A Process and Stream Oriented Debugger for GHC Programs

Munenori MAEDA
International Institute for Advanced Study of
Social Information Science,
FUJITSU LABORATORIES LTD.
Hirotaka UOI Nobuki TOKURA
Faculty of Engineering Science, Osaka University
e-mail: m-maeda@iias.fujitsu.co.jp

Abstract

A new GHC debugger for a typical "Process and Stream" programming paradigm is proposed. When a programmer writes programs following such a paradigm, he has conceptual images about their executions. However the conventional debuggers are inadequate to trace/debug GHC programs because they lack the means to treat both processes and streams directly. Our debugger provides several high-level debugging facilities such as displaying processes connected by streams, editing the data in streams and controlling the execution of processes interactively. As these facilities promote to make programmers trace the execution flow and the data flow from their points of view, it becomes easier to debug programs following process and stream paradigm.

1 Introduction

The GHC debuggers, which have been proposed so far, are classified into two types. One is based on algorithmic debugging[6], and the other on execution tracing[1]. The former is based on the denotational semantics of GHC programs and can prune lots of wasteful information semi-automatically, while the latter, which based on the operational semantics, provides several manual means to prune it. The common philosophy between them is that "Too much of displaying low-level information is as bad as too little. Time consumption for debugging depends on an ability of the debugger as an information filter." This paper aims at proposing a new debugging method belonging to the execution tracing class.

In GHC programming, object-oriented[5] and stream-based[2] programming, which are quite familiar and popular, are centered on the notion of processes and streams. Each abstract programming module is regarded as a process, some of these connected by streams, and communicate with each other concurrently. A typical process repeats the following: it consumes data from an input stream, changes its own internal state, and produces data to an output stream. The GHC programs based on such processes and streams are often called as process oriented programs.

It is difficult to capture a conceptual execution image of a process oriented program when a conventional execution tracer is used. It is because processes and streams generated at runtime are decomposed into GHC primitives, and they are never displayed explicitly.

This paper suggests a tracer that fully reflects the notion of process and enables one to handle both of the specific control flow of processes and the data structure of streams. This tracer is expected as an information filter to makes the causality among processes explicit.

The organization of the paper is as follows: first, in Section 2, we will introduce the model of processes and streams. Additionally some operations and an equivalence relation on streams will be shown. Section 3 will offer a debugging method

[1]The literature[1] is concerned with only the execution tracing of Occum programs, however, its discussion is generally adaptable for the most concurrent/parallel programs debugging.

for the process oriented programs. By considering this method, we will be able to specify some facilities of the tuned debugger. The topics of an implementation of the debugger will be discussed in Section 4, which are how processes and streams can be identified and how processes can be controlled. One of the central points to evaluate the debugger is an artifice for better readability of traces. In Section 5, two visualization techniques adopted by the debugger will be described. A tracing example will be shown in Section 6. Finally, in Section 7, conclusions and future work will be presented.

2 Models of Processes and Streams in GHC

2.1 Process model

Two interpretations of a "process" are familiar. One is to regard a goal as a process. The other is to regard an "object"[5], in other words, a computational entity such as a light weight process of UNIX. In the following, we discuss about processes based on the latter.

A process consists of goals of two types. One type of goal is for the continuation of this process, while the other is for the internal procedures defined in the process. The continuation goal accepts streams in its arguments one by one, and reserves its internal state in the other arguments. The stream argument works as an I/O port of the process. The internal state is not affected by other processes, but only (re)calculated by the previous state and input data captured from streams.

The features of a process are listed up as follows:

Creation: A process is created by the first call of the continuation goal.

One-step execution: Reading data from streams, writing data to other streams, and changing the internal state by using internal procedures are regarded as atomic actions in an execution step.

Continuation and Termination: A process will carry on its computation with a new internal state when the continuation goal is invoked. Otherwise the

process terminates its execution.

2.2 Stream model

A stream is a sequence of any logical terms, whose operations[8] are limited to reading the first term of a stream and writing a term on the tail of a stream.

Streams are constructed by stream-variables **SV**, stream-functors $\langle \mathbf{SH} \| \mathbf{ST} \rangle$ and stream-terminators $\langle\rangle$, where SV is a variable constrained to become either a stream-functor or a stream-terminator, SH is an arbitrary term that denotes the first data of the stream, and ST is a stream representing the rest of the stream.

Creation: Streams are created dynamically when a continuation goal of a process is invoked, where they are assigned to the arguments of the goal.

Data access: The first data D is read from a stream SX by unifying SX with a structure $\langle \mathbf{D} \| \mathbf{ST} \rangle$ in the guard part of a clause at runtime. The data D is written to a stream-variable SX by unifying SX with a structure $\langle \mathbf{D} \| \mathbf{ST} \rangle$ in the body part, where ST, called the tail of the stream SX, is a stream-variable. In the case of reading or writing several times, each operation is done recursively for the rest of the stream, ST.

Connection: Two streams $\mathbf{S_a}, \mathbf{S_b}$ are connected if they are unified in the body part. One of the connected streams are regarded as an alias of the other.

We define an equivalence relation $\cong$ on the set of streams $\mathcal{S}$, which is used for the visualization of streams.

For a substitution σ, the following relation $\simeq_\sigma$ is defined on $\mathcal{S}$, which is the set of streams consisting of variables, terms obtained under the execution.

