PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Implementing a Process Oriented Debugger with Reflection and Program Transformation

Munenori MAEDA
International Institute for Advanced Study of Social Information Science,
FUJITSU LABORATORIES LTD.
17-25, Shinkamata 1-chome, Ota-ku, Tokyo 144, Japan

m-maeda@iias.flab.fujitsu.co.jp

Abstract

Programmers writing programs following a typical process and streams paradigm usually have some conceptual image concerning the program's execution. Conventional debuggers cannot trace or debug such programs because they are unable to treat both processes and streams directly. The process oriented GHC debugger we propose provides high-level facilities, such as displaying processes and streams in three views and controlling a process's behavior by interactively blocking or editing data in its input streams. These facilities make it possible to trace and check program execution from a programmer's point of view. We implement the debugger by adopting reflection and program transformation to enhance standard GHC execution and to treat extended logical terms representing streams.

1 Introduction

Debugging methods for programs in Guarded Horn Clauses(GHC)[Ueda 1985] are classified into those based on algorithmic debugging[Takeuchi 1986] under the denotational semantics of GHC programs, and those based on execution tracing [Goldszmidt *et al.* 1990][1] under the operational semantics. This paper proposes a debugging method belonging to the execution tracing class.

In GHC programming, object-oriented[Shapiro and Takeuchi 1983] and stream-based[Kahn and MacQueen 1977] programming focus on the notion of processes and streams. Individual abstract modules are regarded as processes, some of which are connected by streams, and communicate with each other concurrently. A typical process repeatedly consumes data from a stream, changes its internal state, and generates data for another stream.

In a conventional execution tracer, it is difficult to capture conceptual execution in terms of processes and streams, because they are decomposed into GHC primitives and never displayed explicitly. The tracer we propose fully reflects the notion of processes and streams, and enables both the specific control flow of processes and the data structure of streams to be processed, making the causality among processes explicit.

2 Process Oriented Programs and Debugging

2.1 Models of Processes and Streams in GHC

2.1.1 Process model

A process can be interpreted either as a goal or as a set of goals, e.g., an "object" in object-oriented programming[Shapiro and Takeuchi 1983]. The following sections discuss processes based on the latter.

A process consists of goals for the continuation of the process or goals for internal procedures defined in the process. The continuation goal accepts streams in its arguments one by one, and reserves its internal state in other arguments. The stream argument takes a role of an I/O port for the data migration. The internal state is not affected by other processes, but is calculated by the previous state and input data captured from streams.

A process features:

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 inter-

[1]Even though the literature is concerned only with the execution tracing of Occam programs, its discussion is generally adaptable for most concurrent or parallel program debugging.

nal state 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.

2.1.2 Stream model

A stream is a sequence of logical terms whose operations[Tribble *et al.* 1987] are limited to reading the first term of a stream and writing a term to the tail of a stream.

A simple notation for streams is first introduced. Streams are constructed by stream-variables SV, stream-functors $\langle SH \| ST \rangle$ 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 denoting the first data of the stream, and ST is a stream representing the rest of the stream.

A stream features:

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

Connection: Streams S_a and S_b are connected if they are unified in the body. One of the connected streams is regarded as an alias of the other.

Equivalence relation $\cong$ is defined for the set of streams $\mathcal{S}$, used to visualize streams.

For substitution σ, relation $\simeq_\sigma$ is defined for $\mathcal{S}$, the set of streams consisting of terms obtained in the execution.

1. $S \simeq_\sigma S; {}^\forall S \in \mathcal{S}$
2. $\langle H \| S \rangle \simeq_\sigma S; {}^\forall S \in \mathcal{S}$
3. $S_1\sigma = S_2\sigma \Rightarrow S_1 \simeq_\sigma S_2, {}^\forall S_1, S_2 \in \mathcal{S}$

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 rule means that connected streams are also elements of the same equivalence class. Relation $\cong_\sigma$ is defined as the symmetric and transitive closure of relation $\simeq_\sigma$. Below, relation $\cong$ is written in place of $\cong_\sigma$ if substitution σ is clearly understood from the context.

