ICOT Technical Memorandum: TM-0783

TM-0783

Formal Semantics of Flat GHC

by
K. Ueda & M. Murakami

July, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 – 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Formal Semantics of Flat GHC

Kazunori Ueda and Masaki Murakami
(Institute for New Generation Computer Technology)

1. Introduction

GHC [1] is a simple concurrent programming language based on the process interpretation of Horn-clause programs. This suggests two approaches to the formal semantics of GHC: one is to adapt the formalism of concurrent processes such as CCS [2], and the other is to adapt the declarative semantics of logic programs [3]. This paper informally describes these two approaches.

This paper deals with a subset of GHC called Flat GHC [4], which restricts guard goals to the calls to *test predicates*, predicates defined using clauses with empty bodies. Flat GHC is expressive enough to describe concurrent processes and has two desirable properties; that is, guard goals are deterministic and generate no substitutions.

2. Semantics of Flat GHC as a Process Description Language

The semantics as a process description language, which is fully described in [4], was motivated by our previous work on transformation rules for Flat GHC programs that were finally formalized in [4].

2.1 Design Criteria

- *Modeling Behavior.* A multiset of GHC goals can be regarded as a process that communicates with the outside world by observing and generating substitutions. The semantics should model this behavioral aspect.
- *Abstractness.* The semantics should concentrate on communication. It should abstract away internal affairs of a process such as the number of (sub)goals and the number of commitments done. Also, it should abstract away *how* unification is specified in the source text.
- *Modeling non-terminating programs.* We must be able to define the semantics of programs that do not terminate but are still useful.
- *Modeling anomalous behavior.* Anomalous behavior such as failure of a unification goal in a clause body, irreducibility of a non-unification goal and infinite computation without observable substitution must be modeled, because we have to prove that such behavior is not introduced by program transformation.
- *Simplicity and generality.* The semantics should be as simple and general as possible to be widely used. We decided to use standard tools such as *finite* terms, substitutions defined over them, and least fixpoints. We decided *not* to use a mode system that assigns the mode *input* or *output* to each argument of a predicate. We decided *not* to handle discontinuous concepts like fairness.

2.2 The Semantics

The semantics of a multiset B_0 of goals under a program $\mathcal{P}$, denoted $[\![B_0]\!]_{\mathcal{P}}$, is modeled as the set of all possible finite sequences of transactions with it. A *(normal) transaction*, denoted $\langle \alpha, \beta \rangle$, is an act of providing a multiset of goals with a possibly empty *input substitution* α and getting an observable (see below) *output substitution* β. An output substitution is also called a *partial answer substitution*. We use idempotent substitutions to model information communicated with a multiset of goals.

The first transaction $\langle \alpha_1, \beta_1 \rangle$ must be made through the variables in $\mathrm{var}(B_0)$, called the *interface* of B_0. The above observability condition for β_1 can be written as $B_0\alpha_1\beta_1 \not\equiv B_0\alpha_1$. As the result of the first transaction, B_0 will be reduced to a multiset B_1 of goals, which represents the rest of the computation. Then the second transaction $\langle \alpha_2, \beta_2 \rangle$ must be made through the interface $\mathrm{var}(B_0\alpha_1\beta_1)$.

The 'size' of a transaction depends on how the outside world observes an output substitution. Suppose B_0 returns a complex (or even infinite) data structure t in response to an input α_1. What should β_1 be, or what should the outside world see in one transaction? The answer is that the outside world can observe any *finite* anti instance of t (i.e., a term of which t is an instance). In our semantics, the result of one unification goal may be observed using two or more transactions, and the result of two or more unification goals may be observed in one transaction. A transaction is of a finite nature; it is realized by a finite number of reductions and can return only a finite data structure.

The outside world may not communicate with B_0 at all. This is modeled by always including ϵ (empty sequence) in $[\![B_0]\!]_{\mathcal{P}}$. The empty sequence is used as a base case in defining the semantics of B_0 inductively.

