializer may be called at run time to dynamically yield specialized executable code from run-time values.

A key issue in run-time specialization is amortization, *i.e.* the number of time the specialized code has to be called to recover the cost of its production (because production occurs at run-time). Our approach to lower amortization is to do as much compilation as possible before run-time using code templates, leaving only the assembling of templates for run time. Additionally, we prefer to rapidly produce reasonably fast code rather than aggressively optimize the specialized code: the run-time specializer generates good code (but not optimal), very quickly. In particular, no code optimization is done at run-time.

2.7.1 Construction of a Run-Time Specializer

In Tempo, compile-time and run-time specialization share the same analysis up to the action analysis (although it is possible to parameterize the evaluation-time analysis specifically to get better results from run-time specialization; see `lift_all` and `lift_global` variables). Consequently, the specification of the analysis context is the same in both cases. The only difference lies in the type of programs that can be specialized as it may rely on values that are only known at run-time.

In order to build a run-time specializer, given a successful analysis (*i.e.* an `.at` file), run the `rs` top-level command on your program. (See Building a Run-Time Specializer.) At the moment, only `gcc` (or a slightly modified version of `lcc`) can be used for constructing a run-time specializer; see variable `compiler`.

The result is a compiled run-time specializer in a file suffixed with `.rts.o`. This file is generated in a directory named after the architecture and system of the current machine; see variable `arch_dep_dir`. A `.rts.h` file is also generated in the working directory; it contains the signature of the run-time specialization function.

2.7.2 Invocation of a Run-Time Specializer

In order to install the run-time specializer in your original application, just include the `.rts.h` header file wherever you need to call the run-time specialization function and link the `.rts.o` file with your application.

You get the name of the run-time specialization function by looking in the `.rts.h` header file, that contains its prototype. This name has the form: `rts_`*`entrypointname`*`_`*`suffix`*, *e.g.*

```
/* TEMPO Version 1.193, 04/18/98,
  Copyright (c) IRISA/INRIA-Universite de Rennes */

extern void *rts_pow_1(int);
```

Using the run-time specializer is very simple: just call it with actual values for the static arguments of your entry point. It returns a pointer to the dynamically produced specialized function. *E.g.*,

```
int (*spow)(int);

x1   = pow(base,expon);
spow = rts_pow_1(expon);
x2   = (*spow)(base);
```

Actually, this is note pure ANCI C. This is because the prototype is not "complete". It should be:

```
extern int (*((*rts_power_1)(int)))(int);
```

Thus, to be ANSI compilant, you would not have to write:

```
spow = ((int(*((*)(int)))(int))rts_power_1)(expon);
```

The specialized function whose address is stored in `spow` is generated once, at run-time, and may be called many times to amortize the cost of generating it.

See also:

- Installation of a Specialized Function

2.7.3 Recursive and Multiple Run-Time Specializations

In order to do multiple specializations, it is necessary to allocate new buffer space each time the specializer is called. Recursive functions have multiple specializations for the same function and thus require allocation of buffer space as well. By default, the run-time specializer allocate new buffer space each time the specializer is called and for each function. This also makes certain global variables local, so that generating extensions can be recursive to specialize recursive functions. The `reentrant` top-level variable can be set to false to instruct the specializer to generates code in a static buffer and returns a pointer to this buffer.

Note: The run-time specializer does not have a specialization cache and thus recursive calls with previously specialized values will cause the specializer not to terminate. For example, backward branches in a byte-code interpreter will cause an infinite loop. (See the Limitations to Run-Time Specialization.)

Allocating new space each time a generating extension is called is not always desirable, however. If the entry-point function calls a second function, for example, then two buffers will be allocated but only a pointer to the entry-point function (and thus its buffer) is returned. In this situation, the second buffer can never be deallocated. This behavior can be changed by overriding the allocation functions used by the runtime specializer. The specializer calls the functions pointed to by `rts_alloc_data` and `rts_alloc_code` to allocate data and code buffers respectively. These are global variables and can be used to override the default allocation routines (malloc).

The following is an example of overriding the allocation functions so that all code and data are allocated from different sections of a single buffer, so that the memory can be later released.

```
/* Declare the function pointers for
   run-time specializer allocation */
extern char *(*rts_alloc_data)(unsigned int);
extern char *(*rts_alloc_code)(unsigned int);

/* Define some buffer sizes */
#define BUF_CODE_SIZE 500000   /* Total code buffer size */
#define FUN_CODE_SIZE 500      /* Amount of code buffer
                                  for each function */
#define BUF_DATA_SIZE 50000    /* Total data buffer size */
#define FUN_DATA_SIZE 50       /* Amount of data buffer
                                  for each function */

/* Pointers to the buffers */
char *code_buffer, *data_buffer;

/* The current positions in the buffers */
int code_pos, data_pos;

/* New allocation functions */

char *get_code_mem(unsigned int sz)
{
  char *ret;

  ret = code_buffer + code_pos;
  code_pos += FUN_CODE_SIZE;
  if (code_pos > BUF_CODE_SIZE) {
    puts("Code buffer overflow.");
    exit(1);
  }
  return ret;
}

char *get_data_mem(unsigned int sz)
{
  char *ret;

  ret = data_buffer + data_pos;
  data_pos += FUN_DATA_SIZE;
  if (data_pos > BUF_DATA_SIZE) {
    puts("Data buffer overflow.");
    exit(1);
  }
  return ret;
}

void reset_rts_buffers()
{
  code_pos=0;
  data_pos=0;
}

/* Use run-time specializer */

void main()
{
  /* Override allocation functions */
  rts_alloc_data = get_data_mem;
  rts_alloc_code = get_code_mem;

  /* Allocate buffers */
  code_buffer = (char *) malloc(BUF_CODE_SIZE);
  data_buffer = (char *) malloc(BUF_DATA_SIZE);
  reset_rts_buffers();

  /* Specialize function */
  sp = rts_myfunc_1(5);

  /* Specialize again overwriting previous specialization */
  sp = rts_myfunc_1(10);

  /* Release buffers */
  free(code_buffer);
  free(data_buffer);
}
```

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

3 > Tempo — Reference Manual

3.1 Phases

Specialization with Tempo is separated into many phases. They may be grouped into three stages:

1. Analysis phases,
2. Compile-time specialization phases,
3. Run-time specialization phases.

The analysis phases are compulsory. The output of this stage (the action tree) is the shared input of the two independent specialization processes.

3.1.1 Phases Synopsis

The following tables list the successive phases of Tempo, with required input files and generated output files. Only the suffixes of these files are mentioned (see Files). The list of input and output files is exhaustive, apart from this:

- Each phase reads the `config.sml` file. However, to keep the table readable, we have listed it only for the first (i.e., Suif parsing) phase.

- Phases from binding-time analysis to action analysis generate (by default, but this process can be controlled using variable `output_mode`) an additional file that enables the colored visualization of program annotations. This file is suffixed either with `.color` (MIME text/enriched format) or

`.html`, depending on the configuration variable `viewer` (set to `emacs` by default). It is not listed in the following tables.

Optional input and output files are listed in square brackets [like this], as are optional phase (depending on configuration variables). An optional phase that is not selected, *i.e.* "run", actually behaves as the identify though it does generate output files.

Analysis Phases

The goal of the analysis phases is to compute specialization actions. An important intermediate stage is the binding-time information, which is split into three steps in Tempo: binding-time analysis and evaluation-time analysis (1 and 2).

Input Files	Analysis Phases	Output Files

`c`
`config.sml`
[`actx.c`]
	Suif parsing	`spd`
		[`actx.spd`]

`spd`
[`actx.spd`]
	Suif abstract syntax generation	`st`
		`suif.c`
		[`actx.st`]
		[`actx.suif.c`]

`st` [`actx.st`]	Tempo abstract syntax generation	`as` `decl.h`
`as`	[Early pre-processing]	`preproc1.as`
`preproc1.as`	Goto elimination	`nogoto.as`
`nogoto.as`	Alias analysis	`alias.as`
`alias.as`	Indirect call elimination	`nofp.as`
`nofp.as`	Side-effect analysis	`se.as`
`se.as`	Binding-time analysis	`bta.as`
`bta.as`	Evaluation-time analysis (1)	`eta1.as`
`eta1.as`	Evaluation-time analysis (2)	`eta2.as`
`eta2.as`	Flattening of static returns	`flsret.as`

`flsret.as`	Flattening of Static&Dynamic calls	`flsd.as`
`flsd.as`	[Late pre-processing]	`preproc2.as`
`preproc2.as`	Action analysis	`at`

Synopsis of the Analysis Phases

Compile-Time Specialization Phases

The goal of the compile-time specialization phases is first to build a dedicated specializer (from the original program and the initial binding-time information), and then to let the user generate many specialized source files for different specialization values.

Input Files	Compile-Time Specialization Phases	Output Files

`at`	Compile-time specializer generation	`ev.c`
		`ctspec.C`
		`sctx.h`

`ev.c`
`ctspec.C`
`sctx.h`
`sctx.c`
	Compile-time specializer compilation	`ctspec`

`ctspec`	Compile-time specialization	`rawcts.as`
`rawcts.as`	Post-processing	`postproc.as`

`postproc.as`	Pretty-printing
	& extra code transformations
		`cts.c`
		`cts.h`
		`cts.tempo.c`

Synopsis of the Compile-Time Specialization Phases

Run-Time Specialization Phases

The goal of the run-time specialization phases is to build a dedicated run-time specializer (from the original program and the initial binding-time information). Specialized versions with respect to execution-time values are generated as pointers to dynamically created functions. The invocation of the run-time specializer is left to the user. Architecture dependent files are created under a subdirectory *`ARCH`* named after the current platform. (See also variable `arch_dep_dir`.)

Input Files	Run-Time Specialization Phases	Output Files
at	Run-time specializer generation	gram rts.h temp.c temp.d rtspec.c
		Under *ARCH*/
		rtspec.o temp.i temp.o rts.o
at	`C specializer generation	rtstc.c rtstc.h

Synopsis of the Run-Time Specialization Phases

3.1.2 Description of the Phases

The original program undergoes several transformations before actually being treated by Tempo. Some are just side-effects of the Suif front-end while most others are aimed at translating full ANSI C into a smaller and easier to handle subset of the language. Tempo's post-processing rebuilds some constructs that are transformed at this stage so that the output looks more like the original source.

3.1.3 Parsing

C pre-processing, parsing, and transformation of the original file(s).

The parsing of the c file relies on Suif's `scc`. It includes C pre-processor expansion (using `cpp`) and, optionally, some pre-processing using `porky`. If the `actx.c` file exists, it is parsed as well.

The invocation pattern of the parsing phase is as follows.

```
scc scc_flags -.spd file.c
```

If the configuration variable `porky_flags` is not the empty string, the following additional transformation pass is run:

```
porky porky_flags file.spd file.porky_spd
mv file.porky_spd file.spd
```

A similar pattern is applied to the `actx.c` file, if present.

Suif makes some simplifications of the source program:

1. When there are several occurrences of the same string in a program, Suif creates a static global variable for the string with the value of the string initialized in the variable declaration.

This is common with `printf()` statements, that often have similar format strings. We have no control over that.

2. `switch`es with at most 3 cases are automatically rewritten into a cascade of `if`s. Note that this version of Tempo does not support the `switch` construct. `Switch`es with 4 cases or more have to be manually rewritten into a cascade of `if`s.

3. Suif also turns `while` and `for` loops into `do while` loops. This transformation duplicates the condition expression.

In addition, we use an option of Suif (of `porky` actually, see `porky_flags` variable) to turn static (in the C sense) variable into global variables.

See also:

- Suif environment
- Limitations of the Input Language
- Tempo abstract syntax generation (for other initial source transformations)
- Variable: `scc_flags`
- Variable: `porky_flags`

3.1.4 Suif abstract syntax generation

Transformation of Suif `.spd` files (*i.e.* `file.spd` and, possibly, `file.actx.spd`) into a Suif abstract syntax readable by Tempo, *i.e.* `.st` files.

It is for internal usage.

3.1.5 Tempo abstract syntax generation

Building of a Tempo abstract syntax tree by merging the original program (in `st` form) and, if present, the `actx.c` file (in `st` form as well).

Tempo makes some simplifications of the source program:

1. Function variables that are declared in nested blocks are lifted into the main block of the function.

2. Initializations of global variables (including arrays, structures and unions) are split into a separate declarations and a/some assignment(s). This introduces a new function named `init_compound_data()` that contains all the assignements. A call to this function is inserted as the first instruction in the body of the entry point. *E.g.*,

```
int x = 3;
int u[3] = {1,0,5};
char hip[] = "hop";
```

yields

```
int x;
int u[3];
```

```
char hip[4];

void _init_compound_data_1()
{
  x = 3;
  u[0] = 1;
  u[1] = 0;
  u[2] = 5;
  strcpy(hip,"hop");
}
```

There is a special case for character arrays: By default, initialization is performed using `strcpy()` rather than with tons of single assignments (see the `explode_compound_data_strings` variable).

Note that initialization of compound data (*i.e.* arrays, structures and unions) for local variables seems to be broken at the Suif level.

See also:

- Parsing (for other initial source transformations)
- Limitations due to the transformation of initializations
- Variable: `explode_compound_data_strings` (string initialization control)

3.1.6 Early pre-processing

Inlining of functions with multiple returns.

This phase is used to get better results for run-time specialization until run-time inlining is fully implemented. There are actually two preprocessing inlining phases. Inlining functions with multiple returns introduces gotos, which must be eliminated during goto elimination phase. Thus such functions must be inlined before goto elimination. It is not desirable, however, to perform a complete inlining at this point, because this transformation can greatly increase analysis time and make the code less readable. Thus functions with a single `return` are inlined as late as possible, i.e., in the late pre-processing phase just before the action analysis.

See also:

- Variable: `pre_inlining_time`

3.1.7 Goto elimination

All gotos are eliminated by introducing `if`, `while`, and `break` constructs, using temporary variables. The transformation also removes all labels, whether it is the target of a `goto` or not. This enables Tempo to work internally on a smaller subset of C. It greatly simplifies the subsequent analysis phases.

3.1.8 Alias analysis

Computation of alias annotations, i.e., the set of possible target locations for each pointer dereferencing.

A location is either:

- a scalar variable (local or global),

- an array (one single location for all array cells),
- a structure field (one single location for all instances of this structure type)

Note that there is no implicit location for heap allocated objects. This analysis is:

- flow-sensitive: alias information depends on program point,
- *not* context-sensitive: different calls with different alias information are merged.
- monovariant with respect to structures (being able to point to a single structure instance means being able to point to all the instances), unless polyvariance flags are set.

For the analysis to be correct, Tempo has to know all the possible aliases of locations if they have an impact on the program semantics. Alias information computed outside the program fragment to be specialized can be specified using the `file.actx.c` analysis context file.

See also:

- Limitations on aliases and casts
- The Analyses and Their Precision
- Visualization of analysis information
- Variable: `poly_structs`
- Variable: `struct_version`

3.1.9 Indirect call elimination (a.k.a. function pointer elimination)

Transformation of all indirect calls into standard calls. For example:

```
(*fexp)(exp1,exp2)
```

is rewritten into something like

```
_apply_37(fexp,exp1,exp2)
```

The number that follows `_apply_` is some (unique) integer. The corresponding function definition is generated automatically; it depends on the possible values for `*fexp`, as computed by the alias analysis. This function has the following form:

```
type _apply_37(_q,_a1,_a2)
{
  if (_q == func1)
    return func1(_a1,_a2);
  else if (_q == func2)
    return func2(_a1,_a2);
  else
    return func3(_a1,_a2);
}
```

supposing the possible targets are `func1`, `func2` and `func3`.

If the `_apply_37` function is totally static or totally dynamic, it is turned back into an indirect call during action analysis; indeed there is not any specialization to perform. Otherwise, if some specialization of functions `func1`, `func2` and `func3` is possible, the explicit dispatch stays in the residual program: if there is not too many cases, calling a specialized function should compensate for the ifs.