In GHC, a stream is actually implemented by a list in most programs, i.e. stream-functor $\langle D \| S \rangle$ and stream-terminator $\langle \rangle$ correspond to term $[D' \mid S']$ and atom [].

2.2 Process Oriented Debugging

GHC programs based on the process model are called process oriented programs, each goal in the execution trace belongs to a process, which is either a continuation or a part of an internal procedure of the process. In tracing and checking process oriented programs, the goals belonging to a target process must first be extracted from the "chaotic" execution trace where these goals are interleaved.

The data flow must also be checked. Unless a process inputs intended data, the process outputs incorrect data to its output stream, or becomes permanently suspended. Intended data may not be sent to the process for two possible reasons. First, an adjacent process corresponding to the producer of the data malfunctions. Or, second, the input of the process is disconnected with the output of the producer, an error caused by misuse of a shared variable, in which case it is easier to detect the error if the stream connection between processes, called "a process network," is displayed.

To make process oriented programs execution traces easier to read, the process oriented debugger(**POD**) we propose, visualizes process and stream information structured from input/output data, internal state values, internal procedure traces and the stream connection between processes.

Programs can be debugged as follows:

Step 1 A user starts execution of a target program.

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

Step 3 The program code corresponding to a process where an error occurs is checked in detail, with any adjacent processes possibly contributing to the anomaly also checked.

Step 4 Input/output data sequences are saved for checking an abnormal process because comparing the sequence of output data before and after a program is modified makes it easier to check the behavior.

If the process malfunctions in Step 3 and 4, it is forcibly suspended and overall execution is continued as far as possible because program reexecution takes much time and costs, i.e., reexecution must be avoided if it will take too much data in streams up to a sufficient length. Otherwise the program will have nondeterministic transitions.

Reexecution can be avoided either by giving the debugger the functions to delete or to modify unexpected data and to insert data in a stream interactively, or by having functions preserve data in streams automatically and execute a process in the preserved environment,

Thus the POD requires the following execution control functions:

1. Forcibly suspending, resuming and aborting the execution of each process.

2. Buffering and modifying the data in streams interactively.

3. Reexecuting a process in the preserved environment.

3 Implementing the POD

3.1 Process Declaration

In our process model(Section 2.1.1), goals are classified into those for the continuation and those for internal procedures. They are syntactically the same, and are specified by the user in a process declaration.

The process declaration consists of a predicate specification and continuation marking. The predicate specification begins with the keyword `process` followed by the name of the predicate specifying the usage of each argument. The usage of each argument is specified by declaring keyword `state` or `port` in an appropriate order. Annotation `state` shows that the argument represents a part of the internal state. Annotation`port` shows that the argument represents a process's I/O port. The continuation mark consists of a `@` preceding the goal in a clause. An example of the process declaration is given in Listing 1.

3.2 Stream Treatment

As mentioned previously, streams consist of special variables, functors, and terminators.

In the POD, streams are recognized and supported by introducing tagged data structures. Each variable, functor, and atom that makes up a stream has an auxiliary field to store the stream identifier. An identifier is associated with each stream equivalence class.

In implementing identifiers, note that if two streams with different identifiers are unified, their identifiers should be the same. This is achieved by assigning a variable to each identifier and unifying the identifiers if their streams are to be unified. The problem of whether two streams satisfy equivalence relation $\cong$ is solved in a variable equivalence check.

The POD recognizes and manages streams as follows: First, before starting the execution, a program translator, which is a subsystem of the POD, converts a target program to a canonical form as detailed in Section 3.4. Streams are replaced with special tagged terms, and extended unifications are placed for their unifications which causes accessing data in streams or connecting streams as described in Section 2.1.

All the parameters of each process are stored in its process table every own execution step. Process execution is visualized using these process tables.

3.3 Reflective Extension of Unifier

Section 3.2 addressed a need for extended unification, discussed in more detail together with its implementation with reflection.