An input α_1 to B_0 may not necessarily cause a normal transaction as defined above. First, it may cause failure of a unification goal in a clause body. This is modeled by letting $\langle \alpha_1, \top \rangle \in [\![B_0]\!]_{\mathcal{P}}$, where $\top$ means failure. The transaction $\langle \alpha_1, \top \rangle$ means that the input α_1 may lead the computation to a chaotic situation. Second, B_0 may succeed (i.e., be reduced to an empty multiset) with no observable

output. Third, B_0 may deadlock (i.e., be reduced to a multiset of goals that does not allow further reduction) with no observable output. Fourth, B_0 may fall into infinite computation that does not generate observable output. The last three cases mean *inactivity* of B_0 and cannot be distinguished from outside; so they are all modeled by letting $\langle \alpha_1, \bot \rangle \in [\![B_0]\!]_{\mathcal{P}}$, where $\bot$ stands for 'no output'. However, if necessary, these cases could be distinguished in the semantics by using $\bot_{success}$, $\bot_{deadlock}$ and $\bot_{divergence}$ instead of $\bot$. Failure and inactivity are called *special transactions* and cannot be followed by other transactions; they are used as base cases in defining the semantics of B_0.

The semantics shown above, in retrospect, is similar to Hoare's semantics of nondeterministic processes [5] in the sense that both characterize processes in terms of their possibly anomalous behavior using a similar technique. However, the basic concepts of processes in these formalisms are quite different.

2.3 Examples

Consider a single clause program

```
P:  p(X) :- true | X=f(Y), p(Y).
```

and autonomous (i.e., empty input) transactions with $\mathcal{P}$. Then $[\![\texttt{p(X)}]\!]_{\mathcal{P}}$ has

ϵ,
$\langle \emptyset, \{\texttt{X} \leftarrow \texttt{f(X1)}\} \rangle$,
$\langle \emptyset, \{\texttt{X} \leftarrow \texttt{f(X1)}\} \rangle \langle \emptyset, \{\texttt{X1} \leftarrow \texttt{f(X2)}\} \rangle$,
$\langle \emptyset, \{\texttt{X} \leftarrow \texttt{f(X1)}\} \rangle \langle \emptyset, \{\texttt{X1} \leftarrow \texttt{f(X2)}\} \rangle \langle \emptyset, \{\texttt{X2} \leftarrow \texttt{f(X3)}\} \rangle$,
...

and also

$\langle \emptyset, \{\texttt{X} \leftarrow \texttt{f(f(X2))}\} \rangle$,
$\langle \emptyset, \{\texttt{X} \leftarrow \texttt{f(f(f(X3)))}\} \rangle$,
...

Note that $[\![\texttt{p(X)}]\!]_{\mathcal{P}}$ contains $\langle \emptyset, \bot_{divergence} \rangle$ as well, because the goal `X=f(Y)` may be ignored consistently in an unfair execution.

Our model successfully circumvents Brock-Ackerman anomaly [6]. Let $\mathcal{BA}$ be:

```
d([A|_],O) :- true | O=[A,A].

merge([A|X1],Y,Z) :- true |
    Z=[A|Z1], merge(X1,Y,Z1).
merge(X,[A|Y1],Z) :- true |
    Z=[A|Z1], merge(X,Y1,Z1).
merge([],Y,Z) :- true | Z=Y.
merge(X,[],Z) :- true | Z=X.

p1([A,B|_],O) :- true | O=[A,B].

p2([A|Z1],O) :- true | O=[A|O1], p21(Z1,O1).
p21([B|_],O1) :- true | O1=[B].

g1(I,J,O) :- true |
    d(I,X), d(J,Y), merge(X,Y,Z), p1(Z,O).
g2(I,J,O) :- true |
    d(I,X), d(J,Y), merge(X,Y,Z), p2(Z,O).
```

Then, the computation

$\langle \{\texttt{I} \leftarrow \texttt{[a|\_]}\}, \{\texttt{O} \leftarrow \texttt{[a|O']}\} \rangle$

belongs both to $[\![\texttt{g1(I,J,O)}]\!]_{\mathcal{BA}}$ and to $[\![\texttt{g2(I,J,O)}]\!]_{\mathcal{BA}}$ (O' being a fresh variable), but the computation

$\langle \{\texttt{I} \leftarrow \texttt{[a|\_]}\}, \{\texttt{O} \leftarrow \texttt{[a|O']}\} \rangle \langle \{\texttt{J} \leftarrow \texttt{[b|\_]}\}, \{\texttt{O'} \leftarrow \texttt{[b]}\} \rangle$

belongs only to $[\![\texttt{g2(I,J,O)}]\!]_{\mathcal{BA}}$ and not to $[\![\texttt{g1(I,J,O)}]\!]_{\mathcal{BA}}$.