If the list of possible targets includes some totally unknown function pointer (e.g., provided in a global variable or as an actual argument to the entry point), Tempo may generate something in the form of

```
type _apply_37(_q,_a1,_a2)
{
  if (_q == func1)
    return func1(_a1,_a2);
  else if (_q == func2)
    return func2(_a1,_a2);
  else if (_q == func3)
    return func3(_a1,_a2);
  else
    return (*_q)(_a1,_a2);
}
```

The indirect call `(*_q)(_a1,_a2)` at the bottom of the function assumes that it cannot jump to any of the functions in the program; it is treated as an external call, *i.e.* evaluated or rebuilt (depending on variable `residualize_all_icalls`).

See also:

- Variable: `residualize_all_icalls` (binding time of external indirect calls)

3.1.10 Side-effect analysis

Recording in each function signature the set of non-locals variables that are read or written.

See also:

- Visualization of analysis information
- Variable: `poly_structs`
- Variable: `struct_version`

3.1.11 Binding-time analysis (a.k.a. BTA)

Annotation of the program with binding-time information.

This analysis is:

- flow-sensitive: binding-time information depends on program point,
- context-sensitive: different calls with different binding-time contexts yield different function analyses,
- return-sensitive: there is a separate binding time for the body and the return value of a function

Note that because of the polyvariance (context sensitivity), some functions may appear duplicated due different binding times. Also, notice that sensitivity is limited to scalar values.

The BTA alone does not yield correct binding-time annotations. Evaluation-time analysis is needed, providing the ``use-sensitivity" feature. When visualizing BTA files, note also that static left-hand side of assignments only means that the address of the location is static, not that the content is static.

See also:

- Visualization of analysis information
- Variable: `poly_structs`
- Variable: `struct_version`

3.1.12 Evaluation-time analysis (1) (a.k.a. ETA 1)

This analysis turns into dynamic all static expressions that appear in a dynamic context and whose corresponding values have no textual representation in C (i.e., pointers, structures and arrays). If such an expressions were to remain static, it would not be possible to provide a textual representation for it in the compile-time specialized program. This constraint might be partly removed in the case of run-time specialization since no textual representation is required (see `lift_all`, and `lift_global` variables).

See also:

- Variable `lift_all`
- Variable `lift_global`
- Variable `minimize_holes`

3.1.13 Evaluation-time analysis (2) (a.k.a. ETA 2)

This analysis implements *return sensitivity* and completes the binding-time analysis. It turns into dynamic all the static definitions of variables with dynamic uses. If there is no static use of a definition, it is turned totally dynamic as the static facet is not needed. The analysis is also monovariant with respect to structures (unless polyvariance flags are set).

See also:

- Visualization of analysis information
- Binding-time analysis phase
- Variable: `poly_structs`
- Variable: `struct_version`

3.1.14 Flattening of static returns (a.k.a. Flattening 1)

This transformation rewrites functions (both at the call site and the corresponding definition) that contain a dynamic body but static returns.

The transformation consists of two parts:

- In the function definition:
    1. Replace the `return` *`exp`* statements with an assignment statement to a fresh, new global variable (a.k.a. the return variable).
    2. Turn the return type of the function to `void`.
- At the call site:
    1. Insert a statement before the call point that calls the new void function.
    2. Replace the call by the return variable.

3.1.15 Flattening of static & dynamic calls (a.k.a. Flattening 2)

This transformation flattens nested function calls with S&D parameters. There is no fundamental need for it; it just simplifies the action analysis that follows. After the transformation, function calls with S&D parameters can only be in one of these two forms:

- `func(`*... S&D parameters ...*`);`
- *`lexp`* `= func(`*... S&D parameters ...*`);`

3.1.16 Late pre-processing

Inlining of functions with a single `return`.

It is used to get better results for run-time specialization until run-time inlining is fully implemented. The reason why there are two pre-processing inlining phases (the other is the early pre-processing) is that functions with multiple returns introduce gotos, which must be eliminated (see goto elimination). Functions with single `return` are inlined as late as possible (i.e., just before the action analysis) because pre-inlining can otherwise greatly increase analysis time.

3.1.17 Action analysis

Translation of binding-time information into specialization actions (i.e., atomic transformations). Action analysis also turns explicit `switch`es over functions pointers back into indirect function pointer calls when they are fully static or dynamic (see indirect call elimination).

3.1.18 Compile-time specializer generation

Separation of static code fragments (to be evaluated) from the actions (driving of specialization and code reconstruction) to be performed by the compile-time specializer.

3.1.19 Compile-time specializer compilation

Compilation and linking of all the files needed for compile-time specialization:

- an action tree to be interpreted (file `ctspec.o`),
- the generic action interpreter (library file `libctcg.a`),
- static fragments to be computed (file `ev.o`),
- a function setting the initial specialization context (file `sctx.o`),
- possibly, user libraries (see variable `ctcg_ldlibs`).

See also:

- Variable `ctcg_cflags`
- Variable `ctcg_ldflags`
- Variable `ctcg_ldlibs`

3.1.20 Compile-time specialization

Execution of the compile-time specializer. By default, the compile-time specializer expects non-recursive programs.

See also:

- Variable `explicit_cts_bufsize`
- Compile-time specialization (User manual)

3.1.21 Post-processing

Some transformations on the specialized program, not related to specialization.

- Optimizations:
  - Inlining,
  - Algebraic simplifications.
- Clean-up:

    - Dead-code elimination,
    - Removal of empty blocks and uselessly nested blocks
- Re-sugaring.

See also:

- Variable `post_clean_up` (to enable clean-up)
- Variable `post_common_subexpression` (to enable unfolding of expressions in consecutive assignments)
- Variable `post_do_inline` (to force inlining of specific functions)
- Variable `post_do_not_inline` (to prevent inlining of specific functions)
- Variable `post_inlining` (to enable inlining)
- Variable `post_inlining_max_nb_calls` (threshold for the number of calls to inline)
- Variable `post_inlining_max_nb_stmts` (threshold for the body size to inline)
- Variable `post_inlining_mode` (style of inlining)
- Variable `post_inlining_renaming` (to rename the parameters)
- Variable `post_s2c_flags` (flags passed to `s2c`)

3.1.22 Pretty-printing - Additional `porky` post-processing

Generation of the resulting compile-time specialized program in C text format.

An optional, additional post-processing transformation using `porky` may be performed. For example, porky can be used to perform some copy propagation on the specialized program. The invocation pattern of `porky` is as follows.

```
scc -.spd file.cts.c file.cts.spd
porky post_porky_flags file.cts.spd file.cts.porky_spd
cp file.cts.c file.tempo.c
s2c post_s2c_flags file.cts.porky_spd file.cts.c
```

As may be seen above, the original Tempo specialization is kept in file `file.tempo.c`.

See also:

- `porky` (assorted code transformations)

3.1.23 Run-time specializer generation

Generation of the run-time specializer. The generated `rts.o` file is to be linked to the original application in order to generate specialized programs at execution time. The name of the function to call to do so (i.e., the dedicated specializer) is in the `rts.h` header file.
See also:

- Run-time specialization (User manual)

3.1.24 `C specializer generation

Generation of a run-time specializer as `C (i.e., tick C) input file.

Note that this functionality is unstable.

3.2 Files

Tempo operates on files that are all in the same directory (or in an architecture-dependent sub-directory). Furthermore, they all share the same file name prefix. For example, if the program to specialize is `/home/jake/spec/power.c`, all files are read and generated in the directory `/home/jake/spec/`. Starting from the file `power.c`, Tempo also reads `power.config.sml` and generates, e.g., `power.at` and `power.cts.c`.

3.2.1 Tempo File Suffixes

Below is the list of all Tempo file suffixes and their description.

`actx.c`

Abstract functions for modeling:

- the context of the program before the entry point is called,
- external functions called in the program,
- the context of the program after the entry point is called.

It is a legal C file, optionally provided by the user.
See also:

- Specifying the analysis context
- Suif parsing

`actx.spd`

Suif internal format for the `actx.c` file abstract syntax.
This is an intermediate file, hidden by default to the user. It is not generally useful unless the user is wondering what exactly Suif has done to his file.
See also:

- Specifying the analysis context
- Suif parsing (producer)
- Suif abstract syntax generation (consumer)

`actx.st`

Textual tree representation for the Suif abstract syntax of the `actx.c` file.
It is an intermediate file, hidden by default to the user. It is not generally useful to the user.
See also:

- Suif abstract syntax generation (producer)
- Tempo abstract syntax generation (consumer)

`actx.suif.c`

C text format of the Suif representation for the `actx.c` file.
It is an intermediate file. It enables one to check whether Suif performed any (generally minor) transformation to the original program.
See also:

- Suif abstract syntax generation (producer)

`alias.as`

Program annotated with alias information, i.e., set of possible target locations for each pointer dereference.
It is an intermediate file, hidden by default to the user.

See also:
- Alias analysis (producer)
- Indirect call elimination (consumer)

as

Tempo abstract syntax for the merging of the original program and, possibly, the `actx.c` file.
It is an intermediate file, hidden by default to the user.
See also:
- Tempo abstract syntax generation (producer)
- Early pre-processing (consumer)

at
at.color
at.html

Program annotated with actions.
This generated file is the final result of all the analysis phases. The `color` (or `html`) file shows a program annotated with actions represented as colors. If the user is not satisfied with the degree of specialization, or would like to better understand the stages of the analysis that led to those action annotations, the color files associated with the `bta` or `eta2` can be helpful.
See also:
- Action analysis (producer)
- Compile-time specializer generation (consumer)
- Run-time specializer generation (consumer)

bta.as
bta.color
bta.html

Program annotated with binding-time values.
Note that some functions may appear duplicated due to polyvariance.
See also:
- Binding-time analysis (producer)
- Evaluation-time analysis (1) (consumer)

c

Original C source of the program to specialize.
This file must be provided by the user.
See also:
- Limitations
- Suif parsing (consumer)

config.sml

Configuration and information for analysis and specialization.
This file must be provided by the user. It is a legal SML file. It is read before running any Tempo phase. It is used to define program-specific variables. In particular, the definition of the SML variable `entry_point` is required. Here is an example:

```
entry_point := "foo(S,D)";
static_locations := ["u","v","str.a"];
external_functions := EVALUATE(["bar"]);
post_inlining := true ;
post_inlining_mode := FLAT;
```

See also:

- Tempo variables

`cts.c`
`cts.h`
`cts.tempo.c`

Compile-time specialized program.
The `cts.c` file is the final result of compile-time specialization. The `cts.h` file is used to reduce the size of the `cts.c` file, in case there are many declarations.
This file is overwritten if, after compile-time specialization, an additional post-processing phase is requested. In that case, the result of the original specialization is saved with the name `cts.tempo.c`.
See also:

- Threshold for declaration ellipsis (variable `max_decls_size`)
- Additional `porky` post-processing (producer)
- Specialized program pretty-printing (producer)

`ctspec`

An executable form of a compile-time specializer; it consists of

- an action tree to be interpreted (file `ctspec.o`),
- the generic action interpreter (library file `libctcg.a`),
- static fragments to be computed (file `ev.o`),
- a function setting the initial specialization context (file `sctx.o`),
- possibly, user libraries (see variable `ctcg_ldlibs`).

See also:

- Compile-time specializer compilation (producer)
- Compile-time specialization (consumer)

`ctspec.C`

Actions to be performed by the compile-time specializer.
This is a C++ file (because the compile-time specializer, i.e., the action interpreter, is written in C++). It contains all the dynamic fragments (in abstract syntax form) to rebuild; names of functions to call to evaluate static expressions (found in file `ev.c`); and, the actions to perform.
See also:

- Compile-time specializer generation (producer)
- Compile-time specializer compilation (consumer)

`decl.h`

Type and global variable declarations found in the program.
Sometimes programs have numerous of declarations because many header files are included. Parsing inlines all the `include` directives. In order not to obscure the visualization of analyzed files, when the global declarations are to numerous, they are all removed and replaced by the `include` directive referring to `decl.h`.
See also:

- Threshold for declaration ellipsis (variable `max_decls_size`)

`ds.c`

Data specializer.
This experimental feature is not fully operational. It has been left undocumented. You may peek into the generated code to get some inspiration.

eta1.as
eta1.color
eta1.html

Program where some static expressions have been turned into dynamic expressions because they occur in a dynamic context and do not correspond to representable values (i.e., pointers, structures and arrays).
See also:

- Evaluation-time analysis (1) (producer)
- Evaluation-time analysis (2) (consumer)

eta2.as
eta2.color
eta2.html

Program where static definitions of variables with dynamic uses have been turned static and dynamic, or dynamic only if those variables have no static use.
Note that some functions may get duplicated due to polyvariance.
See also:

- Evaluation-time analysis (2) (producer)
- Flattening of S&D calls (consumer)

ev.c

Code fragments to be evaluated during compile-time specialization.
See also:

- Compile-time specializer generation (producer)
- Compile-time specializer compilation (consumer)

flsret.as
flsret.color
flsret.html

Program where functions with a dynamic body and static returns have been rewritten as void functions performing a side effect on a fresh ``return variable".
The corresponding transformation makes explicit the return sensitivity. It simplifies the specialization actions.
See also:

- Flattening of static returns (producer)
- Flattening of static & dynamic calls (consumer)

flsd.as
flsd.color
flsd.html

Program with flat static & dynamic calls.
The corresponding transformation just simplifies the specialization actions.
See also:

- Flattening of static & dynamic calls (producer)
- Late pre-processing (consumer)

gram

Grammar representing all possible specializations of the program.
This file is generated for information only. See also:

- Run-time specializer generation (producer)

nofp.as

Program with function pointers turned into explicit calls.
The corresponding transformation enables Tempo to work on a smaller subset of C. The `switch`es that make calls explicit are turned back into indirect function pointer calls during the action analysis.
See also:

- Action analysis (back transformation)
- Indirect call elimination (producer)
- Side-effect analysis (consumer)

nogoto.as

Program with gotos turned into conditionals and loops.
The corresponding transformation enables Tempo to work on a smaller and structured subset of C.
See also:

- Goto elimination (producer)
- Alias analysis (consumer)

postproc.as

Post-processed specialized program.
The corresponding transformation does some code clean-up, some simple optimizations not related to specialization, and some re-sugaring.
See also:

- Post-processing (producer)
- Specialized program pretty-printing (consumer)

preproc1.as
preproc1.c

Program with early pre-processing.
The corresponding transformation performs inlining of functions with multiple returns. It is used to get better results for run-time specialization, until run-time inlining is fully implemented.
See also:

- Early pre-processing (producer)
- Goto elimination (consumer)

preproc2.as
preproc2.color
preproc2.html

Program with late pre-processing.
The corresponding transformation performs inlining of functions with single returns. It is used to get better results for run-time specialization, until run-time inlining is fully implemented.
See also:

- Late pre-processing (producer)
- Action analysis (consumer)

rawcts.as
rawcts.c

Raw output of the compile-time specializer.
By default, only the `rawcts.as` file is generated. Use the variable `output_mode` or command `as2c` to visualize the C text version of this file.