Tagged structures must be implemented using wrapped terms 'Sm'(Var,ID), 'Sm'(Atom,ID), and 'Sm'({Cons,Head,Tail},ID). The first term represents the fresh variable of a stream whose first argument, Var, corresponds to the original variable. The second, ID, is a fresh variable that denotes the identifier of its stream. The second term represents the terminator of a stream whose first argument, Atom, abstracts $\langle\rangle$, while the third corresponds to $\langle$Head$\|$Tail$\rangle$ and Cons is a functor for concatenation.

Terms are classified into six types: variable, atom, compound term[2], stream-variable, stream-functor, and stream-terminator.

New unification rules are needed for stream-term $\times$ stream-term and stream-term $\times$ regular-term. The following cases are representative of the extended unification X >< Y:

Case 1	X is stream variable 'Sm'(V,ID), Y is a variabl Assign Y to X.
Case 2	X is stream variable 'Sm'(V_1,ID_1), Y is stream variable 'Sm'(V_2,ID_2). Assign V_2 to V_1 and ID_2 to ID_1.
Case 3	X is stream variable 'Sm'(V,ID), Y is compound term {C,H,T}. Assign {C,H,'Sm'(N,ID)} to V, and execute 'Sm'(N,ID) >< T, where N is a fresh variable.
Case 4	X is stream functor 'Sm'({C_1,H_1,T_1},ID). Y is compound term {C_2,H_2,T_2}. Assign H_2 to H_1 and C_2 to C_1. Execute T_1 >< T_2 recursively. □

[2]In KL1, notation $\{F, A_1, ..., A_n\}$ is allowed to express compound term $F(A_1, ..., A_n)$. We follow the notation for convenience.

The remaining 17 possible cases are omitted here due to space considerations.

The variable check is essential when describing the unifier and is done by reflection [Smith 1984]. Because reflection provides functions to manage memory and goal queue, it becomes easy to implement streams.

Before developing the POD, we added a reflective feature to GHC similar to that for RGHC[Tanaka 1988]. When a user-defined reflective predicate is invoked, its arguments are automatically converted from internal representation to the meta-level ground form. Table 1 shows the correspondence between object-level and meta-level terms.

Case 3 is described using a reflective predicate whose second argument Gs is a stream connected with the goal scheduler.

```
reflect(vector({atom('Sm'),variable(V),ID}) ><
        vector({C,H,T}),Gs,Mm) :- true |
   Gs = [variable(V) = vector({C,H,vector(
               {atom('Sm'),variable(N),ID})}),
         vector({atom('Sm'),variable(N),ID}) >< T ],
   Mm = [ malloc(N)].
%% Gs: Goal scheduler, Mm: Memory manager.
```

The third argument Mm is a stream connected to the memory manager for the object program. Terms written in stream Gs are converted from meta-level ground terms to internal representation and placed in the goal queue. Terms written in stream Mm are understood as messages for memory access. Message *malloc*(N) invokes dynamic memory allocation, and the reference pointer to allocated memory is bound to variable N. Extended unification is defined similarly by the reflective predicate for all cases.

3.4 Tagged Term Transformation

As described above, tagged terms are represented as wrapped functors. The translator converts streams to tagged terms automatically. In the following, we show program examples before and after the conversion, then we explain the detail of the translating process. Furthermore we present additional transformation steps to direct the data migration, in other words, to detect the origin of data.

```
%Original program
process p(port,port).
boot :- true | p([1,2|_],X), q(X).
p([A|X],Y) :- true | Y=[A|Y1], @p(X,Y1).
%Conversion for streams
boot :- true | p([1,2|_],X,'Sm'(N1,ID1),'Sm'(N2,ID2)),
        q(X,'Sm'(N3,ID3)),
        [1,2|_] >< 'Sm'(N1,ID1), NX >< 'Sm'(N2,ID2),
        NX >< 'Sm'(N3,ID3).
p([A|X],Y,NA1,NA2) :- true | Y=[A|Y1],
        p(X,Y1,'Sm'(N1,ID1),'Sm'(N2,ID2)),
        NA1 >< [DA|DX], NA2 >< [DA|DY1],
        DX >< 'Sm'(N1,ID1), DY1 >< 'Sm'(N2,ID2).
```

The converted program differs from the original in the following ways:

1. The arity of predicate p doubles, i.e. the third and fourth arguments are new, and the parameters of p are converted to tagged terms for streams such as `'Sm'(N1,ID1)`.