3. Semantics of Flat GHC as a Logic Programming Language

In this section, we introduce a model-theoretic semantics of Flat GHC programs. For pure Horn-logic programming languages, the results on the model-theoretic semantics are reported in [3]. In that approach, the denotation of a program is given as the minimum model of the set of Horn clauses defining the program, in other words, as the set of ground unit clauses.

The idea of giving the semantics of a program by a set of unit clauses is a useful and elegant one also in the case of Flat GHC programs [7]. Levi [7] introduced the notion of *guarded atoms* as a GHC counterpart of the notion of unit clauses in ordinary logic programming. A guarded atom is a guarded clause such that all guard and body goals are unification atoms. However, in this approach, a guarded atom describes only the relation between input substitutions to a goal and computed substitutions obtained when the goal succeeds. This kind of relation is insufficient for discussing the infinite computation of a program.

The author reported another model-theoretic semantics of Flat GHC programs [8]. In that paper, the notion of I/O histories is introduced as a GHC counterpart of the notion of unit clauses. An I/O history is denoted as H :- GU, where H is the head denoting the form of a(n atomic) process, and GU is the body which is a partially ordered set denoting the inputs and the outputs of the process. H is of the form $p(X_1, \ldots, X_k)$, where p is a predicate symbol and $X_1, \ldots, X_k$ are distinct variables. GU is a set of tuples $<\sigma \mid U>$ that satisfies the conditions shown in Definition 3, where σ is an expression denoting a substitution and U is an expression denoting unification executed in a clause body. Intuitively, $<\sigma \mid U>$ means that when σ has been applied to the process since the beginning of the computation, the unification U is executed. An I/O history denotes a possible computation path of a program which is generated when the program is executed without any deadlock.

This section is mainly concerned with the body of an I/O history.

3.1 Guarded Stream

Let *Var* be an enumerable set of variables, and *Fun* be a set of function symbols. Each element of *Fun* has its arity. Let *Terms* be the set of terms defined from *Fun* and *Var* in a standard way. A term τ is said to be *simple* if it is a variable term, 0-ary function symbol or in the form of $f(X_1, \ldots, X_n)$ where $f \in Fun$ and $X_1, \ldots, X_n$ are distinct

variables. Let τ be a simple term and $X \in Var$. Then $X = \tau$ is called a *substitution form*. In this section, a set $\sigma = \{X_1 = \tau_1, \ldots, X_n = \tau_n\}$ of substitution forms, where X_i's are distinct variables and $X_j \neq \tau_j$ for $1 \leq j \leq n$, is intended to represent a substitution σ^k, where k is such that $\sigma^{k+1} = \sigma^k$ (we suppose such k exists). For example, a substitution represented by the set $\{\mathtt{X = [A|Y]}, \mathtt{A = a}\}$ maps `X` to `[a|Y]`.

Definition 1 Guarded Unification.
Let σ be a set of substitution forms and U be a substitution form. Then $<\sigma \mid U>$ is called a *guarded unification*. ∎

Let GU be a set of guarded unifications, which represents a possible computation of a program. For any $<\sigma_1 \mid U_1>, <\sigma_2 \mid U_2> \in GU$, U_1 is executable before U_2 if $\sigma_1 \subset \sigma_2$.

Definition 2 Ordering Between Guarded Unifications.
Let GU be a set of guarded unifications. For $<\sigma_1 \mid U_1>, <\sigma_2 \mid U_2> \subset GU$, we define

$$<\sigma_1 \mid U_1> \prec <\sigma_2 \mid U_2>$$

to hold if and only if $\sigma_1 \subset \sigma_2$ and $\sigma_1 \neq \sigma_2$. ∎

Obviously, $\prec$ is a well-founded ordering.