1. $\mathrm{S} \simeq_\sigma \mathrm{S};\ ^\forall \mathrm{S} \in \mathcal{S}$
2. $\langle \mathrm{H} \| \mathrm{S} \rangle \simeq_\sigma \mathrm{S};\ ^\forall \mathrm{S} \in \mathcal{S}$
3. $\mathrm{S}_1\sigma = \mathrm{S}_2\sigma \Rightarrow \mathrm{S}_1 \simeq_\sigma \mathrm{S}_2, ^\forall \mathrm{S}_1, \mathrm{S}_2 \in \mathcal{S}$

The first reflective rule implies that two lexically identical variables satisfy the relation. The second rule implies that a stream and its subpart are elements of the same equivalence class. The third means that the connected streams are also elements of the same equivalence class. The relation $\cong_\sigma$ is defined as the symmetric and transitive closure of the relation $\simeq_\sigma$. In the following, we write $\cong$ for $\cong_\sigma$ if σ is clearly understood from the context.

In GHC, a stream is actually realized by a list in most programs, i.e. a stream-functor $\langle \mathrm{D} \| \mathrm{S} \rangle$ and a stream-terminator $\langle\rangle$ correspond to a term $[\mathrm{D} \mid \mathrm{S}']$ and an atom $[]$. We commit this observation, and implement a debugger presented in Section 4.

3 Process oriented debugging

When following the above process model, each goal appearing in the execution trace exactly belongs to a process, which is either a continuation or a part of a internal procedure of the process. When we trace and check process oriented programs, first of all we have to do is to extract the goals belonging to a target process from the "chaotic" execution trace where these goals are interleaved.

The check of the data flow is also required. Unless a process inputs intended data, the process outputs incorrect data to its output stream, or permanently suspends. Two reasons why the intended data is not sent to the process are considered. One is that an adjacent process corresponding to the producer of the data works in malfunction. The other is that the input of the process maintains disconnection with the output of the producer, which is a failure by misuse of a shared variable. In the latter case, it is easier to detect the error if the connection between processes by streams, which is called "a process network", is displayed.

Thus for better readability of the execution traces of the process oriented programs, Process Oriented Debugger(**POD**) visualizes the information of processes and streams in a structured way, which consists of the input/output data, the internal state values, and the internal procedure traces, and the connection between processes by streams.

Then programs can be debugged in the following steps.

Step 1. A user starts to execute a target program.

Step 2. The internal state and the input/output data are displayed and investigated at an appropriate interval. The process network is also checked.

Step 3. The program code corresponding to the process that raises an error is investigated in detail. And any other neighborhood processes that may cause it to get into anomaly should be observed as well.

Step 4. The sequences of input/output data are available for checking an abnormal process by preserving them in the previous execution. Basically, comparing the sequence of the output data before and after the modification of a program makes it easier to check the behavior.

Here we consider **Step 3** and **Step 4**. When the process is in malfunction, we compulsorily suspend its continuation, and proceed to the total execution as far as possible, because the re-execution of the program costs much time and care. Namely, it is important to avoid the re-execution when it takes a long time to instantiate the data on streams up to an enough length, or the program has nondeterministic transitions.

Re-execution can be avoided in two ways: one is to give the debugger the functions to delete or to modify the unexpected data and to insert data in the stream interactively. The other is to make the functions preserve the data in streams automatically and execute a process under the environment made by the preserved ones.

Thus the execution control functions to be requested to POD is

1. to compulsorily suspend, resume and abort the execution of each process,
2. to buffer and to modify the data on streams interactively, and
3. to execute a process with the environment.

```
(1) process gen(state,state,port),sift(port,port),filter(port,state,port).
(2) prime(Max,Ps):- true |  gen(2,Max,Ns),sift(Ns,Ps).
(3) gen(N,Max,Ns):- N>=Max | Ns =[].
(4) gen(N,Max,Ns):- N<Max, N1:=N+1 | Ns=[N|Ns1],@gen(N1,Max,Ns1).
(5) sift([],Ps):- true | Ps=[].
(6) sift([P|Fs],Ps):- true | Ps=[P|Ps1],filter(Fs,P,Fs1),@sift(Fs1,Ps1).
(7) filter([],P,Fs):- true | Fs=[].
(8) filter([N|Ns],P,Fs):- true | sw(N,P,Fs1,[N|Fs1],Fs), @filter(Ns,P,Fs1).
(9) sw(N,P,Fs1,Fs2,Fs):- N mod P=:=0 | Fs=Fs1.
(10)sw(N,P,Fs1,Fs2,Fs):- N mod P=\=0 | Fs=Fs2.
```

List 1: Primes generator program with process declaration

4 Implementation of POD

4.1 Process declaration

In our process model presented in Section 2.1, goals are classified into goals for the continuation and those for the internal procedures. As there is no syntactic difference between them, they should be specified by means of the process declaration by the user.

The process declaration consists of the predicate specification and the continuation marking. The predicate specification begins with the keyword `process` followed by the name of the predicate it specifies the usage of each argument. The usage of each argument is specified by declaring either `state` or `port` in an appropriate order. The keyword `state` denotes that the argument represents a part of the internal state, while `port` means that the argument represents an I/O port of the process. The continuation marking represents a continuation, which is done by annotating `@` in front of a goal in a clause.

See the "Primes generator" program in List 1 where some annotations for the process declaration are included. This program will be executed and traced in Section 6.