See also:

- Compile-time specialization (producer)
- Post-processing (consumer)

`rts.h`
`rts.o`

Run-time specializer.
The generated `rts.o` file is the final result of the run-time specialization phase. The user may link it with his original application in order to generate specialized programs at execution time. The name of the function to call to do so (i.e., the dedicated specializer) is in the `rts.h` header file.
See also:

- Run-time specializer generation (producer)
- [[[run-time specialization]]] (User manual)

`rtspec.c`
`rtspec.o`

Generating extension (i.e., template assembler and hole filler).
The function defined in this file assembles pre-compiled templates (i.e., the dynamic slice of the code) according to the specialization actions. It also plugs into the templates the result of evaluating the static expressions.
See also:

- Run-time specializer generation (producer)

`rtstc.c`
`rtstc.h`

Run-time specializer as a `C input file.
Note that this functionality is unstable
See also:

- `C specializer generation (producer)

`sctx.c`

Compile-time specialization context.
This is a legal C file that must be provided by the user. It must contain the definition of a function named `set_specialization_context()`. This function initializes globals and entry-point parameters declared as static at the analysis phase.
See also:

- Compile-time specializer compilation (consumer)
- Compile-time specialization (caller)

`sctx.h`

Header file for the compile-time specialization context.
This file is automatically generated. Do not attempt to edit it. It makes the connection between the store (file `ev.c`) and the setting of the specialization context (file `sctx.c`).
See also:

- Compile-time specializer generation (producer)
- Compile-time specializer compilation (consumer)

`se.as`

Program with side-effect annotations.
The corresponding transformation records in each function signature the set of non-local variables that

are read or written.
See also:

- Side-effect analysis (producer)
- Binding-time analysis (consumer)

`spd`

Suif internal format for the `c` file abstract syntax.
This is an intermediate file, hidden by default to the user. It is not generally useful unless the user is wondering what exactly Suif has done to his file.
See also:

- Suif parsing (producer)
- Suif abstract syntax generation (consumer)

`st`

Textual tree representation for the Suif abstract syntax of the `c` file.
It is an intermediate file, hidden by default. It is not generally useful to the user.
See also:

- Suif abstract syntax generation (producer)
- Tempo abstract syntax generation (consumer)

`suif.c`

C text format of the Suif representation for the `c` file.
It is an intermediate file. The user may check whether Suif performed any (generally minor) transformation to the original program.
See also:

- Suif abstract syntax generation (producer)

`temp.c`
`temp.o`

Code templates for run-time specialization.
This file contains the templates that represent the dynamic operations of the code to specialize at run time.
See also:

- Run-time specializer generation (producer)

`temp.d`

Symbolic description of the template holes used for run-time specialization.
This file lists each template name of the program to specialize and the symbolic address of its holes. This intermediate file is hidden by default to the user. It is the input of the `tcc` template compiler, together with the `temp.o` file. The output of `tcc` is the `temp.i` file.
See also:

- Run-time specializer generation (producer)

`temp.i`

Physical description of the template holes used for run-time specialization.
This file lists each template name of the program to specialize and the relative address of its holes. This intermediate file is hidden by default to the user. It is the output of `tcc`, that makes explicit the symbolic description of code templates found in the `temp.d` file.
See also:

- Run-time specializer generation (producer)

3.2.2 Tempo File types

Here is the list of all file types manipulated by Tempo.

`.as`
: Concise form of the abstract syntax used by Tempo to represent programs. It includes alias, side-effect, and binding-time information.

`.at`
: Concise form of the abstract syntax used by Tempo to represent action-annotated programs.

`.C`
: C++ file.

`.c`
: C file.

`.color`
: MIME text/enriched format of annotations on a program. It can be viewed using emacs' enriched mode. (See the variable `viewer`.)

`.d`
: Symbolic description of code templates for run-time specialization.

`.gram`
: Grammar representing all possible specializations of a program.

`.H`
: C++ header file.

`.h`
: C header file.

`.html`
: HTML representation of the annotations of a program. (See variable `viewer`.)

`.i`
: Physical description of code templates for run-time specialization.

`.o`
: Object file obtained by compilation of a C or C++ file.

`.sml`
: SML file. (Most parts of Tempo are implemented in SML but the only SML files that the user should ever see are the `config.sml` and the `.tempo.sml` files).

`.spd`
: Suif internal abstract syntax representation of a program.

`.st`
: Tempo representation of the Suif abstract syntax for a program.

3.2.3 Other Files

Here are other files (not suffixes) relevant to Tempo.

`.tempo.sml`
: If the user has a `.tempo.sml` file in his home directory (that is the full name, not just an file extension), it is loaded when `tempo` (the Shell command, not the SML function) is run. This enables the user to customize standard settings. The file is loaded only once.
: See also:
  - Tempo SML top-level variables

- File: `file.config.sml` (per-program configuration file)

3.3 Commands

In the Tempo environment, there are two kinds of commands:

- Shell-level commands (to run the Tempo system or a Suif tool)
- Tempo SML top-level commands (e.g., to run a specialization phase)

The Tempo SML top-level commands are the most commonly used in practice. We consider each kind in turn.

3.3.1 Shell-Level Commands

All the shell-level commands that follow are available in the Tempo distribution for any of supported platforms. In practice, the user only needs to run the `tempo` command; all other commands are implicitly run from the Tempo top-level.

`tempo`

Run the Tempo Top Level. This is interactive SML top level where all of the Tempo functionalities have been pre-loaded. When it is started, the user should see something that looks like this (the output of the system is in bold face):

```
mingus% tempo
Running Tempo on SunOS-5
TEMPO Version 1.191, 03/24/98, Copyright (c) IRISA/INRIA-Universite de
Rennes
val it = () : unit
-
```

The architecture name is that of the current machine. The version number and creation date depend on when Tempo was built (not when it was installed). The ``**-**" sign is the prompt character.

See also:

- Tempo top-level commands
- SML environment

`porky`
`snoot`
`s2c`
`s2st`
`scc`

See the Suif manual page

3.3.2 Tempo Top-Level Commands

Here are the Tempo commands that the user can type at the Tempo SML top-level.

See also:

- Shell command `tempo` (for running the system)
- SML environment (for SML commands)

`an` *`file`*

Running analyses. Starting from *`file`*`.c`, run all the analysis phases and generate a *`file`*`.at` file. The string argument *`file`* must be a name without any path or extension, e.g., `"power"`.
Example:
```
an "power";
```
Note that this is actually an abbreviation for: `tempo` *`file`* `"c" "at"`.

`as2c` *`file.as`*

Generation of C text from abstract syntax format. This command reads *`file.as`*, which must be a valid Tempo abstract syntax file (see the suffix ``` ``.as" ```), and writes it in C text format into *`file`*`.c`. The string argument must be a name with the ``` ``.as" ``` extension, e.g., `"power.bta.as"`.
Example:
```
as2c "power.rawcts.as"; (* generate power.rawcts.c *)
```

`cd` *`directory`*

Changing working directory. Tempo operates on files that are all in the same directory (or in an architecture-dependent sub-directory). This command changes the working directory of Tempo. It also changes the current working directory (in the shell sense) of the SML top-level. (This could have an impact in case of explicit I/O.) The string argument *`directory`* may be absolute or relative. The working directory may also be changed using the SML variable `wd`.
Example:
```
cd "/home/jake/spec/power/";
cd "../rpc";
```

`cs` *`file`*

Generation of compile-time specializer. Tempo operates on files that are all in the same directory (or in an architecture-dependent sub-directory). Starting from the action file *`file`*`.at`, this command generates the compile-time specializer *`file`*`.ctspec.C`. The string argument *`file`* must be a name without any path or extension, e.g., `"power"`.
Example:
```
cs "power";
```
Note that this is actually an abbreviation for: `tempo` *`file`* `"at" "ctspec"`.

`ds` *`file`*

Generation of a data specializer. Data specialization mostly is implemented in Tempo. But it is still experimental and has been left documented. Starting from the action file *file*`.at`, this command generates the data specializer *`file`*`.ds.c`. The string argument *file* must be a name without any path or extension, e.g., `"power"`.
Example:
```
ds "power";
```
Note that this is actually an abbreviation for: `tempo` *file* `"at" "ds"`.

`freeze_points` *`file`*

Listing of static locations turned dynamic. This command analyses *`file`*`.bta.as` for variables/fields declared to be static in *`file`*`.config.sml` which are ``frozen'' (i.e., which become dynamic) and print the function in which they become dynamic. This can be some help in ``debugging'' BTA results in large files.
Warning: static assignments under dynamic control cause the variable/field to become dynamic in the function containing the dynamic control, *not* the function containing the assignment.
Example:

```
freeze_points "power";
```

See also:

- Command `tempo` (to generate *`file`*`.bta.as`)
- Variable `output_mode` (to dump *`file`*`.bta.as`)

`print_config` *`file`*

Printing of configuration variables. Command `print_config` returns (and displays) the values of Tempo configuration variables. If the string argument *`file`* is the empty string, then the current state is returned. If *`file`* is not empty, it must be a name without any path or extension, e.g., `"power"`; it then returns the configuration that would be used when loading the *`file`*`.config.sml` file.
Example:

```
print_config "";
print_config "power";
```

`rs` *`file`*

Generation of a run-time specializer. Starting from the action file *`file`*`.at`, command `rs` generates the run-time specializer *`file`*`.rts.o`. The string argument *`file`* must be a name without any path or extension, e.g., `"power"`.
Example:

```
rs "power";
```

Note that this is actually an abbreviation for: `tempo` *`file`* `"at" "rts.o"`.

`rts_compact_inline` *`()`*
`rts_inlining` *`()`*
`rts_optimized_inlining_one_pass` *`()`*
`rts_optimized_inlining_two_passes` *`()`*

Instruction for additional inlining in the runtime specializer. These commands are experimental and undocumented.
See also:

- Early pre-processing (Reference Manual)
- Late pre-processing (Reference Manual)
- Variable `rts_do_inline`
- Variable `rts_do_not_inline`

`sp` *`file`*

Performing specialization. Starting from the specializer *`file`*`.ctspec.C`, command `sp` generates

the compile-time specialized program `file.cts.c`. The string argument `file` must be a name without any path or extension, e.g., `"power"`.
Example:

```
sp "power";
```

Note that this is actually an abbreviation for: `tempo file "ctspec.C" "cts"`.

`tempo` *`file start_extension end_extension`*

Running phases. Starting from `file.start_extension`, command `tempo` runs all the required phases to produce `file.end_extension`. The string argument `file` must be a name without any path or extension, e.g., `"power"`. The *start_extension* and *end_extension* strings may qualify an actual file without mentioning a specific type extension, e.g., `"bta"` (same as `"bta.as"`), `"at"`.
Example:

```
tempo "power" "bta" "at";
tempo "power" "at" "cts";
```

3.4 Variables

Tempo can be configured with two kinds of user variables:

- Shell-level variables (to define paths)
- Tempo SML top-level variables (to parameterize specialization)

As is the case for commands, the SML variables are the most commonly used in practice. We consider each kind in turn.

3.4.1 Shell-Level Variables

In the following description of variables, we denote by `$TEMPOHOME` the installation directory of Tempo, even though such a Shell variable is not visible to the user. The user does not have to know this directory (apart from the need to run the `tempo` command) as the other paths are relative to this one.

`TEMPOWORK` (default: `""`)

Starting working directory. This Shell variable controls the initializing of the Tempo working directory. If the string is empty (the default), the starting working directory is the directory where the `tempo` Shell command is invoked. If the string is not empty, the working directory is initialized to the directory specified by the string.

See also:

- Command: `cd`
- Variable: `wd`

`TEMPOSUIFHOME` (default: `""`)

Tempo-Suif main directory. This variable can be used to override the default setting of Suif variable `SUIFHOME` (performed by the tempo shell-level command) in order to run Suif commands other than those provided with Tempo. (You will most likely never need to do this.) Be careful though that the `s2st` command will not be found outside of the distribution of Tempo.

See also:

- Suif Variable: `SUIFHOME` (Suif main directory)
- Suif Command: `s2st` (Suif abstract syntax printer)

See also:

- Suif Variable: `MACHINE` (target architecture)
- Suif Variable: `SUIFHOME` (Suif main directory)
- Suif Variable: `SUIFPATH` (Suif path to binaries)

3.4.2 Tempo SML Top-Level Variables

The following SML variables may be set at the SML top-level or in the SML files. A few of them are still undocumented. Some names may seem strange; this is for historical reasons...

See also:

- Command: `print_config`
- Using SML Variables
- Using SML Files
- SML environment

`arch_dep_dir` (default: "*current platform architecture*")

Directory for RT architecture dependent files. Architecture-dependent files generated during run-time specialization phases are stored in a sub-directory of the current working directory that is named after this variable. The default value is set when the session is open, depending on the platform architecture on which Tempo is running. Possible default values are the same as for variable `architecture`.

`architecture` (default: "*current platform architecture*")

Current architecture. This variables stores, for internal purposes, the name of the platform architecture on which Tempo is running. It should not be altered by the user. Possible values are:

`"SunOS-4"`
Sun OS 4.1
`"SunOS-5"`
Sun OS 5.5, i.e., Solaris 2.5
`"Linux-2"`
Linux on PC

`compiler` (default: `""`)

C Compiler for compile-time and run-time specialization. This variable specifies which compiler to use to compile the compile-time specializer and the template binaries. The default value `""` means that gcc will be used.

See also:

- Run-time specializer generation phase

`csd` (default: "*`$TEMPOHOME`*`/ctcg`")

Compile-time specializer internal directory. This is the path to the compile-time specializer internals (library and header files). In practice, it should not be altered by the user.

`ctcg_cflags` (default: `""`)
`ctcg_ldflags` (default: `""`)
`ctcg_ldlibs` (default: `""`)

Compilation flags for the CT specializer. Those string variables allow additional flags to be passed to the compilation of the compile-time specializer:

- `ctcg_cflags` is given to the compiler when compiling CT specializer files, i.e., *`file`*`.ctspec.C`, *`file`*`.ev.C`, *`file`*`.sctx.c`.

- `ctcg_ldflags` is given to the linker when linking the above files (together with the specializer library `libctcg.a`), e.g., `"-L/home/jake/lib"`.

- `ctcg_ldlibs` is given to the linker to specify additional libraries, e.g., `"foo.o /home/bob/lib/bar.o -lsupermath"`. Note that the object files, if any, must be specified before any libraries.

See also:

- Compile-time specializer generation phase

`cts_flags` (default: `""`)

Flags for the CT specializer. This variables is given on the command line that calls the compile-time specializer. Known flags of the CT specializer are:

`-norec`
: Tells the CT specializer that the program does not contains recursive calls, enabling a faster code generation.

Note that the `-norec` option is unsafe.

`default_cts_bufsize` (default: 5M)

Buffer size for the CT specializer. This integer variable specifies the size in bytes of the buffer used by the compile-time specializer to store the specialized functions. The value of this variable cannot be assigned by the user.

`ds_mode` (default: `false`)

Data specialization mode. Undocumented flag.

`dummy_entry_point_name` (default:`"dummy_entry_point"`)

Name for dummy entry point. This variable provides the name for the dummy entry point, which is created when there exist a `set_specialization_context()` or a `set_post_analysis_context()` function in the `.actx.c` file.

`entry_point` (no default value)

Specialization entry-point. This variable specifies both the function entry point in a program (i.e., the original C file) and the binding times of its arguments. The value is a string that mimics the entry point function call, e.g., `"power(D,S)".` Possible values are *`function`*`(`*`arguments`*`)` where *`arguments`* is a (possibly empty) list of the following items:

`S`
: Static argument.

`D`
: Dynamic argument.

`_`
: Undefined or composite (i.e., non-scalar) argument.

An error is reported if the number of arguments of *`function`* specified by the variable `entry_point` is not the same as the number of arguments of the actual function in the program.

For the time being, Tempo is restricted to a single entry point and binding-time informationdefinition.

Note that there is no default value for this variable. Not only must it be defined explicitly by the user, but also the definition must appear in the `config.sml` file. Setting it at the top-level has no effect. This restriction is to prevent unwanted ``side-effects" when specializing several programs in the same Tempo session.