Definition 3 Guarded Stream.
A set of guarded unifications GU is called a *guarded stream* if the following hold.

(1) For any $<\sigma_1 \mid X_1 = \tau_1>, <\sigma_2 \mid X_2 = \tau_2> \in GU$, τ_1 and τ_2 are unifiable if $X_1 \equiv X_2$.

(2) For any $<\sigma_1 \mid X_1 = \tau_1>, <\sigma_2 \mid U_2> \in GU$ (that are possibly identical), $(X_1 = \tau_1) \notin \sigma_2$.

(3) For any $<\sigma_1 \mid U_1>, <\sigma_2 \mid U_2> \in GU$, if $(X = \tau) \in \sigma_1$ and $(X = \tau') \in \sigma_2$, then τ and τ' are unifiable. ∎

Next, a composition operation called *synchronized merge* is defined in order to obtain the guarded stream representing the computation of a whole goal clause from the guarded streams representing the computations of each goal in the goal clause [8]. We first introduce the basic idea of Definition 4 shown below. Let $GU_1, \ldots, GU_n$ be guarded streams. Each GU_i represents the computation of the ith process. Then, the synchronized merge of $GU_1, \ldots, GU_n$ (denoted $GU_1 \parallel \cdots \parallel GU_n$) is the union of $GU_1, \ldots, GU_n$, the following cases excepted. Assume $<\sigma_i \mid X = \tau> \in GU_i$, $<\sigma_j \mid U> \in GU_j$, and $(X = \tau) \in \sigma_j$. Namely, the ith process is the producer of X and the jth process is a consumer of X. Then $<\sigma_j \mid U>$ is replaced with $<\sigma_i \cup \sigma_j \setminus \{X = \tau\} \mid U>$ in $GU_1 \parallel \cdots \parallel GU_n$. Intuitively, this means that if these two process are executed in parallel, the communication that sends τ through the stream X is localized and no process waits for τ to come through X from the environment. The formal definition is as follows:

Definition 4 Synchronized Merge.
Let $GU_1, \ldots, GU_n$ be guarded streams, and $Gu_k (0 \leq k)$ be as follows:

$$Gu_0 = \{<\sigma \mid U> \mid (\exists i\,(<\sigma \mid U> \in GU_i)) \wedge (\forall U' \in \sigma\, \forall j\, \forall \sigma'\, (<\sigma' \mid U'> \notin GU_j))\}$$

$$\begin{aligned} Gu_{k+1} = Gu_k \cup \{<\sigma \mid U> \mid \exists \sigma'\,(& (\exists i\,(<\sigma' \mid U> \in GU_i)) \\ & \wedge (\forall U' \in \sigma'\,((\forall j\, \forall \sigma''(<\sigma'' \mid U'> \notin GU_j)) \\ & \qquad \vee (\exists \sigma''\,(<\sigma'' \mid U'> \in Gu_k)))) \\ & \wedge (\sigma = (\sigma' \setminus \{U'' \mid \exists \sigma''\,(<\sigma'' \mid U''> \in Gu_k)\}) \\ & \qquad \cup \{U'' \mid \\ & \qquad\quad \exists \sigma''\,((\exists U' \in \sigma'\,(<\sigma'' \mid U'> \in Gu_k)) \\ & \qquad\qquad \wedge (U'' \in \sigma''))\}) \\ &)\} \end{aligned}$$

Moreover, let GU be $\bigcup_{k=\infty} Gu_k$. If GU is a guarded stream and if

$$\{U \mid \exists \sigma\,(<\sigma \mid U> \in GU)\} = \{U \mid \exists i \exists \sigma\,(<\sigma \mid U> \subset GU_i)\},$$

then the *synchronized merge* of $GU_1, \ldots, GU_n$ (denoted $GU_1 \parallel \cdots \parallel GU_n$) is defined to be GU. ∎

Informally, the Gu_k's can be understood as follows. Let $<\sigma \mid U> \in GU_i$. If $<\sigma \mid U> \in Gu_k$, it means that when σ is provided from the environment of $GU_1 \parallel \cdots \parallel GU_n$, U can be executed after at most k rounds of internal communication among the GU_j's. For instance, $<\sigma \mid U> \in Gu_0$ means that (the computation represented by) GU_i can execute U without receiving any substitution from the other GU_j's, and $<\sigma \mid U> \in Gu_1$ means that GU_i can execute U after receiving some substitution that the GU_j's have generated without internal communication.