4.2 Treatment of streams

Recall that streams are constructed by special variables, functors and terminators.

In POD, streams are recognized and supported by introducing the tagged data

structures. Each variable, functor, and atom that realizes a stream has an auxiliary field to store the identifier of the stream. An identifier is associated with each equivalence class of the streams.

Now the problem is how to implement the identifiers. Note that if two streams whose identifier are different each other are unified, their identifiers should be observed as the same one. This can be naturally done by assigning a variable to each identifier and unifying the identifiers when their streams are unified. The problem whether two streams satisfy the equivalence relation $\cong$ is solved as the "variable equivalence check."

Our implementation of streams is as follows: the logical terms; 'Sm'(**Var**,**ID**), 'Sm'([],**ID**) and 'Sm'([**Head**|**Tail**],**ID**) are used for representing a stream-variable **Var**, a stream-terminator [] and a stream-functor [**Head**|**Tail**] respectively, while the second argument **ID** denotes the identifier of its stream, and is provided as a fresh variable.

POD recognizes and manages streams in the following way. First if the goal to be reduced is declared as a process, POD replaces the arguments whose modes are `port` by the stream-tagged fresh variables. Then the original arguments and the stream-tagged variables are unified by using the extended unification procedure shown in Appendix **A**. The unification procedure can treat tagged data and manipulate streams in the same manner in Section 2.2.

The stream-tagged variables and other arguments whose mode is `state` are stored in a process table. A process table is prepared for each process, which preserve a couple of arguments every execution step. The visualization of the execution for processes is achieved by using the process tables.

4.3 Execution Control

In POD, the specific control of a process proposed at Section 3 is realized by introducing the "valve" inserted into a stream(Figure1). The valve takes the role of an intelligent data buffer, which has two input ports, one output port, and a programmable conditional switch to close the output port. One of the two input

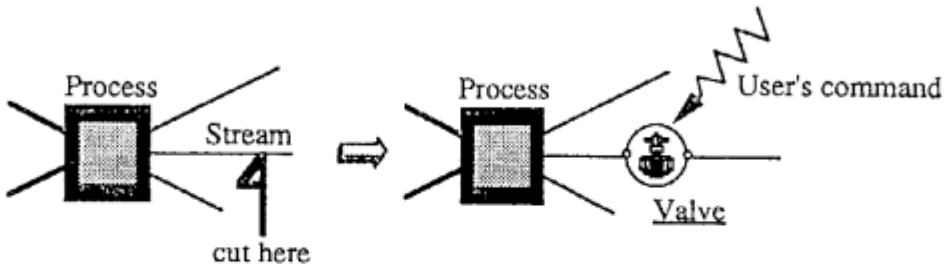

Figure 1: Inserting a valve between adjacent processes

ports is connected to the original stream, and the other is connected to a user's console. A user can send the commands to the valve. The programmable conditions are associated with the amount of buffered data and the description of a type of storable data.

The valve has three states, and each state is changed to another by the user's command or evaluating the programmable conditions. The three states are named as the automatic migration mode, the conditional migration mode, and the manual edit mode. In each mode, the valve behaves as follows.

- Automatic migration mode

  The valve gets data from its input port then it stores in own buffer. If the buffer becomes full, the valve outputs the first data in the buffer from the output port.

- Conditional migration mode

  The valve gets data then it stores in the buffer. If the buffer becomes full or the data does not satisfy the condition then it displays an alert and changes its mode to the manual edit mode.

- Manual edit mode

  In this mode, the valve never gets new data. The conditions for data to be stored and the actual data to have already stored can be referred and modified by using a text editor. After finishing editing, the mode is changed and goes back to the previous mode.

The data checking condition is provided as the conjunction of GHC goals. The goal is one of the built-in predicates or an user-defined goal. The built-in predicates are classified into the type check, the arithmetic comparison and the guard unification $=/2$. The type check goal is like $atom/1, integer/1, list/1$, etc., the arithmetic one is like $>/2$, $\geq/2$, $</2$, $\leq/2$. The user-defined goal is a combination of built-in goals which follows to GHC(Prolog) like syntax.

5 Visualization of the execution

POD provides two different views, which are called as "stream graph" and "process chart", for the visualization of the program execution.

5.1 Stream graph

A stream graph shows the dynamic change of a network graph concerning processes and streams by the animated icons and lines respectively.

The rule of drawing the stream graph is given as follows.

1. An icon representing a process is newly displayed in a graphical window when the process goal is invoked, which is painted by a specific color that represents the activity of the process. No lines are connected to the icon, which corresponds to that the process has no communication channels yet.

2. As soon as two stream-tagged terms which have different identifiers are unified, all I/O ports which has been connected to those streams already are re-linked to each other.

3. All lines connected to a process icon are erased when a continuation goal of the process is invoked.

4. The process icon is re-painted with a different specific color which represents termination of it when no continuation goal is invoked.

5.2 Process chart

We show an effective display technique for streams. The following observation is acceptable.

- As a stream is a variable length and unbounded in nature, the whole sequence of the stream can not be displayed. It is enough to display only a subsequence generated or consumed newly.

For instance, when a stream S is unified with $[\mathbf{D_a}, \mathbf{D_b} \mid \mathbf{T}]$ in the guard part of a clause and T is referred in its body, only the different sequence between two streams S and T, i.e. $[\mathbf{D_a}, \mathbf{D_b}]$, is displayed in Process Oriented Debugging.