See also:
- Variable static_locations (binding time of globals and structures)
- Variable external_functions (binding time of external functions)
- Analysis context file `actx.c` (more precise binding times)
- Specifying the analysis context (User manual)

`explicit_cts_bufsize` (default: `NONE`)

Buffer size for the CT specializer. At the moment, the compile-time specializer requires the size of the specialization buffer to be specified. The `explicit_cts_bufsize` enables the user to override the default buffer size. Possible values are:

`NONE`
: The default buffer size value is used. The current default value is given by the SML variable `default_cts_bufsize`. (Note that, unlike other Tempo SML variable, the user cannot modify this variable as its value is not a reference.) The current default size is 5M, i.e., 5000 * 1024 bytes.

`SOME` *`size`*
: The specified integer *`size`* (in bytes) is used.

`explode_compound_data_strings` (default: `false`)

String initialization control. The C subset treated by Tempo does not include initializations. As a result, Tempo rebuilds explicit assignments for all initializations, including compound data. As a string is seen as an array of characters, one explicit assignment is introduced for each array cell, i.e., each character in the string. This variable permits such initializations to be performed in one single statement using C function `strcpy()`. For example, the following initialization:

```
char my_array[3] = "hi";
```

is rewritten as

```
char my_array[3];
my_array[0] = 'h';
my_array[1] = 'i';
my_array[2] = '\0';
```

when `explode_compound_data_strings` is true.

Initializations are in this more compact form

```
char my_array[3];
strcpy(my_array,"hi");
```

when `explode_compound_data_strings` is false. Note that this variable does not affect other kinds of compound data initializations.

`external_functions` (default: `EVALUATE[]`)

Binding time of external functions. This variable specifies whether external functions should be evaluated if possible (i.e., all arguments are static) at specialization time or should always be residualized (regardless of the arguments). In practice, the former case is often used for calling library functions and the latter for preventing I/O side-effects at specialization time (such as printing). Possible values are:

`EVALUATE [`*`functions`*`]`
All external functions whose name appears in the string list *`functions`* can be called at specialization time if their arguments are available (i.e., static). The other external functions are considered dynamic and are always residualized. In particular, `EVALUATE[]` forces the residualization of all external functions.

`RESIDUALIZE [`*`functions`*`]`
All external functions whose name appears in the string list *`functions`* are always residualized. The other external functions can be called at specialization time if their arguments are available (i.e., static). In particular, `RESIDUALIZE[]` enables all external functions of the program to be called at specialization time, when possible.

See also:
- Analysis context file `actx.c` (abstract description of the behavior of external functions)
- Variable `residualize_all_icalls` (binding time of external indirect calls)

- Specifying the analysis context (User manual)

`extra_rts` (default: `false`)

Size of run-time specialized functions. This is an ***UNSTABLE*** feature. When this variable is true, building a run-time specializer generates an extra function. This functions returns the size of a specialized function (it has the same interface as the run-time specializer generated along) instead of a pointer to it. This only used to satisfy curiosity and fill tables in papers.

The funny things are that you have to set `arch_dep_dir` to `"."`, and that the generated files are named `*file*.rts.sh` and `*file*.rts.so`.

`gnumake` (default: machine dependent)

Path for gnu tools. This is an internal variable, set via the `tempo` shell-level command, that need not be altered by the user.

`keep_headers` (default: true)

Controlling whether information about read/written/etc. variables is included in the `at` file. When this variable is true, the information about read/written/etc. variables is kept in the `at` file. When this variable is false, this information is removed. Note that this flag is different than `verbose_headers`, which only affects the color files, although if there is no such information in the `at` file, it won't appear in the `at.color` file either, regardless of the value of `verbose_headers`. Setting this flag to false is useful to cut down the size of the at file, either to save disk space or to make it quicker to read by the `cs` and `rs` commands.

`lift_all` (default: `false`)

Controlling the lifting of pointer values for run-time specialization.

`true`
: The ETA allows all pointer values to be lifted.

`false`
: The ETA disallows pointer values to be lifted.

`lift_global` (default: `false`)

Controlling the lifting of global pointer values.

`true`
: Setting variable `lift_global` to true only causes pointers to globals to be lifted. This kind of pointer values represents a safe set of pointer values to lift.

`false`
: It disallows pointers to globals to be lifted.

`live_locations` (default: `[]`)

Definition of the locations which are live after the end of the program.

Live locations refer to locations that are live at the end of the program being specialized. This can happen during *modular-oriented* specialization, when the program is extracted from a larger program and then reinserted after specialization.

The variable is a list of strings (location names). A location may be:

- A global scalar variable, e.g., `"foo"`.
- A structure field. The notation is then `structure_type.field`, e.g., `point.x` where `point` is the name of a structure type (as given by the declaration `struct point {...}`), not an instance of that type. This restriction is due to the monovariance of the BTA on structures.

See also:

- Variable `static_locations` (syntax and restrictions on locations)
- File `actx.c` (fine analysis context description)
- Structure Mono/polyvariance
- User manual

`max_decls_size` (default: `100`)

Threshold for declarations ellipsis. If the number of global declarations (for types, variables and functions), at the beginning of the file is greater than the integer `max_decls_size`, then the declarations are hidden in all intermediate C files generated by Tempo (even color files) and can be found in a separate file suffixed by `.decl.h`. The declarations are replaced by an explanation comment and an `include` directive.

Similarly, for the compile-time specialized file `.cts.c`, if there are more declarations than the limit specified by `max_decl_size`, the declarations are put in a file suffixed by `.cts.h` and replaced by an `include` directive.

`max_width` (default: `78`)

Maximum width of the line for printing code. This variable is used for line wrapping when pretty-printing C code.

`memoize_aliases` (default: `true`)

Aliases memoized. This variable tells whether alias analysis results should be memoized at call sites. Experiments on some large files have shown that the alias analysis can be much faster with memoization (from more than 3 hours down to a quarter of an hour).

This flag is for internal purposes.

See also:

- Alias analysis phase

`minimize_holes` (default: `false`)

The number of holes in the run-time specialized program is minimized. A static variable in a dynamic context does not lead to any performance improvement from run-time specialization. A hole is generated for such a term. It can be useful to minimize the number of such holes to promote reuse of values stored in registers in the run-time specialized code.

`true`
: Number of holes is minimized.

`false`
: Number of holes is not minimized.

`output_mode` (default: `COLOR`)

Control of the intermediate files to generate. This variable specifies which intermediate files to generate when running Tempo phases. Possible values are:

`CONCISE`
: No (unnecessary) intermediate files are generated.

`COLOR`
: Only colored intermediate files are generated (see the variable `viewer`).

`FULL`
: All intermediate files are generated.

`FILES` *`suffixes`*
: Generation of files limited to the target of the last phase and the files specified by their suffixes in the string list *`suffixes`*, e.g., `FILES ["eta2.color","rawcts.c"]`.

`override_remove_compound_data` (default: `false`)

Remove residualized compound data assignments.

`poly_structs` (default: `"POLY_STRUCTS[]"`)

Mono/polyvariance of structures. This variable specifies whether (some) structures should be treated with monovariance or polyvariance, with respect to alias, side-effect and binding-time information, i.e., if the information should be merged for all instances of a given structure type (less precise but faster analysis) or computed for each structure instance (more precise but slower analysis). Possible values are:

`MONO_STRUCTS[`*structnames*`]`
: All structures whose name appears in the string list *structnames* are treated monovariantly. Other structures are treated polyvariantly.

`POLY_STRUCTS[`*structnames*`]`
: All structures whose name appears in the string list *structnames* are treated polyvariantly. Other structures are treated monovariantly.

Warning: This feature is operational only if variable `struct_version` is appropriately set.

See also:
- Variable: `struct_version`

`porky_flags` (default: `"-unused-types -ucf-opt -for-bound -no-index-mod -no-empty-fors -no-empty-table -control-simp -fold -globalize"`)

Flags for `porky`. This string variable permits additional flags to be passed on to `porky` when updating spd files just after parsing. The invocation pattern of `porky` is as follows.

```
    porky porky_flags file.spd file.porky_spd
```

In case the value of the variable is altered, the user must make sure to keep the `-globalize` flag as it turns static variables (in the C sense, i.e., local variables that retain their value between successive function calls) into global variables; the C subset treated by Tempo does not handle such static variables.

See also:
- Suif environment
- Command `porky`
- Suif parsing
- Variable `scc_flags`

`post_clean_up` (default: `true`)

Clean-up in post-processing is enabled. This variable tells whether clean-up should be performed during the post-processing, i.e.,
- Removal of unused labels,
- Removal of empty blocks and lines,
- Flattening of simply nested blocks.

`post_common_subexpression` (default: `true`)

Unfolding of expressions in consecutive assignments in post-processing.. This variable tells whether unfolding of expressions in consecutive assignments should be performed during the post-processing, i.e., whether a piece of code :

```
      x1 = e1;
      x2 = e2; /* x1 occurs in e2 */
```

should be turned into :

```
    x2 = e2[x1 <- e1];
```

warning : this transformation can be performed only if the clean up flag is activated.

`post_dead_code` (default: `true`)

Dead-code elimination in the post-processing. This variable tells whether dead-code elimination should be performed during the post-processing. Because dead-code elimination is quite buggy at the moment, this is turned off by default.

`post_do_inline` (default: `[]`)

Inlining in post-processing is forced. This variable holds a list of strings corresponding to function names. If inlining is enabled, it is performed (at least) on all these functions, regardless of other inlining filter flags.

See also:
- Variable `post_inlining` (to enable inlining)

`post_do_not_inline` (default: `[]`)

Inlining in post-processing is prevented. This variable holds a list of strings that are function names. Even if inlining is enabled, it is never performed on all those functions, regardless of other inlining filter flags.

See also:

- Variable `post_inlining` (to enable inlining)

`post_inlining` (default: `false`)

Inlining in post-processing is enabled. This variable tells whether inlining should be performed. (In this version of Tempo, functions returning a value are not inlined). Whether inlining of a given call actually takes place also depends on the total number of calls to this function and the size of its body.

See also:

- Variable `post_inlining_max_nb_calls` (threshold for the number of calls)
- Variable `post_inlining_max_nb_stmts` (threshold for the body size)

`post_inlining_max_nb_calls` (default: `5`)

Number of calls threshold for inlining. If the number of calls to a function is greater than this variable, then the function is not inlined.

See also:

- Variable `post_inlining` (to enable inlining)

`post_inlining_max_nb_stmts` (default: `25`)

Body size threshold for inlining. If the number of statements in the body of a function is greater than this variable, then the function is not inlined.

See also:

- Variable `post_inlining` (to enable inlining)

`post_inlining_mode` (default: `BLOCK`)

Style of inlining. This variable specifies the ``style'' of inlining. Possible values are:

`BLOCK`
: Inline functions as blocks: the body structure of the inlined function is retained.

`FLAT`
: Inlined functions are flattened: the body of the inlined function is merged into the list of the statements of the calling procedure.

See also:

- Inlining (User manual)

`post_inlining_renaming` (default: `true`)

Renaming of the parameters of inlined functions. In order to perform inlining, three operations are performed in the calling function.

- Declarations for the callee formals are inserted.
- Before the inlined call, assignments are inserted: the formals of the callee are assigned to the actual arguments provided by the caller.
- The inlined call is replaced by a copy of the body of the callee.

This can cause problem when the formal is of array type, as it is not possible in C to assign an array. (An array is a kind of constant pointer variable).

The possible values of `post_inlining_renaming` are:

`true`
: Renaming of all the parameters of a function before inlining so that they do not conflict with the actual arguments that are passed.

`false`
: This renaming is not performed.

Note that setting `post_inlining_renaming` to `false` can cause bugs. If your program has functions with parameters of pointer type and arguments of array type, the arrays is not referenced correctly once the function is inlined.

This flag is for internal purposes.

See also:

- Inlining (User manual)

`post_porky` (default: `false`)

Additional post-processing after CT specialization. The variable indicates whether, on top of Tempo post-processing, an additional post-processing using `porky` should be performed. Even if this variable is set to `true`, no additional post-processing is performed if the variable `post_porky_flags` is set to the empty string.

See also.

- Suif
- Command `porky`
- Post-processing phase
- Additional `porky` post-processing phase
- Variable `post_porky_flags` (to parameterize additional post-processing)

`post_porky_flags` (default: `"-ucf-opt -unused-syms"`)
`post_s2c_flags` (default: `"-omit-header"`)

Flags for additional post-processing after CT specialization. Those variables are used to parameterize an extra `porky` pass after CT specialization, if requested. See the description of the additional `porky` post-processing phase for the invocation pattern. Common porky options include:

`-dead-code`
: Simple dead-code elimination.

`-Dblocks`

Blocks created from inlining functions are dismantled.

`-ucf-opt`
: Simple optimization on unstructured control flow (branches and labels).

`-unused-syms`
: Removal of symbols that are never referenced and have no external linkage, or that have external linkage but are not defined in this file (i.e., no function body or variable definition).

See also:

- Suif
- Command `porky`
- Additional `porky` post-processing phase

`post_start_inlining_func` (default: `[]`)

Call-graph regions for performing inlining. This variable holds a list of strings which correspond to function names. Inlining is performed only ``below" those functions in the call-graph. In other words, no inlining is performed on the functions that may eventually call those that are specified in the `post_start_inlining_func` variable.

In particular, when this list is empty, there are no restriction on where inlining begins. In this case, inlining is allowed everywhere in the call-graph.

Note that this specification is overridden by the effect of the following variables:

- `post_inlining` (to enable inlining)
- `post_do_inline` (to force inlining)
- `post_do_not_inline` (to prevent inlining)
- `post_inlining_max_nb_calls` (threshold for the number of calls)
- `post_inlining_max_nb_stmts` (threshold for the body size)

`post_transform` (default: `true`)

Control over algebraic simplifications during post-processing. If true, some algebraic simplifications are performed during post-processing, *e.g.* addition to 0, multiplication by 0 or 1, etc.

`pre_common_subexpression` (default: `true`)

This variable has the same meaning as the corresponding post-processing variable **`post_common_subexpression`**.

`pre_do_inline` (default: `[]`)
`pre_do_not_inline` (default: `[]`)
`pre_inlining_max_nb_calls` (default: `5`)
`pre_inlining_max_nb_stmts` (default: `25`)
`pre_inlining_mode` (default: `BLOCK`)
`pre_inlining_renaming` (default: `true`)
`pre_start_inlining_func` (default: `[]`)

Pre-inlining variables. Those variables are parameters for pre-inlining occurring during early and late pre-processing. They have the same meaning as the corresponding post-processing variables (starting with a ``post_" prefix rather than `` pre_").

See also:

- Early pre-processing phase
- Late pre-processing phase
- Variable `pre_inlining_time`

`pre_inlining_time` (default: `NEVER`)

Pre-inlining time. The goal of the pre-processing phase is to perform function inlining. It is split into two phases, one early and one late. Each phase can be ignored or forced depending on the variable `pre_inlining_time`. Possible values are:

`NEVER`
: No inlining performed during either of the pre-processing phases.

`EARLY`
: Inlining limited to the first, early pre-processing phase. All specified functions, whether with a single return or with multiple returns, are inlined.

`LATE`
: Inlining only during the second, late pre-processing phase. Only functions with single returns are inlined in this phase. Functions with multiple returns are not inlined and a warning is issued. This is because the inlining of this kind of functions introduces gotos which cannot be eliminated at this stage.

`AUTO`
: Functions with multiple returns are inlined in the early pre-processing phase and functions with single returns are inlined in the late pre-processing phase.

In most cases, since pre-inlining makes the code less readable, the user wants (in case pre-inlining is really required) to use the `AUTO` value. This hides pre-inlining as much as possible, i.e., delay it in the course of the analysis phases.

Note that pre-inlining can be useful only when doing run-time specialization. For compile-time specialization, the inlining that occurs during the post-processing is sufficient. When inlining during run-time specialization is implemented, pre-inlining at both pre-processing stages will disappear.

See also:

- Early pre-processing
- Goto elimination
- Late pre-processing

`print_dir_in_html_title` (default: `false`)

Path in HTML title is displayed. This variable specifies whether the full path of the file should be displayed as a title in HTML generated files. Possible values are:

`true`
: The full path, e.g., `/home/jake/spec/power.eta2.html` is included.