2. The first and second arguments of the converted p are the same as those of the original, and the corresponding parameters are maintained.

3. Several extended unifications are added in the body.

The above points characterize the transformation: Two kinds of variable bindings are treated. One is the same as the original bindings and is used for the execution of the guard goal. The other consists of tagged terms for streams, and is used in extended unifications.

According to GHC semantics, unification invoked in a guard can not export any bindings to the caller. Furthermore user-defined predicates can not be placed in a guard. Because it is not easy to extend the guard execution rule of GHC, we follow the semantics as much as possible.

In our transformation, by maintaining the original bindings, the guard execution involving the parameter passing is independent of the term extension, and the extension never causes execution errors. The memory consumption by storing two kinds of bindings is, however, at least twice as much as that of the original.

Transformation processes are detailed as follows:

Step 1 Choose a clause, and erase all the guard unifications by partial evaluation[Ueda and Chikayama 1985]. Replace nonvariable argument Arg to fresh variable Var, and add goal Arg = Var in the guard. By applying the replacement for every argument, we get a canonical form such that every argument is a variable and every guard goal is ether a unification = of a variable and a nonvariable term, a difference \=, an arithmetic comparison or a type checker. We write a canonical clause as `P(A1,...,An):- G(A1,...,An) | Q(A1,...,An,B1,...,Bm)`, where `G(A1,...,An)` and `Q(A1,...,An,B1,...,Bm)` represent a conjunction of goals.

Step 2 Rename all variables in the clause and get a clause: `P(A1',...,An'):- G(A1',...,An') | Q(A1',...,An',B1',...,Bm')`. Extract all the unifications from `G(A1',...,An')`, and replace symbol = of unification with symbol >< of extended

Table 1: Representations of meta-level terms

Level	Term		
	Unbound variable	Atom	Compound
Object	*unobservable*	*Atom*	$\{C_1, \cdots, C_n\}$
Meta	variable(*Addr*)	atom(*Atom*)	vector($\{C'_1, \cdots, C'_n\}$)

unification. The obtained conjunction is written as `G'(A1',...,An')`.

Step 3 For goals defined as processes or as continuations in `Q(A1',...,An',B1',...,Bm')`, get conjunction `Q'(A1',...,An',B1',...,Bm')` by repeating follower. Replace port-declared parameter Param with stream 'Sm'(N,ID), where N and ID are fresh variables. Place extended unification Param `><` 'Sm'(N,ID) in the body of the new clause.

Step 4 For two goals, `B(D1,...,Di)` in `Q(A1,...,An, B1,...,Bm)` where `Dj`,1 $\leq$ `j` $\leq$ `i`, is ranged over `{A1,...,An,B1,...,Bm}`, and `B'(D1',...,Di')` in `Q'(A1',...,An',B1',...,Bm')`, goal `B''(D1,..,Di,D1',..,Di')` is defined as their concatenation. Conjunction `Q''(A1,...,An, B1,...,Bm,A1',...,An',B1',...,Bm')` is defined by combining every `B''`.

Step 5 An objective clause is obtained by combining `G`, `G'` and `Q''` as follows:

```
P(A1,...An,A1',...An'):- G(A1,..An) |
  G'(A1',..An'),
  Ci >< 'Sm'(S1,ID1),...,Ci >< 'Sm'(Si,IDi),
  Q''(A1,...,An,B1,..,Bm,A1',...,An',B1',..,Bm').
% Replace Cj in A1',...,Bm' to 'Sm'(Sj,IDj)
```

Detecting the origin of the data is achieved by using a tag similar to that stated above. A tagged functor 'Sb'(Term,PID) is introduced, where Term corresponds to the original term and may include other tagged structures, PID is an unbound variable used as a process identifier.