Here a new graphical view, called as a "process chart", is proposed, which consists of many dots and lines. A dot corresponds to an argument which is obtained from each execution step of a process. A line connects two dots $\mathbf{P_a}$ and $\mathbf{P_b}$ when $\mathbf{P_a}/\mathbf{P_b}$ corresponds to a stream $\mathbf{S_a}/\mathbf{S_b}$ respectively, and one of the following two relations is satisfied.

- The "identical" relation $\mathbf{P_a} \rightsquigarrow \mathbf{P_b}$ denotes that all the elements of $\mathbf{S_a}$ and $\mathbf{S_b}$ are equal.
- The "adjacent" relation $\mathbf{P_a} \rightarrow \mathbf{P_b}$ denotes that $\mathbf{S_a}$ is the longest tail of $\mathbf{S_b}$.

The definitions of them are presented in Appendix B.

For convenience of the following description, we give the informal explanations of terms: $\overline{\mathcal{PS}}$, $\mathcal{OA}(\overline{\mathcal{PS}})$ and $\mathcal{RT}(\overline{\mathcal{PS}})$, which are formalized in Appendix B. $\overline{\mathcal{PS}}$ is defined as a set of all the streams with their corresponding positions of dots. $\mathcal{OA}(\overline{\mathcal{PS}})$ is a set of relations defined under $\overline{\mathcal{PS}}$: $\rightsquigarrow$ on $\mathcal{P} \times \mathcal{P}$ and $\xrightarrow{I}$ on $\mathcal{P} \times \mathcal{T} \times \mathcal{P}$ where $\mathcal{P}$ is a set of positions of dots and $\mathcal{T}$ is a set of text. $\mathcal{RT}(\overline{\mathcal{PS}})$, called as all the "root" nodes of $\overline{\mathcal{PS}}$, is a set of positions which appear only the first arguments of the above relations, i.e. is used as start positions.

A process chart is drawn by the following steps.

Step 1. Depict N dots rightward which is aligned horizontally with a left margin L and an appropriate gap H, where the dots represent all arguments of a process

with an arity N. Then connect them by a thin dashed line to emphasize that they are arguments in an execution step of a process. A dot is painted with black or gray when the corresponding argument is available in input or output mode.

Step 2. Apply **Step 1** downward with an appropriate vertical gap V to each execution step up to the current execution. If a subprocess is forked among the i-th and the $i+1$-th steps, depict the dots of the subprocess's in a similar way with a left margin Ld, Ld > L.

Step 3. Let a stream S be assigned to a j-th argument declared as `port` of i-th step of a process PID. Then $loc(PID, i, j)$, the location of the stream, could be mapped to the position of the dot one to one in the plane. A binary tuple $(loc(PID, i, j), S)$ is called a "stream with its location", which corresponds to the dot. For all the dots depicted in **Step 1** and **2**, make a set $\overline{\mathcal{PS}}$ of "streams with their locations", then compute and make an output set $\mathcal{OA}(\overline{\mathcal{PS}})$.

Step 4. Connect two dots P_a and P_b with a line if $P_a \stackrel{l_{a,b}}{\rightarrow} P_b$ exists in $\mathcal{OA}(\overline{\mathcal{PS}})$, and write a text $l_{a,b}$ along the line. Similarly connect two dots with a double line if $P_a \rightsquigarrow P_b$ exists in $\mathcal{OA}(\overline{\mathcal{PS}})$. In both cases, a line is painted with the color of the dots of both ends.

Step 5. Write the values of all the arguments declared as `state` under their dots, and also write the values of all the members of $\mathcal{RT}(\overline{\mathcal{PS}})$ under their dots.

6 A tracing example

POD is developed by extending GHC compiler[3, 10] on Macintosh. A user can trace and debug a GHC program with a direct manipulation interface of **POD**.

The interface provides several control facilities for the target program in a menu, so the user can easily manipulate **POD** by selecting one in the menu with mouse. Currently the facilities, (1) the compulsive suspension and (2) the resumption of

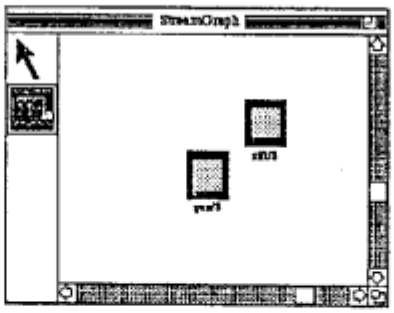

Figure 2: Initial view of stream graph

processes, (3) the insertion and (4) the control of valves, and (5) the deletion of terminated processes are prepared in the menu.

The usages of the menu and the views are described using the program in List List 1. First suppose that the program and a query goal `prime(10,Ps)` are given to **POD**. Figure 2 shows the initial stream graph. Here the user is able to investigate the data on a stream that connects `gen` and `sift` in two ways. One is to stop `gen` in order to prevent the creation of new data and to set the valve behind the output port of `gen`. The other is to stop `sift` in order to avoid the consumption of data and to set the valve in front of the input port of `sift`. Let the former be chosen. He selects the item (1) and makes `gen` suspended. Selecting item (3) followed by item (2) resumes `gen`. The generated valve is displayed as an icon in the window similarly as that of a process. Initially the valve is in automatic migration mode and the default size of the buffer is set to one hundred.