`false`
: Only the name of the file, e.g., `power.eta2.html` is included.

`rebuild_compound_data` (default: `false`)

Reconstruction of initializations. This variables enables the reconstruction of initialized scalar (for the moment) variables during post-processing. This has not much be tested, hence the default value.

`reentrant_rts` (default: `true`)

Controlling the allocation strategy of the specialization buffer for run-time specialization. For run-time specialization, the buffer where specialized functions are recorded can either be global or local. In the former case, a unique specialization buffer is used for any specialization; this buffer is allocated statically. As a result, one specialization replaces a previously specialized function. In the latter case, many specialization functions can be used at the same time. A new buffer is thus allocated dynamically each time a specialization is triggered. This feature is essential when the function to specialize is recursive or when multiple specialized versions are needed at a given time. Regardless of the strategy, the run-time specializer always returns the address where the specialized function is stored.

Possible values for this variable are:

`true`
: A new specialization buffer is allocated for each specialization.

`false`
: A unique specialization buffer, statically allocated, is used for any specialization.

Note: Unlike the compile-time specializer, the run-time specializer does not perform a memoization of the functions previously specialized. As a consequence, recursive specialization with previous specialized values cause the specializer not to terminate. For example, backward branches in a byte-code interpreter.

`remote_target` (default: `""`)

Enabling the compilation of the templates and the run-time specializer to be performed on a remote machine. This variable can be set to a remote machine to define where the compilation of the templates and the run-time specializer should occur. This functionality is typically used to perform the analyses on a fast machine and to run the machine-dependent tasks on the target machine. This is implemented using `rsh`; the variable should thus be set to a valid machine name.

Note that this facility assumes that there is a shared file system between the machines used.

`residualize_all_icalls` (default: `true`)

Binding time of external indirect calls. This variable specifies whether indirect external function calls can be evaluated if possible (i.e., the function pointer and the arguments are static) at specialization time or should always be residualized (independently of arguments). Possible values are:

`true`
: Residualization of all indirect external calls at specialization-time.

`false`
: Evaluation of indirect external calls at specialization-time whenever possible.

See also:

- Variable `external_functions` (binding time of external functions)
- Indirect-call elimination phase

`rsd` (default: `"`*`$TEMPOHOME`*`/rtcg"`)

Run-time specializer internal directory. This is the path to the run-time specializer internals (library and template compiler). In practice, it should not be altered by the user.

`rts_buffer_size` (default: `20000`)

Size of the specialization buffer for the run-time specializer.

`rts_compiler_options` (default: `"-O0"`)

Setting the compiler options for the run-time specializer. Variable `rts_compiler_options` defines the compiler options to be used for the compilation of the run-time specializer. Unlike the templates, the run-time specializer can be compiled without any restriction as to the optimization level. Optimizing the run-time specializer mainly impacts the static computations of the program to specialize.

`rts_do_inline` (default: `[]`)
`rts_do_not_inline` (default: `[]`)

Inlining after runtime specialization. List of function names can be provided to instruct the runtime specializer to perform additional inling. It presently only is effective for Sparc. Pentium support is not fully implemented. In any case, this is still an experimental feature.

See also:

- Early pre-processing (Reference Manual)
- Late pre-processing (Reference Manual)
- Command `rts_compact_inline`
- Command `rts_inlining`
- Command `rts_optimized_inlining_one_pass`
- Command `rts_optimized_inlining_two_passes`

`s2c_flags` (default: `""`)

Flags for `s2c` and `s2st`. This variable can be used to provide options to `s2c` and `s2st` when producing *`file`*`.st` and *`file`*`.st` files.

See also:

- Suif Environment
- Suif Command: `s2c`
- Suif Command: `s2st`
- Suif abstract syntax generation

`scc_flags` (default: `""`)

Flags for scc. This string variable allows additional flags to be passed on to `scc` when parsing C files (program and analysis context) in order to produce `spd` files. The invocation pattern of `scc` is as follows.

```
    scc scc_flags -.spd file.c
```

Typical flags are `-Ipath` to specify the directory in which to find include files, and `-D` and `-U` to specify the values of C preprocessor variables.

See also:

- Suif
- Command `scc`
- Suif parsing
- Variable `porky_flags`

`sh_command` (default: `""`)

Shell setting. This string variable represents a command to be put in front of each Shell escape that Tempo runs to perform parsing, compilation, specialization and additional post-processing. It is mainly used to set Shell variables, e.g.,

```
sh_command := "
  SPE_FLAGS=\"-DSPECIALISATION\";
  CHORUS=/udd/pe/chorus/ClassiX1.1/sr;
  CHORUSFLAGS=\"-DCOMPAQ386 -DKERNEL -DMULTI_IOM
     -I$CHORUS/chorus_3.5\" ";
```

Common specifications include paths and `cpp` variable definitions for parsing.

`specialized_entry_point_name` (default: `""`)

Name of CT-specialized entry point. The compile-time-specialized entry point function is named after the content of this variable. If it is the empty string, the name of the specialized entry point function will be generated automatically.

`static_locations` (default: `[]`)

Initial static locations. This variable specifies which locations should initially be treated as static by the BTA analysis (apart from entry point arguments which are specified using variable `entry_point`). All other locations are assumed dynamic.

The variable is a list of strings (location names). A location may be:

- A global scalar variable, e.g., `"foo"`.
- A structure field. The notation is then `structure_type.field`, e.g., `point.x` where `point` is the name of a structure type (as given by the declaration `struct point {...}`), not of an instance of that type. This restriction is due to the monovariance of the BTA on structures.

Note that, at the moment, it is not possible to use a type name that has been created with the `typedef` construct. Only the structure name that follows the keyword `struct` in the declaration of the structure can be used. If there is no name following the keyword `struct` (an anonymous structure), Suif generates one automatically (named `__tmp_structn` where *n* is some number). It is very dangerous to rely on Suif to choose any particular name for this structure. For now, whenever possible, the user is encouraged to modify the source file to give explicit names for such anonymous structures. In any case, an error is reported if the location does not exist.

See also:
- Variable entry_point (binding time of arguments of the entry point)
- Variable external_functions (binding time of external functions)
- Analysis context file `actx.c` (more precise binding times)
- Specifying the analysis context (User manual)
- Structure Mono/polyvariance (User Manual)

`struct_version` (default: `MONO`)

Structure polyvariance. This variable indicates whether structure polyvariance is active or not. or not, with respect to alias, side-effect and binding-time information, i.e., if the information should be merged for all instances of a given structure type (less precise but faster analysis) or computed for each structure instance (more precise but slower analysis). Possible values are:

`MONO`
: Consider structure monovariance only. Use robust code.

`POLY`
: Possibly analyse structures with polyvariance, depending on variable `poly_structs`. Use experimental code.

See also:
- Binding-time analysis (producer)
- Variable: `poly_structs`

`templates_compiler_options` (default: `"-O0"`)

Setting the compiler options for the templates. Variable `templates_compiler_options` defines the compiler options to be used for the compilation of the templates used for run-time specialization.

Note that, in our experience, templates can be optimized up to `"-O2"`. Further optimization levels interfere with our template strategy.

`trim_specialized_func_ids` (default: `"false"`)

Trimming of specialized functions names. Specialized functions have funny names like `_Gfoo_1_2_3()`. When set to true, names look like `foo_3()`.

`verbose_aliases` (default: `true`)

Alias information displayed. This variable specifies whether alias information should be included as comments when displaying the program as (possibly colored) C text files. Possible values are:

`true`
: Alias information (possible values for pointer dereference) displayed in generated program text files, i.e., `c` and colored files (either `color` or `html`).

`false`
: This information is omitted.

Example:

```
get_addr(&a[1])

int get_addr(int *x)
{
  return *x/* a[] */;  /* alias information in comments */
}
```

Note that alias information is always removed from the `cts.C` file, unless the `verbose_aliases_in_specializations` flag is set.

See also:

- Visualization of analysis information

`verbose_aliases_in_specializations` (default: `false`)

Alias information displayed in specialized files. This variable specifies whether alias information should be mentioned when displaying the specialized program as a C text file. The alias information is not generally meaningful as the alias information is not re-computed on the specialized file but only ``inherited'' from the original program via specialization reconstructions. However, some users rely on it to ``guess'' what specialization did. This should not be encouraged; the action file, as well as the BTA and ETA files, contain the right information to predict specialization. Possible values are:

`true`
: Alias information (possible values for pointer dereference) is displayed.

`false`
: This information is omitted.

Note that variable `verbose_aliases` must be set to `true` for alias information to be maintained in the specialized program file.

`verbose_callsig` (default: `true`)

Polyvariance index is displayed. When displaying program as (possibly colored) C text files, this variable says whether an index should be displayed next to function calls and definitions in order to differentiate between different polyvariant binding-time instances. A function call or definition looks like this: `fun/*37*/(arg1,arg2).`

Note that both the BTA and the ETA may introduce different polyvariant instances; the resulting index differs from one analysis to the another. In other words, the index is only meaningful for a given file and should not be compared with indices in other files (whether previous or subsequent in the flow of phases). It does not differentiate either between the BTA and the ETA polyvariance: both kinds of polyvariance are ``merged''. Possible values are:

`true`
: Polyvariance index displayed in comment just after a function name at definition and call site.

`false`
: This information is omitted.

See also:

- Visualization of analysis information

`verbose_headers` (default: `true`)

Signature information is displayed. This variable specifies if function signature information should be mentioned when displaying program as (possibly colored) C text files. Possible values are:

`true`
: Signature information is displayed in generated program text files, i.e., `c` and colored files (either `color` or `html`). It includes side-effects of the function with associated binding times, as well as evaluation-time information.

`false`
: This information is omitted.

Example:

```
int get_addr(int *x)
/*
 binding times of read non_locals: a
 residual non-locals in: a
*/
{
  return *x/* a[] */;
}
```

The signature follows in comments just after the function definition header.

See also:

- Visualization of analysis information

`verbose_rts` (default: `false`)

Verbose generation of run-time specializer. When set to `true`, prints all the commands that build a run-time specializer.

`viewer` (default: `emacs`)

Viewer target for colored files. This variable specifies the format (and name) of colored files when displaying program as C text files. Possible values are:

`emacs`
: MIME text/enriched format (see Suffix `.color`)

`html`
: HTML format (see Suffix `.html`)

`wd` (default: `"."`)

Working directory. Tempo operates on files that are all in the same directory (or in an architecture-dependent sub-directory). This variable contains the path of the current working directory.

See also:

- Command `cd` (to change the working directory of the SML top-level as well)
- Introduction to section on Tempo files
- Variable `arch_dep_dir` (sub-directory for architecture-dependent RT files)

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

4 > Tempo — Installation Manual

4.1 The Tempo Specializer system

Tempo is an partial evaluator for C programs. The system consists of:

- A kernel (written in SML/NJ), that implements all the analyses and basic transformations (except specialization itself) and that sequences all phases.
- A library (written in C++) for compile-time specialization which performs source-to-source specialization actions.
- A template compiler (written in C) that prepares the assembly of object code templates for run-time specialization.
- A "front-end" (part of the Suif compiler) that handles the parsing of C and some additional source-to-source transformations.

The last official release of Tempo dates February 11th, 2003.

4.2 Requirements

You first need to have Standard ML of New Jersey (SML/NJ) installed. It can be downloaded from http://www.smlnj.org/.

Note however that using Tempo with recent versions of SML will result in a "bad magic number" error. As a workaround, please use SML/NJ version 110.0.7. This now old version can be downloaded from http://www.smlnj.org/dist/release/110.0.7/.

Tempo might work with a little more recent version of SML/NJ though, but this one will work for sure and, as explained in the SML Environment description, this has no impact on the performance of specialization itself.

Besides SML/NJ, the following tools must be installed on the machine for doing **runtime specialization**:

- `gcc` (or a slightly modified version of `lcc`)
- `gmake`
- `objdump`

No special tool is required for **compile-time specialization**, except a C/C++ compiler.

4.3 Available Platforms

The system runs on the following platforms:

- Sun OS 4.1
- Sun OS 5.5, *i.e.* Solaris 2.5
- Linux on PC

The system used to perform poorly on PC/Linux as we were relying on an old version of SML/NJ (see the SML Environment). However, only analysis and post-processing are affected, not compile-time nor run-time specialization themselves.

4.4 Content

The distribution for, *e.g.*, SunOS 4 contains the following directories.

- `tempo` : Root directory
- `tempo/bin/tempo` : Script that runs the Tempo top level
- `tempo/bin/`*`ARCH`*`/tempo` : Tempo top-level binary
- `tempo/ctcg/`... : Compile-time specializer header files
- `tempo/ctcg/`*`ARCH`*`/libctcg.a` : Compile-time specialization library
- `tempo/rtcg/tcc/`*`ARCH`*`/tcc` : Template compiler
- `tempo/rtcg/rts/`*`ARCH`*`/`... : Run-time specializer files
- `tempo/demos/`... : Demos directory
- `tempo/doc` : Documentation directory
- `tempo/suif/`*`MACHINE`*`/bin/`... : Suif binaries
- `tempo/suif/`*`MACHINE`*`/solib/`... : Dynamic libraries (Linux only)
- `tempo/emacs/`... : Emacs files for visualizing color-annotated files

where *`ARCH`* and *`MACHINE`* are among one of the following couples.

ARCH	*MACHINE*
`SunOS-4`	`sparc-sun-sunos4`
`SunOS-5`	`sparc-sun-sunos4`
`Linux-2`	`i386-linux`

There is no special reason why there are two different names for the same thing, just history of existing systems...

4.5 Installation Procedure

The distribution files are named `tempo`*`N.NNN_ARCH`*`.tar.gz`. There is one such file for each *`ARCH`* platform:

- SunOS 4.1: `SunOS-4`
- SunOS 5.1: `SunOS-5`
- Linux on PC: `Linux-2`

Each distribution contains all the required files. In order to install Tempo for different platforms at the same time, follow the installation procedure starting from the same root directory; appropriate sub-directories will be created.

The installation process is as follows:

1. Choose the location of your installation:

   ```
   mv tempoN.NNN_ARCH.tar.gz INSTALL_DIR
   cd INSTALL_DIR
   ```

2. Extract the distribution:

   ```
   gunzip tempoN.NNN_ARCH.tar.gz
   tar xf tempoN.NNN_ARCH.tar
   ```

3. Add *`INSTALL_DIR`*`/tempo/bin` to your path.
4. Modify the `tempo` script in the directory `INSTALL_DIR/tempo/bin` so that `SML_PATH` specifies the path used to invoke SML. This should include both the directory containing SML and the SML command itself, not just the directory.

You can now start Tempo by running the command `tempo`.

To view the color annotated files produced by Tempo under certain versions of Emacs (19.34 and potentially other versions including XEmacs, see the detailed section for more details), it is necessary to load the `format-patch` elisp file from the `.emacs` file:

- insert the line `(load "/myDir/tempo/work/format-patch")` in your `.emacs` This and other Emacs-related installation issues (potential problems) is thoroughly described the detailed section on Emacs installation.

4.6 Emacs Installation Issues

Two problems are known to occur when using Emacs to view "`.color`" files (MIME Enriched format) generated by Tempo:

1. Color file loading failing with a "`Wrong type argument`" error (or in some cases loading but displaying without certain color annotations).

2. Color file filling messing up the program.

This section describes the proper installation procedures for circumventing these problems.

4.6.1 Loading / Coloring Failure

Loading of color files may fail with a "`Wrong type argument`" error; in some cases loading may succeed but without displaying certain color annotations.