We show a modified example program using 'Sb' tag, then the additional transformation steps are detailed.

```
%Conversion for detecting the origin of the data
boot(PIDself) :- true |
        p([1,2|_],X,'Sm'(N1,ID1),'Sm'(N2,ID2),PID1),
        q(X,'Sm'(N3,ID3),PID2),
        'Sb'(['Sb'(1,PIDself)|
        'Sb'(['Sb'(2,PIDself)|_],PIDself)],PIDself)
        >< 'Sm'(N1,ID1),
        NX >< 'Sm'(N2,ID2), NX >< 'Sm'(N3,ID3).
p([A|X],Y,NA1,NA2,PIDself) :- true |
        Y=[A|Y1],
        p(X,Y1,'Sm'(N1,ID1),'Sm'(N2,ID2),PIDself),
        NA1 >< 'Sb'([DA|DX],PID1),
        NA2 >< 'Sb'([DA|DY1],PIDself),
        DX >< 'Sm'(N1,ID1), DY1 >< 'Sm'(N2,ID2).
%% PID1 specifies the origin of the input list.
```

Step 6 Add argument $\mathrm{PID}_{\mathrm{self}}$ to the head of the selected clause.

Step 7 Select predicate p/n in the body of the clause and, if p/n is declared as a process, add new parameter $\mathrm{PID}_{\mathrm{p/n}}$ or else add parameter $\mathrm{PID}_{\mathrm{self}}$.

Step 8 Recursively replace every nonvariable term T_i except streams in `G'(A1',...,An',B1',...,Bm')` with term 'Sb'(T'_i,PID_i). Each PID_i is used to indicate the origin of corresponding data.

Step 9 Replace every nonvariable parameter T of the extended unifications with term 'Sb'(T',$\mathrm{PID}_{\mathrm{self}}$).

3.5 Execution Control

In the POD, the specific control of a process proposed in Section 2.2 is achieved by introducing a valve inserted into a stream. The valve serves as an intelligent data buffer having two input ports, one output port, and a programmable conditional switch to close the output port. One of the two input ports is connected to the original stream, and the other is connected to the user's console. The user can send commands to the valve. The amount of buffered data and the description of the type of storable data are programmable conditions.

The valve has three states, automatic migration mode, conditional migration mode, and manual edit mode, each changed by a user command or by evaluating programmable conditions. The valve operates as follows:

- In automatic migration mode, the valve receives data from its own input port and it stores the data in its own buffer. Once the buffer becomes full, the valve outputs the first data in the buffer through the output port.

- In conditional migration mode, The valve gets data then stores it in the buffer. Once the buffer becomes full or if data does not satisfy a condition, the valve displays an alert and changes to manual editing mode.

- In manual editing mode, the valve receives no new data. The number and the description of data to be stored, and data actually in the buffer can be referenced and modified using a text editor. After editing, the mode returns to the previous mode.

A data checking condition is provided as the conjunction of GHC goals. The goal is a built-in or user-defined predicate. Built-in predicates are classified into type check, arithmetic comparison, and guard unification. The type check goal is, e.g., *atom*(), *integer*(), or *float*(). the arithmetic one is, e.g., $>, \geq, <, \leq$. The user-defined goal is a combination of built-in goals.

4 Examples of Tracing

The POD is developed by extending the GHC interpreter with reflection in Prolog. A user can trace and debug a GHC program with a direct manipulation interface provided by the POD.

The interface provides several control facilities for the target program in a menu, enabling the user to easily manipulate the POD by selecting a facility from a menu with a mouse. The menu currently provides, (1) compulsive process suspension, (2) process resumption, (3) valve insertion, (4) valve control, and (5) terminated process deletion.

The POD provides different three views to visualize program execution: the stream graph, process char, and communication flow.