After the process `gen` finishes generating data, the information in the valve is displayed in a new window if he selects the item (4). Figure 3 shows that the content of the buffer is modified by deleting the number 8. Now suppose that he closes the window and flushes all data in the buffer.

By flushing the data, the execution of `sift` is resumed spontaneously, and finally the stream graph becomes stable, which is depicted in Figure 4.

Now process charts of each processes in a window are shown in Figure 5, which explains explicitly the following.

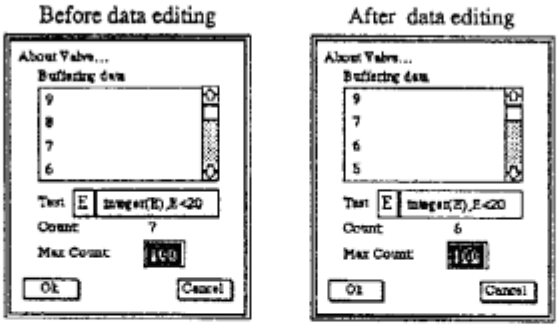

Figure 3: Display of valve controller

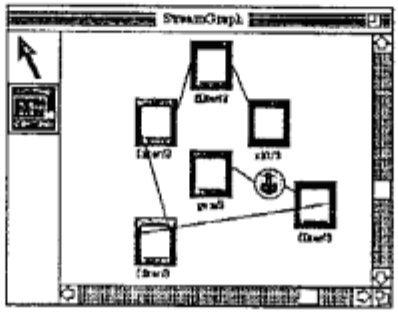

Figure 4: Final view of stream graph

- The process `gen` maintains an output stream that is specified by a vertical gray line in the lefthand side of the window, which connects all of the third arguments obtained every execution step. Numbers generated by `gen` are aligned and displayed along the line.

- The process `filter` maintains both an input and an output stream specified by two vertical lines; the black and the gray in the middle of the window. The input sequence of numbers on the black line ranges from 3 to 9 except 8 because 8 is deleted in the valve. It is possible to comprehend that `filter` generates a number or none every execution step when the output sequence on the gray line is referred.

- The difference between the process chart of `sift` and others is the presence of a process fork specified by a dashed line. The process `sift` also has both an input stream and an output stream. The output stream remains unchanged, while the input stream is dynamically created. `sift` consumes a number from the input stream in the first argument, and it generates a prime number and a `filter` by an execution step, which is passed to the stream in the second argument. The input and output stream of the created `filter` are connected to the original input and the new input stream of `sift` respectively.

7 Conclusions and future work

This paper has proposed a process oriented debugger, POD, for GHC programs which are based on a computation model of processes and streams. In this paper, we have argued the model of processes and streams, the debugging method for programs based on processes and streams and the facilities of the debugger. The equivalence relation on streams is the bases of the visualization, which depicts whether two streams are connected or not.

The features of POD are summarized as follows:

- showing the interprocess causality by lines and icons in a stream graph, where lines and icons represent streams and processes respectively,
- presenting the connectivity relationship between streams by lines in a process chart, where lines represent the difference parts of two related streams,
- inserting a valve into the front/behind of a process, where the purpose of a valve are
  - avoiding the re-execution of the process by storing input data,
  - validating the process modification by comparing its output data before and after the source code changes and

- controlling the execution interactively by changing input data or blocking data to output.

Thus our debugger could handle both of the specific control flow of processes and the data structure of streams.

The future work is further improvement of the graphical view, stream graph and process chart.

Stream graph: All generated processes are located in the graphic window freely, which causes less readability in the case of more large scale programs. An approach is to adopt an appropriate drawing rule for drawing processes and streams. To make such a rule be programmable is interesting.

Process chart: A behavior of a process which generates subprocesses, such as a process `sift`, is not self-evident, so sometimes their process charts should be referred simultaneously. The readability becomes worse even if multi windows are available. An approach to display many process charts is to adopt a perspective view such as Figure 6. A process chart of a subprocess is connected to the corresponding process-fork point in the parent process chart. By combining these charts, It will be possible to get various images by changing view points, i.e. selecting the direction and zooming/unzooming the point of the chart.

Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project of Japan. The authors would like to express thanks to Dongwook Shin, Youji Kohda and Jiro Tanaka for their help and useful comments. Masaki Murakami and Hiroyasu Sugano indebted to construct the matematical formalization. The authors are also appreciative of their assistance.

Appendix A. Extended Unifier for Streams

The terms occurring at runtime are classified into six types; variable, atom, compound term, stream-variable, stream-functor, and stream-terminator.

Here we show the following central cases of the extended unification $X \asymp Y$.

Case 1 X is a stream variable represented as 'Sm'(V,ID), Y is a variable.
Then assign Y to X.

Case 2.1 X is a stream variable 'Sm'(V,ID), Y is an atom [].
Then assign [] to V.

Case 2.2 X is a stream variable, Y is an atom except [].
Then assign Y to V, and display a warning dialog.

Case 3.1 X is a stream variable 'Sm'(V_1,ID), Y is a list [H|T].
Then assign [H|'Sm'(V_2,ID)] to V_1, and execute 'Sm'(V_2,ID)$\asymp$T.

Case 3.2 X is a stream variable, Y is a compound term except list.
Then assign Y to V_1, and display a warning dialog.