There exist a temporary patch to fix this for certain versions of Emacs:

- With Emacs 19.30 and lower, we have observed no problems. Thus, the patch should not be needed for such installations.
- With Emacs 19.34, the color files cannot be loaded. When loading a color file, Emacs generates a "Wrong type argument" error, and does not display the colored file. In some cases, the file will load, but not all the annotations will display correctly. Thus, the patch is needed for such installations.
- We have not thoroughly tested higher version of Emacs. (Emacs 20.2.4 seems to be OK though.) If Emacs cannot load the color files produced by Tempo, the patch is needed, and will probably work.
- Certain versions of X-Emacs display similar problems, and should thus use the patch.

To use the patch, in the following line in your `.emacs` file.

- under Emacs:

  ```
  (load "INSTALL_DIR/tempo/emacs/format-patch")
  ```

- under X-Emacs

  ```
  (load "INSTALL_DIR/tempo/emacs/format-patch-xemacs")
  ```

The patch is necessary because the enriched mode (used for displaying enriched MIME color files under Emacs) was updated between versions 19.30 and 19.34 of Emacs. The updated version no longer supports multiple color overlays for the same text. The support of multiple overlays (and the selection of "innermost" as the visible overlay for a piece of text) worked well with a recursive-descent way of generating colored files. For this reason, Tempo generates color files that have overlapping color overlays. This causes some versions of Emacs to generate an error when the color file is loaded, or simply to not display the appropriate colors.

The patch consists of a single elisp file that contains the appropriate functions from the standard encriched mode from version 19.30 of Emacs. When the file is loaded, the appropriate functions are replaced with the compatible, older (and more tolerant) versions. The patch is needed only as long as the the color files generated by Tempo have overlapping overlay annotations. However, most color files don't have these overlaps.

4.6.2 Automatic Filling

The color files generated by Tempo are sometimes inadvertently filled: all spaces and indentation is reduced to single spaces and lines are wrapped as if they were ordinary text (rather than part of a program), making the program unreadable.

The (temporary) solution is to disable automatic filling. For this, add the following line in your `.emacs` file.

```
(setq enriched-fill-after-visiting nil)
```

To restore automatic filling, do:

```
(setq enriched-fill-after-visiting 'ask)
```

.....

5 > Tempo — FAQ

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

5.1 Goal and conventions

The Tempo FAQ keeps track of the Frequently Asked Questions from Tempo developers and users.

We follow the convention that unless specified otherwise, all references below can be found in the User Manual, except references marked "Variable:" and "File:", which can be found in the Reference Manual.

5.2 The Tempo top level

What does the "=" prompt mean?

In the SML top level, every command has to be terminated with a semicolon. "=" indicates that the semicolon was not provided.

When I run the `tempo` command, nothing happens.

The `tempo` SML command requires three arguments, the name of the file (without the `.c` extension), the name of the starting phase, and the name of the ending phase. Because the `tempo` function is curried, if you leave off one of the arguments, a partially-applied function is returned.

See also: The SML Top Level, Using SML commands

What files are needed at minimum to specialize a program?

The C source file (`*file*.c`) and the SML file containing configuration information (`*file*.config.sml`). The specialization context file initializing the static parameters of the program (`*file*.sctx.c`) is needed for compile-time specialization. The SML file containing configuration information must at least define the `entry_point` variable.

See also: Specifying the Program to Specialize
Entry Point and Binding Times For Its Arguments
Variable: `entry_point`
File: `.c`
File: `.config.sml`
File: `.sctx.c`

Why doesn't ~ work to specify the path of a file or directory?

This is a "feature" of Tempo. Use the complete path name when changing directories. When you are in the directory of the program to specialize, just the file name is sufficient.

See also: Specifying the Program to Specialize

How can I specify an include path for the analysis phase and for the specialization phase?

Any flags can be provided to Suif (the parser used for the analyses) using the `scc_flags` variable. Any flags can be added to the compilation of the specializer using the `ctcg_cflags` variable.

See also: Variable: `scc_cflags`
Variable: `porky_flags`
Variable: `ctcg_cflags`

How can I set my preferences (e.g. **`viewer := html`** *rather than* **`emacs`***) once and for all rather than explicitly at each Tempo session?*

Create a file named `.tempo.sml` in your home directory (that is the full name, not just an file extension); it will be loaded each time you run the the `tempo` top level.

See also: File: `.tempo.sml`

How do I get out of Tempo?

Type control-D (*i.e.* end-of-file character).

5.3 Properties of global variables

What are the initial binding times of global variables?

Uninitialized global variables are dynamic by default. Initialized global variables are static by default. Global variables can be declared static explicitly using the `static_locations` variable.

See also: Binding Time of Global Variables
Variable: `static_locations`

What are the initial alias properties of global variables?

By default, it is assumed that there are no alias relationships among global variables. Aliases can be specified using the `set_analysis_context()` function in the `.actx.c` file.

See also: Specifying Complex Analysis Contexts
File: `.actx.c`

5.4 Properties of parameters and local variables

Why does a local variable have bottom binding time?

A local variable has binding time bottom until it is assigned.

See also: Colored Files

When does a parameter become a local variable?

If a parameter value is static and its address is used at runtime, the parameter is suppressed and a local variable is allocated. Take for example the following program:

```
void proc2(int *p)
{
  show(p);
}

void proc1(int c)
{
  proc2(&c);
}

void entry()
{
  proc1(1);
}
```

Here, `show()` is an external function: it can not be called at specialization time. There is no `.actx.c` file, so by default this function does not access the value stored at its parameter address.

The value of `c` is static. So the parameter `c` could be suppressed. On the other hand, the address of `c` is dynamic. So the parameter `c` should be residualized. In order to reconcile both aspects, the parameter `c` is suppressed and a local variable `c` is introduced in `proc1()` (where its address is needed). The specialized code looks like:

```
    static void proc2(int *p)
      {
        show(p);
      }

    static void proc1()
      {
        int c;

        proc2(&c);
      }

    extern void entry()
      {
        proc1();
      }
```

See also: Evaluation-time analysis (1) (Reference Manual)
Evaluation-time analysis (2) (Reference Manual)

5.5 Properties of array cells and structure and union fields

After the assignment `a[3] = 0`, why is `a[3]` still dynamic?

In Tempo, all array cells have the same binding time. Thus if any cell is dynamic, they are all dynamic, even after a static assignment to a particular cell.

See also: Array Monovariance
Binding Time of Arrays
Composite Locations and Binding Times

After the assignment `s.x = 0`, why is `s.x` still dynamic?

In Tempo, there is only one binding time description for all instances of a structure or union of a given type. Because just assigning the field of one instance to a static value does not ensure that the corresponding field of all instances has a static value, the binding time remains dynamic.

See also: Structure Mono/polyvariance
Binding Time of Structures and Unions
Composite Locations and Binding Times

Even though there is no assignment of an array cell (or structure or union field) to a dynamic value, why are all the array cells (or structure or union fields) dynamic?

In Tempo, all locations are considered dynamic, unless specified otherwise (except initialized scalar variables). This includes the content of arrays and the fields of structures and unions.

See also: Variable: `static_locations`
Binding Time of Global Variables
Binding Time of Arrays
Binding Time of Structures and Unions

Where does the `__tmp_struct1` structure type come from?

SUIF introduces names of this form when an anonymous structure type is defined in the program. It is not safe to rely on Suif to choose a particular name for a particular structure.

See also: Anonymous Structures or Unions

If `s` is the name of a structure with field `x` , why isn't `s.x` a valid entry in the list `static_locations` ?

The entry in the `static_locations` list uses the type name rather than the name of a particular instance, because of structure monovariance.

See also: Structure Mono/polyvariance
Binding Time of Structures and Unions

If `str` is the name of a structure type declared using `typedef`, why isn't `str.x` a valid entry in the list `static_locations` ?

Typedefs are eliminated by Suif. The entry for a structure or union field in the `static_locations` list has to be the name of the structure or union type, followed by "dot", followed by the name of the desired field.

See also: Restrictions on Typedef
Binding Time of Structures and Unions

Variable: `static_locations`

If `str` ***is the type of a structure or union, and*** `x` ***is a field of*** `str` ***having structure or union type, why is it an error to include*** `str.x` ***in the list*** `static_locations` ***?***

Structures as a whole do not have a static binding time. Instead, the individual fields should be specified to be static.

See also: Locations
Nested Structures and Unions

5.6 Specialization of a module

Tempo may be applied to only part of a complete application. In this situation it may be necessary to specify extra information about the context in which the specialized code will be invoked, about functions called from the code to specialized, and about the context following the invocation of the specialized code. Furthermore, the specialized code needs to be reintegrated into the application.

5.6.1 Calling context

How can the binding times of values pointed to by global variables or entry-point parameters be specified?

By default, pointed values have the same binding time as the pointer. A different binding time can be specified using the `set_analysis_context()` function in the `.actx.c` file.

See also: Specifying Complex Analysis Contexts
Binding Times of Parameters Passed By Reference
File: `.actx.c`

How can alias relationships among global variables or entry-point parameters be specified?

Aliases can be specified using the `set_analysis_context()` function in the `.actx.c` file.

See also: Specifying Complex Analysis Contexts
Initial Alias Relation
File: `.actx.c`

5.6.2 Returning context

The code to be specialized contains a static assignment to a global variable. When the variable is used in the application after the specialized code is called, how can I force the assignment to be residualized?

The variable `live_locations` allows one to specify what locations are used in the application after the call to the specialized code. More complex relationships can be specified using the `set_post_analysis_context()` function in the `.actx.c` file.

See also: Live Locations After The Entry Point
Variable: `live_locations`
Specifying Complex Analysis Contexts
File: `.actx.c`

5.6.3 External functions

When are external function calls residualized?

An external function call is residualized when any of the arguments are dynamic. An external function call is also residualized when the `external_functions` variable is set to a `RESIDUALIZE` list including the name of the function, or when the `external_functions` variable is set to an `EVALUATE` list not including the name of the function. By default, all external functions are residualized.

See also: Variable: `external_functions`
Binding Time of External Function Calls
Variable: `residualize_all_icalls`
Binding Time of External Indirect Function Calls

When are external function calls evaluated?

An external function call is never evaluated when any of the arguments are dynamic. When all of the arguments are static, an external function call is evaluated if the `external_functions` variable is set to an `EVALUATE` list including the name of the function, or when the `external_functions` variable is set to a `RESIDUALIZE` list not including the name of the function. Note that the decision whether all the arguments are static depends on just the argument values, not on values the arguments may point to.

See also: Variable: `external_functions`
Binding Time of External Function Calls
Variable: `residualize_all_icalls`
Binding Time of External Indirect Function Calls

How is the definition of an evaluated external function obtained?

Evaluated external functions have to be linked in with the specializer. Libraries and files defining external functions can be specified using the `ctcg_ldlibs` variable.

See also: Variable: `ctcg_ldlibs`

How can the effect of an external function on binding times be specified?

By default, an external call is assumed to have no effect on binding times. Binding-time effects can be specified using an abstract definition of the function in the `.actx.c` file.

See also: Abstract Function
Behavior of External Functions

How can the effect of an external function on aliases be specified?

By default, an external call is assumed to have no effect on aliases. Alias effects can be specified using an abstract definition of the function in the `.actx.c` file.

See also: Abstract Function
Behavior of External Functions

How can a location be made static in the `.actx.c` file?

A location can be made static by just assigning it to a constant, or to a global variable declared in the `.actx.c` file and specified as static using the `static_locations` variable.

See also: Variable: `static_locations`
Dummy Static Location

How can a location be made dynamic in the `.actx.c` file?

A location can be made dynamic just assigning it to a global variable declared in the `.actx.c` file or by assigning it to the result of a residualized external function call.

See also: Dummy Dynamic Location

How can aliases be described in the `.actx.c` file?

Alias relationships between variables can be described using the addresses of variables declared in the `.actx.c` file. A collection of aliases for a single location can be specified using a sequence of `if` statements.

See also: Initial Alias Relation

Pointer to a Set of Locations

How can memory allocation by external functions be modeled?

Memory allocation can be modeled by defining an abstract function that returns a pointer to some data structure defined in the `.actx.c` file.

See also: Named Memory Cells
Modeling Dynamic Memory Allocation
Limitations concerning casts (Limitations)

5.6.4 Global variables

During specialization, variables declared in the specialized program don't seem to communicate with variables declared in the rest of the application.

In this implementation of the specializer, global names are renamed during specialization. This can cause problems if they are are referenced externally during specialization, because references in already-compiled code will not be renamed appropriately. A solution is to include the `.sctx.h` file in all of the files of the application, and recompile the entire application at specialization time, rather than using existing `.o` files.

See also: Identifier Naming In Compile-Time Specialization (Limitations)
File: `.sctx.h`

5.6.5 The names of the specialized functions

Will the names of specialized functions conflict with the names of existing functions in the original application?

All specialized functions except the specialized entry point are declared as `static`, so they are not visible from any other file.

See also: Specifying Several Entry Points

What is the name of the specialized entry point?

A fresh name is chosen for the specialized entry point by default. The name can be specified using the variable `specialized_entry_point_name`.

See also: Specifying Several Entry Points
Variable: `specialized_entry_point_name`

5.7 Running the Analysis

My analysis takes a lot of time to compute? Do I have to always re-run it from scratch?

As long as your source files (".c" and ".actx.c") and configuration file (".config.sml") do not change, you can always restart from some already generated intermediate file (suffixed ".as") The `an` command runs the whole sequence of analysis phases, starting from the C files. But you may use the `tempo` top-level command to run only a fraction of thoses analyses.

See also: Command: `tempo`
Variable: `output_mode`

5.8 Compile-time specialization

How are the values of the static parameters and globals provided before specialization?

Static variables are initialized using the `.sctx.c` file

See also: Compile-Time Specialization
File: `.sctx.c`

Why are there unbound variables when compiling the `.sctx.c` file?

Tempo changes the names of some variables. Thus, the `.sctx.c` file must #include the `.sctx.h` to perform the renaming.

See also: Compile-Time Specialization
File: `.sctx.h`

What does it mean when the specializer just sits there, doing nothing?

Perhaps your program is in an infinite loop (see also Static Loops Containing Dynamic Exits in the Known Bugs collection), or perhaps there is too much specialization being performed. If a single static value varies a lot, you may end up creating many very similar specialized functions. In this case it may be useful to make some static values dynamic, to perform less specialization.

See also: Turn a Location Dynamic

What does it mean when there is a lot of garbage collection during specialization?

Once you see messages about garbage collection, specialization has successfully completed, and Tempo is performing postprocessing. Probably your specialized program is just very large. In this case it may be useful to make some static values dynamic, to perform less specialization.

See also: Turn a Location Dynamic

What does it mean if my static parameters are initialized to zero or to some strange random values?

Initialization of the static parameters is specified by a user-written function `set_specialization_context()` in a file suffixed `.sctx.c` . A common error is to set the local variables of this function, not their pointer values (*i.e.* not the static arguments of the entry point).

See also: Invocation of a Compile-Time Specializer
Common Errors in Setting Specialization Contexts

What does it mean if there is a bus error, segmentation fault, or illegal instruction error during specialization?

If the error does not come rapidly when specialization starts, it may mean that the specialized code has exceeded the allocated buffer size. A larger buffer can be requested using the `explicit_cts_bufsize` variable.

If the error comes rapidly when specialization starts, it is likely that some data structure is not properly initialized (see Common Errors in Setting Specialization Contexts). Alternatively, there may be an error in your program. You can use the debugger `gdb` by compiling the specializer files with the `-g` option (specified by adding `-g` to the value of `ctcg_cflags`). Errors are likely to occur in the `.ev.c` file. Each function in this file contains a comment indicating the name of the function in the source program that it comes from.

See also: Common Errors in Setting Specialization Contexts
Variable: `explicit_cts_bufsize`
Variable: `ctcg_cflags`
File: `.ev.c`

How can I see the result of the specialization before post-processing?

The result of raw specialization is dumped into the `file.rawcts.as` file. It is not turned into C text by default as it is post-processed right away. To view it as C text, type this command at the top level.

```
as2c "file.rawcts.as";
```

The corresponding *file.rawcts.c* will be generated in the working directory. Alternatively, you may also set variable `output_mode`.