The stream graph uses animated icons and lines to show dynamic changes in a network graph of processes and streams.

The process chart displays, in a structured diagram, consumed and generated data from or to streams in a process and its subprocesses. More specifically, the diagram contains dots, two kinds of lines, and data. A dot represents a process's argument in each execution step. One kind of line connecting two dots is associated with relation $\cong$ between them. Consumed or generated data is located along this line. The other kind of line represents a subprocess fork point.

The communication flow[Shin 1991] shows I/O process causality. When a substitution generated in process X is referenced in a committed clause of process Y, a directed arrow from X to Y is displayed. we say in this case that data from process X makes Y active.

The usages of the menu and the views are described using a program in Listing 1, first suppose that the program and query `prime(10,Ps)` are given to the POD.

Listing 1: Primes generator program with process declaration

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

Figure 1 shows the initial stream graph. Data in a stream that connects `gen` and `sift` can be checked in two ways, by setting a valve to either an output port of `gen` after suspending `gen` to prevent the creation of new data, or an input port of `sift` to avoid consuming data. Let item (1) be selected to gen suspend rather than `sift`. Selecting item (3), then (2), resumes `gen`. The generated valve is displayed as an icon in the window as for a process. Initially, the valve is in automatic migration mode and the default buffer is set to 100.

After process gen finishes generating data, information in the valve is displayed in a new dialog window if item (4) is selected. Figure 2 shows that buffer contents are modified by deleting the number 8. 0Assume, then, that the window is closed and flushes all buffer data flushed.

Flushing data causes `sift` to resume(Figure 3) with the stream graph eventually becoming stable.

Process charts for each process in a window are shown in Figure 4. In this figure:

- Process `gen` maintains an output stream specified by a vertical gray line at left in the window which connects all third arguments obtained at each execution step. Numbers generated by `gen` are aligned and displayed along this line.

- Process `filter` maintains both an input and an output stream specified by two vertical lines – black and gray in the middle of the window. The input sequence of numbers beside the black line, ranges from 3 to 9, with 8 deleted. Process `filter` generates or does not generate at each execution step when the output sequence on the gray line is referenced.

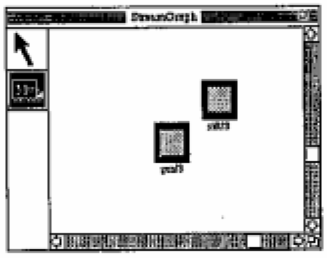
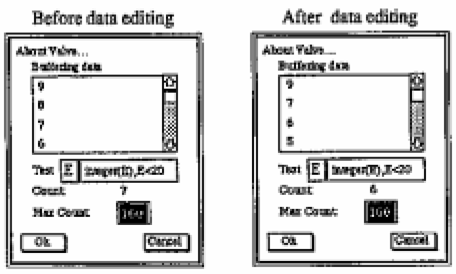


Figure 1: Initial stream graph



About Valve...
Buffering data
9
8
7
6
Test E integer(E),E<20
Count 7
Max Count 100
Ok Cancel

About Valve...
Buffering data
9
7
6
5
Test E integer(E),E<20
Count 6
Max Count 100
Ok Cancel