Case 4 X is a stream variable 'Sm'(V_1,ID_1), Y is a stream variable 'Sm'(V_2,ID_2).
Then assign V_2 and ID_2 to V_1 and ID_1 respectively.

Case 5 X is a stream-functor 'Sm'([H_1|T_1],ID), Y is a list [H_2|T_2].
Then unify H_2 with H_1, and execute $T_1 \asymp T_2$ recursively.

Case 6 X is a stream-functor 'Sm'([H_1|T_1],ID_1), Y is a stream-functor 'Sm'([H_2|T_2],ID_2)
Then unify H_2, T_2, ID_2 with H_1, T_1, ID_1 respectively and
execute $T_1 \asymp T_2$ recursively. □

A case given two stream-terminators as an input, and a case between a stream-terminators and a nil list are omitted because it is analogized easily.

The unifier displays a warning dialog as soon as detecting a failure of the unification. At this time a user can choose two alternatives; terminating the total execution or ignoring this failure to continue the execution.

Appendix B. Arrow relations on streams

Let $\mathcal{P}$ be the totally ordered set of position identifiers, $\mathcal{S}$ be the set of streams. An element $PS = (P, S)$ of the set $\mathcal{P} \times \mathcal{S}$ is called a "stream with its location", and we use a notation $\overline{\mathcal{PS}}$ for a finite set of "streams with their locations". We also define two unary functions, f and t, on $\mathcal{P} \times \mathcal{S}$ as follows; for $PS = (P, S) \in \mathcal{P} \times \mathcal{S}$, $h(PS) = P$ and $t(PS) = S$. And let a function c be a type converter from stream to list.

We extend the equivalence relation $\cong$ on $\mathcal{S}$ defined before to $\simeq$ on $\mathcal{P} \times \mathcal{S}$; $(P_a, S_a) \simeq (P_b, S_b)$ iff $S_a \cong S_b$. Since the relation $\simeq$ is clearly an equivalence relation, we can get the quotient set $\overline{\overline{\mathcal{PS}}}$ of $\overline{\mathcal{PS}}$ divided by the relation $\simeq$ where $\overline{\mathcal{PS}}$ is a finite set of "streams with their locations"; $\overline{\overline{\mathcal{PS}}} = \{\overline{\mathcal{PS}_0}, \cdots, \overline{\mathcal{PS}_m}\} \stackrel{def}{=} \overline{\mathcal{PS}}/\simeq$.

Definition 1 (Identical arrows) For two position identifiers P_a, P_b, we introduce a notation $P_a \rightsquigarrow P_b$ to indicate that the content of the stream assigned on P_a is equal to that of the stream on P_b, and P_a appears just before P_b. The set of identical arrows $\mathcal{IA}(\chi)$ for $\chi \in \overline{\overline{\mathcal{PS}}}$ is defined as follows; $\mathcal{IA}(\chi) = \{h(PS_a) \rightsquigarrow h(PS_b) \mid PS_a, PS_b \in \chi, c(t(PS_a)) = c(t(PS_b)), h(PS_a) < h(PS_b), \neg\exists PS_c \in \chi, h(PS_a) < h(PS_c) < h(PS_b), c(t(PS_b)) = c(t(PS_c))\}$. □

Definition 2 (Extermination of identical streams) First we define an equivalence relation $\doteq$ for $PS_a, PS_b \in \chi$ as follows; $PS_a \doteq PS_b \stackrel{def}{=} c(t(PS_a)) = c(t(PS_b))$. Then for χ, $\overline{\chi}$ is defined by collecting all representatives of χ; $\overline{\chi} = \{PS \in \chi \mid \forall PS' \in [PS] \in \chi/\doteq, h(PS') \leq h(PS)\}$ where $[PS]$ is an equivalence class in $\chi/\doteq$ containing PS. □

Here we define a set $D_{X,Y}$ for two streams X, Y such that $X \cong Y \wedge c(X) \neq c(Y)$ as follows; $D_{X,Y} = \{(U, V, W) \mid append(U, W, c(X)) \wedge append(V, W, c(Y))\}$. Then there obviously exist the unique $U@_{X,Y}, V@_{X,Y}, W@_{X,Y}$ which satisfy $(U@_{X,Y}, V@_{X,Y}, W@_{X,Y}) \in D_{X,Y} \wedge \forall (U_i, V_i, W_i) \in D_{X,Y} \wedge len(W@_{X,Y}) \geq len(W_i)$ where for $\forall L \in List$, $len(L)$ returns the number of elements in a list L. In the following, we abbreviate $U@_{X,Y}, V@_{X,Y}, W@_X$ as $U@, V@, W@$ if X, Y are clearly understood from the context.

Definition 3 (Streams and all their bifurcations) We can construct $\mathcal{C}_{\overline{\chi}}$ with $\overline{\chi}$ such that there exists a "stream with its location" in $\mathcal{C}_{\overline{\chi}}$ for two streams in $\overline{\chi}$ where such a stream is shared as their common tails. The set $\mathcal{C}_{\overline{\chi}}$ is a minimum set satisfying the following condition.

1. $\overline{\chi} \subseteq \mathcal{C}_{\overline{\chi}}$.
2. $\forall PS_a, PS_b \in \mathcal{C}_{\overline{\chi}}, \exists PS_c \in \mathcal{C}_{\overline{\chi}}, W@_{t(PS_a),t(PS_b)} = c(t(PS_c))$
   if $U@_{t(PS_a),t(PS_b)} = [], V@_{t(PS_a),t(PS_b)} = []$.
3. $\forall PS_a, PS_b \in \mathcal{C}_{\overline{\chi}}, h(PS_a) \neq h(PS_b)$.