See also: Command: `as2c` (generation of C text from abstract syntax format)
Variable: `output_mode` (control of the intermediate files to generate)

Is there any way to specify that a single function should always be inlined and no other function should ever be inlined, without saying `do_not_inline` and listing the names of all the functions?

The SML variable `post_do_inline` is bound to a list of functions that must be inlined. The SML variable `post_do_not_inline` is bound to a list of functions that must not be inlined. But setting `post_do_inline` to a list of function names does not does not mean that no other functions will be inlined. Similarly setting `post_do_not_inline` to a list of function names does not mean that all other functions will be inlined. Instead the inlining of functions is controlled by the variables `post_inlining_max_nb_stmts` and `post_inlining_max_nb_calls` . These integer variables are the thresholds at which to stop inlining functions not explicitly included in the `post_do_inline` list. Thus to inline the function "foo", but no other functions, set the flags as follows:

```
post_inlining := true;
post_do_inline := ["foo"];
post_inlining_max_nb_stmts := 0;
post_inlining_max_nb_calls := 0;
```

Note that no inlining happens, regardless of the value of `post_do_inline` if `post_inlining` is set to false.

If you want the calls within "foo" to be inlined as well, use the following settings:

```
post_inlining := true;
post_do_inline := ["foo"];
post_start_inlining_func := ["foo"];
```

Variable `start_inlining_func` specifies the function at which inlining begins. Function calls not directly or indirectly within these functions are not inlined. inlining, and stops any functions outside the branch of the function(s) specified from being inlined.

See also: Post-processing
Variable: `post_do_inline`
Variable: `post_do_not_inline`
Variable: `post_inlining`
Variable: `post_inlining_max_nb_calls`
Variable: `post_inlining_max_nb_stmts`
Variable: `post_inlining_mode`
Variable: `post_inlining_renaming`
Variable: `post_start_inlining_func`

5.9 Run-time specialization

Once the run-time specializer is created, how is it used in the application?

The run-time specializer is invoked with the list of values for the static arguments. The result is a pointer to a specialized function that takes as inputs the values of the rest of the arguments.

See also: Invocation of a Run-Time Specializer

The run-time specializer is generated in an architecture-dependent directory; how do I know its name, how can I change it?

All architecture-dependent files are generated, starting from in the working directory, in a sub-directory given by variable `arch_dep_dir`. If you want it to be written in the working directory instead, you may assign `arch_dep_dir` to `"."`.

See also: Variable: `arch_dep_dir`

How can multiple specializations of a single function be created?

By default, the run-time specializer allocate new buffer space each time the specializer is called and for each function. Other buffer manipulation strategies can be implemented by the user. The `reentrant` flag can be set to false to instruct the specializer to use a single buffer to store the specialized code. Subsequent specializations overwrite previous specializations.

See also: Recursive and Multiple Run-Time Specializations
Variable: `reentrant_rts`

What does a segmentation fault during run-time specialization mean?

If the segmentation fault occurs during specialization, as opposed to when running the specialized program, the problem might be that the program is recursive. If the program is recursive, the `reentrant_rts` flag must be set to true.

See also: Recursive and Multiple Run-Time Specializations
Variable: `reentrant_rts`

5.10 Visualization

Why does emacs give an error when loading color files?

There is a problem with the enriched mode of emacs version 19.34. A patch is available, as described in the installation manual.

See also: Emacs installation issues (Installation Manual)

Why are some of the color annotations missing when viewing the color files under emacs?

There is a problem with the enriched mode of emacs version 19.34. A patch is available, as described in the installation manual.

See also: Emacs installation issues (Installation Manual)

Why are lines being wrapped when viewing color files under emacs?

The enriched mode of emacs will sometimes automatically perform filling at a specific width. The solution is to turn off filling, as described in the installation manual.

See also: Emacs installation issues (Installation Manual)

Why does my color file have lots of strange annotation when I view it under emacs?

The enriched mode of emacs will sometimes fail to display the file. Reloading the file normally solves the problem.

Why don't the color files look like the source program?

Several transformations are performed on the source program by Suif and by Tempo. For example, Suif breaks up complex conditions, rewrites all pointer dereferences to apply to a single variable, and rewrites all loops as do-while loops. Early phases of Tempo eliminate gotos and indirect function calls.

See also: Parsing (Reference Manual)
Indirect call elimination (Reference Manual)
Goto elimination (Reference Manual)
Anonymous Structures or Unions
Display of Alias information
Pointers to Strings and Arrays

How can I get rid of all the comments in the color files?

Alias comments can be eliminated by setting `verbose_aliases` variable to false. Function-header comments can be eliminated by setting `verbose_headers` variable to false.

See also: Variable: `verbose_aliases`
Variable: `verbose_headers`
Variable: `verbose_callsig`

Variable: `verbose_aliases_in_specializations`

What do the colors mean?

There is a legend at the top that describes the meaning of each color.

See also: Visualization of Colored Files

What happened to the variable declarations?

If there are more global variable declarations than the threshold specified by the variable `max_decls_size`, the variable declarations are put in a `.decl.h` file.

See also: Variable: `max_decls_size`
File: `.decl.h`

How do I print a color file in emacs?

```
M-x ps-print-buffer-with-faces
```

Prints the current buffer on the current printer.

```
M-1 M-x ps-print-buffer-with-faces
```

Prompts for the name of a file where to save a postscript version (with colors) of the current buffer.

```
M-x eval-expression
```

```
(ps-print-buffer-with-faces "file.ps")
```

Saves a postscript version (with colors) of the current buffer into *`file`*`.ps`.

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

6 > Tempo — Limitations

Caution: The limitations listed in this document are not meant to be exhaustive.

6.1 Missing Features

Some features are missing in Tempo. Though some of them are still challenges, others are just due to a (relative) lack of manpower.

6.1.1 Limitations of the Input Language

Though Tempo internally works on a small C subset, it accepts most ANSI C constructions. However, the following constructs are not handled:

- Bit fields,
- Switch (with 4 cases or more),
- Functions with a variable number of arguments.
- Non local jumps (`setjmp`/`longjmp`)

Suif automatically rewrites `switch`es with at most 3 cases into a cascade of `if`s. `Switch`es with 4 cases or more have to be manually rewritten into a cascade of `if`s.

6.1.2 Limitations on Aliases and Casts

For the time being, in order to guarantee that the alias analysis will produce correct results, the following rules should be obeyed:

1. Pointer arithmetic can only be applied to pointers pointing to the contents of an array. (A warning is raised.)
2. The only scalar that can be cast to a pointer is `0`, i.e. the pointer `NULL` pointer. (A fatal error is raised: Tempo stops.)
3. Casting a pointer to a pointer is allowed only if the pointed objects are scalars. (A pointer is a scalar.) A warning is raised when the target type is not a pointer to a scalar.
4. If a pointer to a structure or union is cast to another pointer type it should be considered ``opaque'' (i.e. not dereferenced, neither for reading nor writing) until it is cast back into the original type.

All cast restrictions apply to unions that are used as casts, i.e. when an object is stored in the union under a certain name (and type) and retrieved under another name.

Some of these limitations are due to the monovariance of the alias analysis, others to the lack of a store model in Tempo. (Locations rely on names rather than physical, even abstract, memory.) You may want to check out the experimental structure polyvariance though.

In order to express non-opaque casts between structure types (as may be needed for object-oriented programming), consider modeling them explicitly as abstract functions in the `actx.c` analysis context file.

For the analysis to be correct, Tempo has to know all the possible aliases of locations if they have an impact on the semantics. This can also specified using the `actx.c` analysis context file.

6.1.3 Mutually Recursive Structure Declarations

Tempo doesn't give an error if there is mutual recursion among structure declarations. However side-effect analysis just might not collect as many read/written non-locals as it should. Self recursion is OK though.

6.1.4 Binding-Time Imprecision Due to Goto elimination

Tempo rewrites gotos using a combination of new variables, conditionals, while loops, breaks, and continues, using an algorithm developed by Ana Erosa and Laurie Hendren at McGill University (described in ACAPS Technical Memo 76). The conditionals introduced by the goto elimination can interfere with each other causing binding-time problems. Consider the following program:

```
int if_test(int S, int D)
{
  int x,y;

  if (S > 32) goto L33;
  x=150;
  if (D > 32) goto L38;
  y=100;
  goto L38;
L33:
  x=200;
L38:
  return x + y;
}
```

This program corresponds to the following program without conditionals:

```
int if_test(int S, int D)
{
  int x,y;

  if (S <= 32) {
    x = 150;
    if (D <= 32) {
      y = 100;
    }
  else x = 200;
  return x + y;
}
```

Suppose we specialize these programs with `S` static and `D` dynamic. In the second case, `S` should be static, because it is not assigned within the dynamic conditional. In the first program, the goto elimination moves what should be the else branch of a static conditional, i.e. the assignment of `x` to `200`, inside of the dynamic conditional. This behavior is shown by the following `at.color` (or `at.html` file). (If this document is printed in black and white, this example will not tell you much.)

```
extern int if_test_1/*0*/(int S, int D)   {
    int goto1_L33;
    int goto1_L38;
    int x;
    int y;

    goto1_L33 = 0;
    goto1_L38 = 0;
    goto1_L33 = 32 < S;
    if (! goto1_L33)
      {
        x = 150;
      }

    if (goto1_L33 || 32 >= D)
      {
        if (! goto1_L33)
          {
            y = 100;
            goto1_L38 = 1;
          }

        if (! goto1_L38)
          {
            goto1_L33 = 0;
            x = 200;
          }

      }
    goto1_L38 = 0;
    return x + y;
  }
```

The slightly more complicated program below, exhibits similar behavior.

```
int if_test(int S, int D)
{
  int x,y;

  if (S > 32) goto L33;
  x=150;
  if (D > 32) goto L27;
  y=100;
  goto L38;
L27:
  y=45;
  goto L38;
L33:
  x=200;
L38:
  return x + y;
}
```

Here the lines below label `L27` correspond to the else branch of the inner dynamic conditional statement. In this case, the else branch of the outer conditional is not actually placed within the inner conditional, but dynamic tests of goto variables are placed around the code, making `x` again dynamic:

```
extern int if_test_1/*0*/(int S, int D)  {
    int goto1_L27;
    int goto1_L33;
    int goto1_L38;
    int x;
    int y;

    goto1_L27 = 0;
    goto1_L33 = 0;
    goto1_L38 = 0;
    if (32 >= S)
      {
        x = 150;
        if (32 >= D)
          {
            y = 100;
            goto1_L38 = 1;
          }
        if (! goto1_L38)
          {
            goto1_L27 = 0;
            y = 45;
            goto1_L38 = 1;
          }
      }

    if (! goto1_L38 && ! goto1_L38)
      {
        goto1_L33 = 0;
        x = 200;
      }
    goto1_L38 = 0;
    return x + y;
  }
```

Essentially the problem is that because else branches are not introduced by the goto elimination, dynamic tests are introduced to jump over the else branch of the static conditional in all cases that can result from taking its then branch.

The PORKY_DEFAULTS and PORKY_PRE_DEFAULTS phases of Suif rearrange goto statements in a way that can also be detrimental to binding times. In many cases, these options can be turned off (which is the default for this version of Tempo), but in some cases porky will not work if these phases have not been performed.

6.1.5 Identifier Naming In Compile-Time Specialization

In the implementation of the compile-time specializer for this version of Tempo, global names are renamed during specialization. This can cause problems if they are external, or are referenced externally during specialization.

The renaming of global variables is controlled by the .sctx.h file, which for the global variable var might contain a definition akin to:

```
#define var _store.var
```

As long as the .sctx.h file is included into all C files used by the specializer, this works fine. However, if the program slice being specialized is part of a larger set of object files that are linked together, problems may occur. Specifically, object files that are not recompiled into the specializer will all refer to var , whereas those files that were recompiled will refer to _store.var. If the specializer depends upon side effects from other parts of the program to var, it is likely to fail.

One solution is to only give Tempo the relevant program slice during analysis, and then give it the whole program during specialization. This allows any redefinitions in the sctx.h file to take effect without incurring a large overhead (unless the program is very large). Of course, this only works if source code is available for the whole program, and may require some minor rewriting of the program. Other alternative solutions include referencing such variables through functions or to copy their value when the specialization starts.

This problem was planed to be removed in a future version, by avoiding any renaming of variables during specialization.

6.1.6 Limitations on Multiple Compile-Time Specialization

There is no specific support for multiple compile-time specialization. In particular, when linking simultaneously different specializations, redundant definitions may appear. They must be removed by hand. Similarly, useless declarations may occur.

Note that identifiers are renamed by compile-time specialization. Global identifier are prefixed with _G, local identifiers are prefixed with _L (or _Y for nested block declarations) and the name of the function they belong to.

6.1.7 Limitations to Run-Time Specialization

At the moment, the run-time specializer does not support:

- Store management in rebuilt conditionals
- Management of specializations, i.e. no memoization or memory reuse

This may not preserve the semantics of the run-time specialized function (see below).

Only `gcc` and a slightly modified version of `lcc` can be used for constructing a run-time specializer; see variable `compiler`.

Also, the runtime inlining has been implemented for Sparc and partly for Pentium, but this is still an experimental (and undocumented) feature.

6.2 Limitations on the Preservation of the Semantics

While a program transformer ideally always preserves the semantics of programs, this might not always be the case with Tempo, in very limited cases though!

6.2.1 Limitations Common to Off-Line Partial Evaluators

As is the case for most off-line partial evaluator, Tempo will blindly evaluate all static expression. In particular,

- Tempo is stuck if a static expression loops. If that expression happens to be dead code (remember that this is undecidable), Tempo will loop whereas the original program terminates.
- If a static expression raises an error (like division by zero), Tempo's behavior is unpredictable but probably wrong.

6.2.2 Naming of Identifiers

Tempo does not check that the identifiers that it creates do not already exist. Uncommon suffixes and counters seem to do the trick in most cases. However, theoretically, a name clash is possible.

6.2.3 Initializations

Other limitations are due to the transformations that Tempo performs in order to work on a smaller C subset. In particular, definitions of global variables of scalar type are turned into assignments in the entry point function to facilitate analysis. At the same time, the initializer is removed from the variable declaration, which may be a problem if the variable is actually used in some other files.

6.2.4 Limitations to Run-Time Specialization

As mentioned above, there is no store management in the run-time specialization. This can produce an incorrect result from run-time specialization. A conditional statement is annotated ``rebuild'' when the test is dynamic by there is code that can be evaluated during specialization in one of the branches. In compile-time specialization, when specializing each branch the store is initialized to the store that existed just after specializing the conditional test. Thus the static side-effects in one branch have no influence on the other branch. This is not the case for run-time specialization. In run-time specialization the store that exists at the end of specializing the first branch is the store that is used to specialize the second branch. This strategy can lead to wrong results, as illustrated by the following example:

```
Original program     Wrong specialization    Correct specialization

x=1;
if(Dyn)              if(Dyn)                 if(Dyn)
{                    {                       {
  x=2;                 f(2);                   f(2);
  f(x);              }                       }
}                    else                    else
else                 {                       {
{                      g(2);                   g(1);
  g(x);              }                       }
}
```

Note that no warning will be issued. If such a situation is spotted, one possible workaround is to prevent speculative evaluation by turning dynamic the dangerous side-effects.

Another effect of the lack of store management is that there is no memoization of the current store when specializing a function. This is in contrast with compile-time specialization that able to recognize that a function has already been specialized (or is being specialized) and to yield a call to that specialized function rather than invoking the specializer anew. As a result, run-time specialization may loop if the specialized function is recursive. This is typically the case when specializing interpreters with backward jumps. There are workarounds though; see Recursive and Multiple Run-Time Specializations in the User's Manual.