Figure 2: Valve controller display

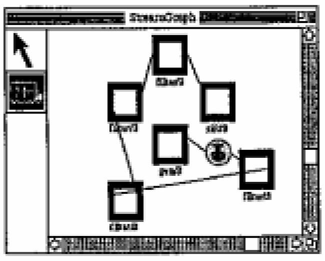


Figure 3: Final stream graph

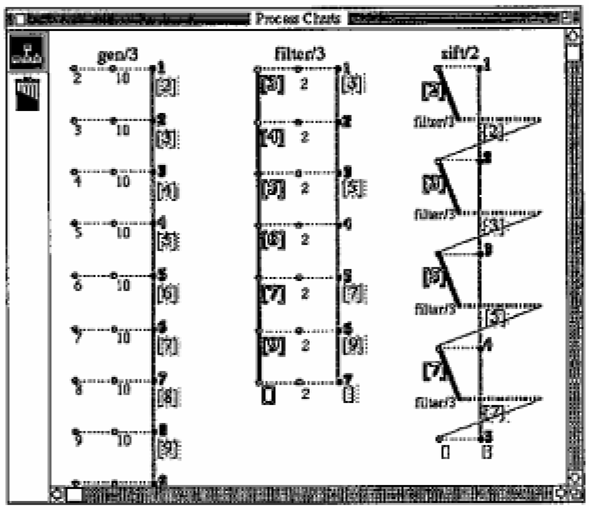


Figure 4: Process chart display

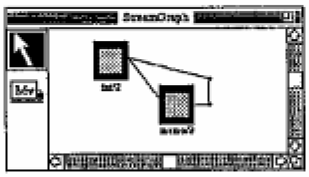


Figure 5: Stream graph for `int/2` and `memo/3`

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

Listing 2: Bounded buffer program

```
process int(state,port), memo(port,port,state).
bb(N):- true | open(N,H,T), int(0,H), memo(H,T,C).
open(0,H,T):- true | H=T.
open(N,H,T):- N>0 | N1:=N-1, H=[_|H1],
        open(N1,H1,T).
int(N,[X|S]):- true | X=s(N), @int(s(N),S).
memo([s(X)|S],T,C):- true | T=[_|T1],
        @memo(S,T1,s(X)).
```

The bounded buffer program is shown in Listing 2. Assume that the program and query `bb(5)` are given. The query goal invokes processes `int` and `memo`, which are connected after internal procedure open terminates. Figure 5 shows the stable stream graph. The communication flow of these processes indicates the alternate transition of two states. At left in Figure 6, Process `memo` becomes active by consuming data derived by the inactive `int` and a stream functor derived by the previous `memo`. At right, data from the inactive `memo` activates `int`.

5 Conclusion

We have proposed a process oriented debugger(POD) for GHC programs based on a computation model for processes and streams. The POD enables

- Overall behavior of a process to be controlled by manipulating data in streams and arbitrary delaying the transmission and reception of data between processes,

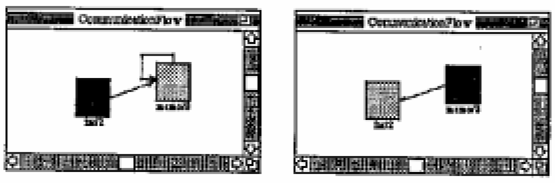


Figure 6: Communication flow transition

- Process causality to be shown using animated figures of processes and streams in both stream graph and communication flow displays,
- Stream connectivity to be organized and shown in a process chart, as a structure of lines connecting the arguments of a process.

Because individual goal execution is not a concern, our debugger gives some information such as input and output substitutions and timing in less detail, making it necessary to include a viewpoint in the future that interprets the original sequence of primitives in such a way that the user can follow it.

Our debugger is implemented using reflection and program transformation. Reflection makes it easy to describe extended unification, and program transformation guarantees the efficient execution of guard goals under the standard guard execution mechanism.

Acknowledgments

This research has been carried out as part of the Fifth Generation Computer Project of Japan. Dongwook Shin and Youji Kohda contributed insightful comments. Masaki Murakami assisted in formalizing streams. Simon Martin helped with the English. The author would like to express thanks to them.

The research originated in his postgraduate study, and he is indebted to Hirotaka Uoi and Nobuki Tokura of Osaka University for their invaluable advice.

References

[Goldszmidt *et al.* 1990] 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, November 1990.

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

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

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

[Shin 1991] D.Shin: "Towards Realistic Type Inference for Guarded Horn Clauses", Proceedings of Joint Symposium on Parallel Processing '91, pp.429-436, 1991.

[Smith 1984] B.C.Smith:"Reflection and Semantics in Lisp", Conference Record of the 11th Annual Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

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

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

[Tribble *et al.* 1987] 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).

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

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