Definition 4 (Reachable arrows) For two position identifiers P_a, P_b, and for a list labeled $l_{a,b}$, a notation $P_a \overset{l_{a,b}}{\Rightarrow} P_b$ introduced newly is used to say that P_a and P_b are connected by an reachable arrow, which implies that a list $l_{a,b}$ append a stream identified by P_a becomes the other stream identified by P_b. The set of reachable arrows $\mathcal{RA}(\mathcal{C}_{\overline{\chi}})$ for $\chi \in \overline{\overline{\mathcal{PS}}}$ is defined as follows; for $PS_a, PS_b \in \mathcal{C}_{\overline{\chi}}$, compute $U@_{t(PS_a),t(PS_b)}, V@_{t(PS_a),t(PS_b)}, W@_{t(PS_a),t(PS_b)}$ then classify by their values.

Case 1. $U@ = []$. Then $ra(PS_a, PS_b) = \{h(PS_a) \overset{V@}{\Rightarrow} h(PS_b)\}$.

Case 2. $V@ = []$. Then $ra(PS_a, PS_b) = \{h(PS_b) \overset{U@}{\Rightarrow} h(PS_a)\}$.

Case 3. $U@ \neq [] \wedge V@ \neq []$. Then $ra(PS_a, PS_b) = \{h(PS_c) \overset{U@}{\Rightarrow} h(PS_a), h(PS_c) \overset{V@}{\Rightarrow} h(PS_b) \mid \exists PS_c \in \mathcal{C}_{\overline{\chi}}, \rfloor(\sqcup(\mathcal{PS}_\rfloor)) = \mathcal{W}@\}$.

Here a set of the reachable arrows $\mathcal{RA}(\chi)$ is defined by using the above auxiliary function ra as follows;

$$\mathcal{RA}(\chi) = \bigcup_{\forall PS_a, PS_b \in \mathcal{C}_{\overline{\chi}}} ra(PS_a, PS_b).$$

Definition 5 (Adjacent arrows) A notation $P_a \overset{l_{a,b}}{\rightarrow} P_b$ for $P_a, P_b \in \mathcal{P}$ is introduced newly, which is used to say that P_a and P_b is connected by an adjacent arrows, or P_a is reachable to P_b through the shortest distance $l_{a,b}$. First we define an auxiliary set $G_i(\xi), 0 \leq i \leq n$, for a set of reachable arrows, ξ, in an inductive way. These sets correspond to a notion of "reduced reachable arrows".

Base case. $G_0(\xi) = \xi$.

Induction step. $G_{i+1}(\xi) = G_i(\xi) - (R_a \overset{l_{a,c}}{\rightarrow} R_c)$ where $R_a \overset{l_{a,c}}{\rightarrow} R_c \in G_i(\xi) \wedge R_a \overset{l_{a,b}}{\rightarrow} R_b, R_b \overset{l_{b,c}}{\rightarrow} R_c \in \xi$.

The set $G_i(\xi)$ is monotonically decreased, we finally get an irreducible set $G_n(\xi)$ on finite reduction steps.

Then the set of adjacent arrows is defined by replacing the symbol $\Rightarrow$ on G_n to the $\rightarrow$, i.e., $\forall R_a \overset{l_{a,b}}{\Rightarrow} R_b \in G_n, \exists R_a \overset{l_{a,b}}{\rightarrow} R_b \in \mathcal{AA}(\xi)$. □

Definition 6 (Generation of an output set) An output set $\mathcal{OA}(\overline{\mathcal{PS}})$ to represent relationship between "streams with their locations" is defined as follows; $\mathcal{OA}(\overline{\mathcal{PS}}) = \mathcal{IA}(\overline{\mathcal{PS}_0}) \cup \cdots \cup \mathcal{IA}(\overline{\mathcal{PS}_m}) \cup \mathcal{AA}(\mathcal{RA}(\overline{\mathcal{PS}_0})) \cup \cdots \cup \mathcal{AA}(\mathcal{RA}(\overline{\mathcal{PS}_m}))$ where $0 \leq i \leq m, \overline{\mathcal{PS}_i} \in \overline{\overline{\mathcal{PS}}}$. □

Definition 7 (A set of root nodes) For $\forall P_a \rightsquigarrow P_b \in \mathcal{IA}(\chi)$, if $\neg\exists P_c \rightsquigarrow P_a \in \mathcal{IA}(\chi)$ then P_a is called "root" of $\mathcal{IA}(\chi)$. Similarly the root P'_a of the set $\mathcal{AA}(\chi)$ can be defined as follows: for $\forall P'_a \rightarrow P'_b \in \mathcal{AA}(\chi)$, $\neg\exists P'_c \rightarrow P'_a \in \mathcal{AA}(\chi)$.

A set of root nodes $\mathcal{RT}(\overline{\mathcal{PS}})$ is defined as a set of all root nodes of $\mathcal{IA}(\chi)$ and $\mathcal{AA}(\chi)$, where χ is ranged from $\overline{\mathcal{PS}_0}$ to $\overline{\mathcal{PS}_m}$. □

References

[1] G.S.Goldszmidt, S.Yemini, S.Katz: "High-level Language Debugging for Concurrent Programs", ACM Transactions on Computer Systems, Vol.8, No.4, pp.311-336, Nov. 1990.