6.3 Limitations on the Improvements Gained by Specialization

Partial evaluation is not magic. There are several reasons why the specialized program could be worse that the original one. In particular,

- As is the case for most program transformations, even though it respects the operational semantics, specialization may alter the cache's behavior. This can have large (positive or negative) impact on modern architectures.
- Cache impact is important in particular for loop unrolling. As Tempo blindly evaluates any static expression, it will unroll all static loops, whether large or small. Not only the program can be much bigger, but the instruction cache behavior can be very bad.
- Initializations are turned into explicit assignments so that Tempo can work on a smaller C subset. While theoretically less efficient (execution time and program size), this has not been a problem in practice in our examples.

6.4 (Ex-)plans for the Future

The following features had been being considered for the next releases of Tempo. Some had already been implemented in large part and being used internally, although they did not make it into a released version. They are listed in order of release (ex-)likelihood.

1. Switch (full support, instead of the rewriting into a cascade of ifs for switches with less than 4 cases)
2. Inlining at run time *[implemented for Sparc, mostly for Pentium, still experimental]*
3. User-refined memory management (compile-time specialization)
4. Data specialization *[implemented, but still experimental]*
5. More efficient compile-time specialization
6. Structure (and union) polyvariance *[implemented, but still experimental]*
7. Better handling of casts

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

7 > Tempo — Known Bugs

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

7.1 Recursive Entry Point

Because of dark implementation details, the action analysis may produce wrong annotations if the entry point is recursive (actually when there is mutual recursion). The workaround is to encapsulate the recursive entry point in a (non-recursive) function.

7.2 Mutually Recursive Structure Declarations

Tempo doesn't give an error if there is mutual recursion among structure declarations. However side-effect analysis just might not collect as many read/written non-locals as it should. Self recursion is OK though.

7.3 Static loops containing dynamic exits

Static loops containing dynamic conditionals which exit the loop with a dynamic `break`, `return`, or `continue`, will generate a specialization whose semantics is not equivalent to the original code. The problem arises from the analysis not capturing the dynamic conditions which exit the loop. For example, consider the following loop.

```
for( n = 0; n < 5; n++ )
{
  if (Dyn)
    return Stat;
  *side_effect++;
}
```

Because the loop is static, the `side_effect` variable will be incremented 5 times during specialization even if the dynamic `Dyn` condition exits the loop, say, after 3 iterations.

For the same reason, Tempo will loop forever during specialization inside infinite static loops containing dynamic conditionals that exit the loop. E.g.,

```
do
{
  if (Dyn)
    return Stat;
  Something;
}
while (1);
```

The workaround is to change the loop conditional in order to take into account the dynamic control. In the following example, the loop condition becomes dynamic because it is side-effected under dynamic control.

```
exit = 1;
do
{
  if (Dyn)
  {
    exit = 0;
    return Stat;
  }
  Something;
}
while (exit);
```

Another approach is to explicitly force a dynamic conditional, and manually restore the static value in the residualized program. For example,

```
do
{
  if (Dyn)
    return Stat;
  Something;
}
while (dyn_exit);
```

Variable `dyn_exit` must then be set to `1` in the specialized program.

Of course, the loop will not be evaluated away since it is has become dynamic.

7.4 Return in conditionals

If there is a `return` in the branch of a conditional, then you might also need `returns` in other branches.

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

8 > Tempo — SML Environment

8.1 Use of SML in Tempo

The Tempo Specializer is mainly built in Standard ML (SML). The user interacts with the Tempo system through an interactive top level that encapsulates all functionalities. In practice, the user does not need to know much about SML (see SML Top Level in the User's Manual).

This release of Tempo works with version 110.0.7 of SML of New Jersey. This is quite an old version. See the installation instruction for additional details. Information regarding this version of SML/NJ can be found at http://www.smlnj.org/dist/release/110.0.7/.

Tempo might work with a little more recent version of SML/NJ though, but version 110.0.7 will work for sure, and getting the latest SML version is not crucial for using Tempo anyway. Releases of Tempo relying on older version of SML/NJ used to performs very badly on PC. However, only analysis and post-processing used to be affected, not compile-time nor run-time specialization performance themselves.

8.2 Documentation

If you want to know more about SML in order to built more complex interaction with Tempo, the following documents should provide you with some information.

- An introduction to Standard ML by Bob Harper
- SML User's guide
- SML/NJ Base environment, System modules, Library, Release notes of version 110.0.7
- Various tools:
  CML, Debugger, eXene, Info, makeml, ML-Lex, ML-Yacc, ML-Twig, Profile, SourceGroup

Other valuable SML information can be obtained form the SML/NJ web site (http://www.smlnj.org/).

.....

9 > Tempo — Suif Environment

9.1 Use of Suif in Tempo

The front-end of Tempo relies on the Suif compiler, developed at Standford University. Tempo uses Suif only for parsing and doing simple source-to-source transformations.

This version of Tempo relies on version 1.1.2 of Suif. (A port on version 2.0 had been planed.) See the suif web pages for additional details (http://suif.stanford.edu/suif/).

9.2 Commands

In practice, the user only needs to run the `tempo` command; all other Suif commands are run indirectly from the Tempo top-level.

`porky`

Assorted code transformations. In practice, the user does not have to invoke this command explicitly; it is run when needed through Tempo SML top-level commands.

See also:

- `porky` manual
- Suif parsing
- Additional post-processing
- Variable: `porky_flags`

- Variable: `post_porky_flags`

`scc`

Suif compiler driver program. In practice, the user does not have to invoke this command explicitly; it is run when needed through Tempo SML top-level commands.

See also:
- `scc` manual
- Suif parsing
- Additional post-processing
- Variable: `scc_flags`

`snoot`

Translate pre-processed C to SUIF. In practice, the user does not have to invoke this command explicitly; it is run by the `scc` shell command.

See also:
- `snoot` manual
- Suif parsing

`s2c`

Convert a SUIF file to C. This tool is used to translate the Suif abstract syntax tree (in `spd` format) into a C text representation. In practice, the user does not have to invoke this command explicitly; it is run when needed through Tempo SML top-level commands.

In ANSI C, an uninitialized variable is implicitly set to 0 (NULL). So, `s2c` removes the useless initialization to 0. In order to feed Tempo with the original source C file, we have patched the orginial, distributed `s2c` so that it keeps variable initializations to zero. Thus we can know that the variable is static as, by default, uninitialized variables are dynamic.

See also:

- `s2c` manual
- Suif parsing
- Additional post-processing

`s2st`

Suif abstract syntax printer. This tool is used to translate the Suif abstract syntax tree (in `spd` format) into a representation readable by Tempo (in `st` format). In practice, the user does not have to invoke this command explicitly; it is run when needed through Tempo SML top-level commands.

This commands is behaves exactly as `s2c` (in particular, it accepts the same options) except that it displays a textual tree representation of the Suif abstract syntax instead of C text.

In ANSI C, an uninitialized variable is implicitly set to 0 (NULL). So, `s2c` removes the useless initialization to 0. In order to feed Tempo with the original source C file, we have patched the orginial, distributed `s2c` so that it keeps variable initializations to zero. Thus we can know that the variable is static as, by default, uninitialized variables are dynamic.

As `s2c`, `s2st` keeps variable initializations to zero (see `s2c` entry above).

See also:

- `s2c` manual
- Suif abstract syntax generation

9.3 Variables

Below are shell variables used by Suif. Except `TEMPOSUIFHOME`, you do not have to set them explicitly as the Tempo top level run by the `tempo` shell-level command will set them automatically for you.

`MACHINE`

Target Architecture. This variable is set automatically to one of the following possible values:

- `i386-linux`
- `sparc-sun-sunos4`

At the moment, we use to SunOS4-Solaris compatibility package. That is why a case like `sparc-sun-sunos5` does not exist.

`SUIFHOME`

Suif main directory. This variable is set to `$TEMPOHOME/suif` where `TEMPOHOME` is the name of the installation directory of Tempo.

`SUIFPATH`

Path to binaries. This variable is set to `$SUIFHOME/$MACHINE/bin`.

`TEMPOSUIFHOME`

Tempo-Suif main directory. This variable can be used to override the default setting of Suif variable `SUIFHOME` (performed by the tempo shell-level command) in order to run Suif commands other than those provided with Tempo. Be careful though that `s2st` will not be found outside of the distribution of Tempo (see `s2st` above).

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

10 > Tempo — Demos

A set of demos is provided with Tempo, in the distribution files.

10.1 Basic

- power: a simple basic example to start with.

10.2 Binding time

- context-sens: illustrates context-sensitivity of BTA.
- flow-sens: illustrates flow-sensitivity of BTA.
- Interpret: simple imperative interpreters.
- return-sens: illustrates return-sensitivity of BTA.

10.3 Interpretation

- new_printpower: a print with simple formating.
- Interpret: simple imperative interpreters.

10.4 System

- bpf: this code is derived from the Stanford/CMU enet packet filter (on SunOS5 only).
- rpc: Sun RPC implementation (on SunOS5 only).

10.5 Numerical

- cubic_splines: Glueck's cubic splines computation.
- matmult: matrix product.
- romberg: performs Romberg integration.
- computes the Chebitchev function.
- FFT: the Fast Fourier Transformation.

Most of these directories have a read-only sub-directory `Source` containing the required files. See the `README` files for more information.

.....

11 > Tempo — History and Contributions

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

11.1 History

(by Charles Consel)

Partial evaluation is a program transformation approach which is applicable to a large class of programming languages. Although this simple statement is well-accepted in the community, for many years, most research efforts focused on the partial evaluation of prototype languages, instead of widely used languages. As a result, until recently, the potentials of partial evaluation had mainly been demonstrated on toy examples. This situation often led to consider partial evaluation as a minor topic considering the lack of impact it had as a technique (or tool) on other research areas.

The need to go beyond a specific programming language in studying partial evaluation became clear to me when I arrived at Oregon Graduate Institute in 1992 and started collaborating with a group of operating system researchers, led by Calton Pu and Jonathan Walpole. Surprisingly, they were interested in investigating the use of partial evaluation in an operating system. Since I was then working on a partial evaluator for pure Scheme programs (Schism), I naturally attempted to promote functional programming for this project (later named Synthetix). In fact, soon enough, we realized that a functional language would lead us, not only to work on partial evaluation problems raised by operating systems, but also to address implementation problems of functional languages (some of these issues have been studied by the Fox project at CMU).

As a consequence, we decided to use a language whose implementation fulfilled the basic requirements of operating systems so as to focus our effort on the partial evaluation aspects of the study --- not the language design and implementation aspects. In this context, the C language seemed a natural choice. Because a language like C is widely used, the design and implementation of a partial evaluator could be attacked from a new viewpoint: instead of guessing interesting problems, the partial evaluator could be developed with some practical goals in mind. Indeed, the variations in designing and implementing a partial evaluator are boundless; as a tool, what makes a good partial evaluator is its usefulness. That is, it should offer a set of features that enables it to successfully handle practical situations. Because various domains require various features, and given my undergoing collaboration in operating system, I initially restricted my study to systems programs.

In fact, targeting a particular domain brought up a lot of new and challenging problems that I could not have come up with by myself. In my opinion, the most interesting problem which came up certainly was the need to specialize programs at run time. This need arose when we studied the specialization of the Unix file system. The idea was to use specialization values which are only available when a file is opened, that is at run time. There were clear opportunities for specialization but the technology only permitted programs to be specialized at compile time. This situation later led a student and myself to develop a template-based approach to perform specialization at run time.

When I arrived at Irisa in 1993, my immediate goal was to start the design and development of a partial evaluator for C. In fact, it took more than a year of preliminary study to actually launch the project. The design and development of Tempo started in 1994. The group of people working on Tempo, besides myself, then included Luke Hornof, François Noël, Jacques Noyé and Nic Volanschi. Later Gilles Muller and Renaud Marlet joined the team. Gilles Muller brought his expertise in operating systems to make Tempo's features suitable for the needs of systems programs. He lead the work aimed at specializing the Sun RPC. Renaud Marlet brought his expertise in programming languages and software engineering. He studied software architectures in the context of partial evaluation. Besides, he organized and made major contributions to the development of Tempo.

The partial evaluation principles and techniques developed in the context of simple languages like a pure **version** of Scheme represented a valuable basis on which our partial evaluator for C could rely. In fact, our approach aimed at applying the basic design and implementation techniques developed for Schism. It seemed obvious to me that there was nothing in principle which would forbid the re-use of this work. Of course, other aspects were completely unexplored territory. These aspects mainly included the imperative features C (most notably pointers), and the features specific to the systems programs.

The first successful specialization of an existing real-size application occurred in January 1997 with the specialization of the XDR layers of the Sun RPC. Other notable successful applications since then include GAL (a domain-specific language for video device drivers) and, featuring run-time specialization, the specialization of some system layers in the Chorus IPC and the PLAN-P on-the-fly compiler (Active Network language for application protocols).

The first Tempo workshop was organized on March 16-18, 1998. This successful spring school gathered 24 academic and industrial participants.

This workshop was the prelude to Tempo's first release. The original name, which was just *Tempo*, had to be changed to *Tempo Specializer* because of existing registered trade marks.

Tempo Specializer was first released in April 1998 to the participants of the workshop, as well as to other people that had shown earlier interest to our work. A broader distribution was done in June 1998. The binary version of Tempo has been made available without any restrictions as of July 2000.

11.2 Contributions and Contributors

Here is the list of people that had the most important impact on the first versions of Tempo (1994-2000) as well as their main contributions and their position at that time. These lists of people and contributions are incomplete. Also, Tempo actually was the result of many group discussions.

Philippe Boinot, *PhD student*

- Run-time specialization for Linux/Pentium

Sandrine Chirokoff, *PhD student*

- Indirect call elimination
- Data specialization

Charles Consel, *professor* (Tempo visionary and inexhaustible source of inspiration)

- Project leader

- General architecture of Tempo
- Direction and insight provider

Rémi Douence, *post-doc*
- Integration of structure polyvariance (partial)
- Maintenance

Ronan Gaugne, *post-doc*
- Improvements in evaluation-time analysis
- Maintenance

Luke Hornof, *former PhD student* (Btaman)
- Binding-time analysis
- Evaluation-time analysis

Julia Lawall, *post-doc, then visiting associate professor at Oberlin College, research associate at Brandeis University, and then at University of Copenhagen* (sharp as a blade, sweet as marshmallow)
- Side-effect analysis
- Better goto elimination
- Zillions of improvements all over Tempo
- Thorough user (numerical algorithms, graphics, specialization of Java)
- Documentation

Renaud Marlet, *research associate* (dogged as a wart, pitiless for developers)
- Technical supervision
- Abstract models for specialization context and external functions
- Documentation
- Many improvements all over Tempo

Gilles Muller, *research associate* (torrent of ideas, after decryption :-)
- Demanding and leading user (in particular for the RPC specialization)

François Noël, *PhD student*
- Run-time specialization

Jacques Noyé, *research associate* (the Gustave Eiffel of Tempo)
- General architecture of Tempo
- Alias analysis
- Binding-time analysis
- Evaluation-time analysis
- Action analysis

Alan Sayle, *programmer* (the hard-working bug fixer)
- Development and maintenance

Scott Thibault, *PhD student* (can't stop him)
- Goto elimination
- Recursion in compile-time specializer

- Many improvements for the run-time specialization
- Thorough user (GAL, PLAN-P, various interpreters)
- Many other contributions all over Tempo

Eugen-Nicolae Volanschi, a.k.a. Nic, *PhD student*

- Compile-time specializer
- Extensive user (Chorus IPC)

I accept all consequences regarding the side comments above — Renaud.

MAIN TUTOR USER REF INSTALL FAQ LIMIT BUGS SML SUIF DEMO CONTRIB

.....

Table of Contents

Centre de recherche INRIA Bordeaux – Sud Ouest
Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex (France)

Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr

ISSN 0249-0803