[2] G.Kahn, D.B.MacQueen: "Coroutines and Networks of Parallel Processes", Information Processing 77, North-Holland, pp.993-998, 1977.

[3] Y.Kohda, J.Tanaka: "Deriving a Compilation Method for Parallel Logic Languages", Logic Programming '87(LNCS 315), Springer-Verlag, pp.80-94, 1988.

[4] M.Maeda, H.Uoi, N.Tokura: "Process and Stream Oriented Debugger for GHC programs", Proceedings of Logic Programming Conference 1990, pp.169-178, ICOT, Jul. 1990.

[5] E.Shapiro, A.Takeuchi: "Object Oriented Programming in Concurrent Prolog", New Generation Computing, Vol.1, No.1, pp.25-48, 1983.

[6] A.Takeuchi:"Algorithmic Debugging of GHC programs and its Implementation in GHC", ICOT Tech. Rep. TR-185, ICOT, 1986.

[7] J.Tanaka:"Meta-interpreters and Reflective Operations in GHC", Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.774-783, ICOT, Nov. 1988.

[8] E.D.Tribble, M.S.Miller, K.Kahn, D.G.Bobrow, C.Abbot and E.Shapiro: "Channels: A Generalization of Streams", Proc. of 4th International Conference of Logic Programming(ICLP)'87 Vol.2, pp.839-857 (1987).

[9] K.Ueda:"Guarded Horn Clauses", ICOT Tech. Rep. TR-103, pp.1-12 (1985-06).

[10] K.Ueda, T.Chikayama: "Concurrent Prolog Compiler on Top of Prolog", in Proc. of Symp. on Logic Prog., pp. 119-126, 1985.

[11] K.Ueda, T.Chikayama: "Design of the Kernel Language for the Parallel Inference Machine", The Computer Journal, 1990.

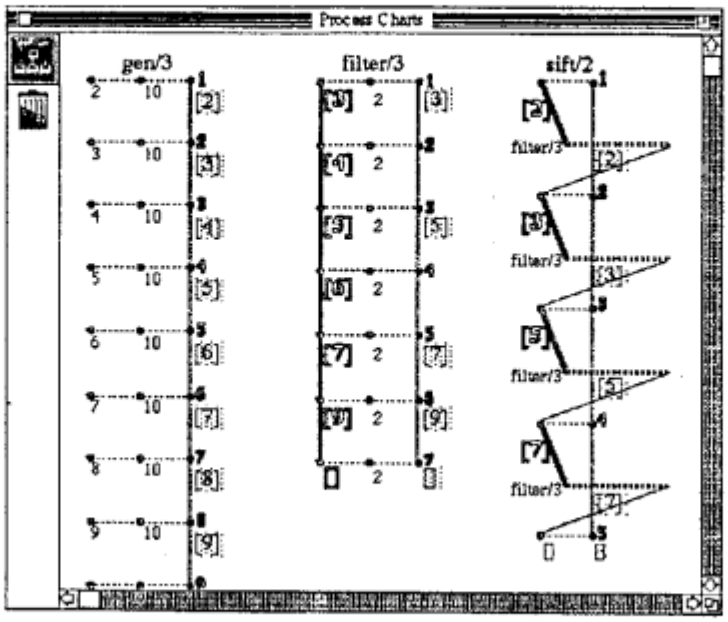

Figure 5: Display of process charts

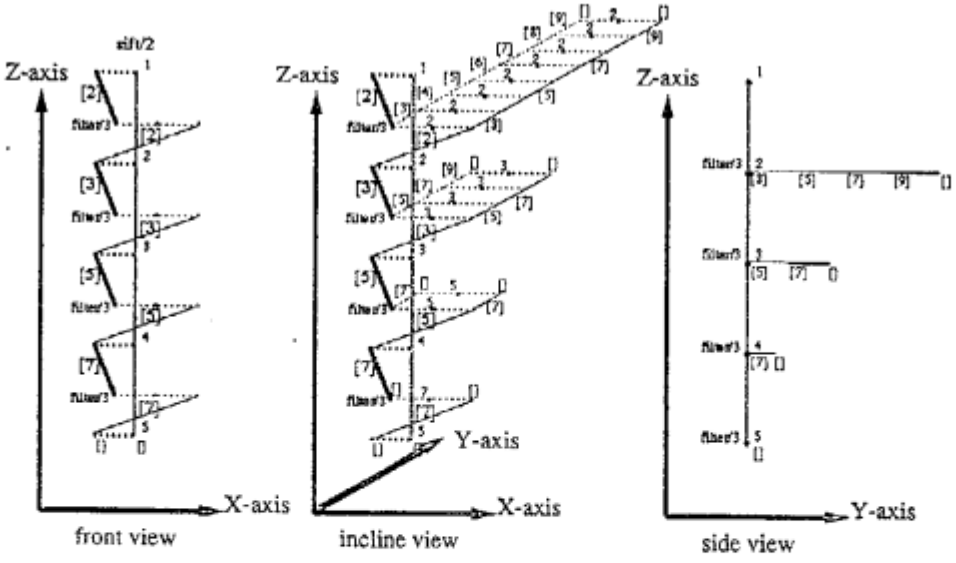

Figure 6: Image of an extended three-dimensional process chart