3.2 Example

We consider the program $\mathcal{BA}$ in Section 2.3 augmented with the following clauses:

```
inv([a|I],O) :- true | O=[b].
inv([b|I],O) :- true | O=[a].
```

Let GU_1 represent the computation obtained by executing the goal `g1(I,J,O)` with the input

$$\sigma_1 = \{\mathtt{I = [A|I1]}, \mathtt{A = a}, \mathtt{J = [B|J1]}, \mathtt{B = b}\}.$$

Then

$$\begin{aligned} GU_1 = \{ & <\sigma_2 \mid \mathtt{O = [A1|O1]}>, <\sigma_1 \mid \mathtt{A1 = a}>, \\ & <\sigma_1 \mid \mathtt{O1 = [B1|O2]}>, <\sigma_1 \mid \mathtt{B1 = b}>, \\ & <\sigma_1 \mid \mathtt{O2 = []}>\}. \end{aligned}$$

Let GU_2 represent the computation obtained by executing the goal `inv(O,J)` with the input

$$\sigma_2 = \{\mathtt{O = [A1|O1]}, \mathtt{A1 = a}\}.$$

Then

$$\begin{aligned} GU_2 = \{ & <\sigma_2 \mid \mathtt{J = [B|J1]}>, <\sigma_2 \mid \mathtt{B = b}>, \\ & <\sigma_2 \mid \mathtt{J1 = []}>\}. \end{aligned}$$

In this case, $Gu_1 = \emptyset$ and hence $GU = \emptyset$. Thus,

$$\{U \mid \exists \sigma\,(<\sigma \mid U> \in GU)\} = \emptyset.$$

However,

$$\{U \mid \exists i \exists \sigma (<\sigma | U> \in GU_i)\}$$
$$= \{\texttt{O=[A1|O1], A1=a, O1=[B1|O2], B1=b, O2=[],}$$
$$\texttt{J=[B|J1], B=b, J1=[]}\}.$$

Thus,

$$\{U \mid \exists \sigma (<\sigma \mid U> \in GU)\} \neq \{U \mid \exists i \exists \sigma (<\sigma | U> \in GU_i)\}$$

and therefore the synchronized merge of GU_1 and GU_2 cannot be defined.

On the other hand, let GU_1' represent the computation of the goal g2(I,J,O) in which

(1) the input $\sigma_3 = \{\texttt{I=[A|I1], A=a}\}$ is given first,

(2) O=[a|O1] is output in response, and then

(3) σ_1 is given.

Then

$$GU_1' = \{<\sigma_3 \mid \texttt{O=[A1|O1]}>, <\sigma_3 \mid \texttt{A1=a}>,$$
$$<\sigma_1 \mid \texttt{O1=[B1|O2]}>, <\sigma_1 \mid \texttt{B1=b}>,$$
$$<\sigma_1 \mid \texttt{O2=[]}>\}.$$

The synchronized merge of GU_1' and GU_2 can be defined as follows:

$$GU_1' \parallel GU_2$$
$$= \{<\sigma_3 \mid \texttt{O=[A1|O1]}>, <\sigma_3 \mid \texttt{A1=a}>,$$
$$<\sigma_3 \mid \texttt{J=[B|J1]}>, <\sigma_3 \mid \texttt{B=b}>,$$
$$<\sigma_3 \mid \texttt{J1=[]}>,$$
$$<\sigma_3 \mid \texttt{O1=[B1|O2]}>, <\sigma_3 \mid \texttt{B1=b}>,$$
$$<\sigma_3 \mid \texttt{O2=[]}>\}.$$

$GU_1' \parallel GU_2$ represents the computation obtained by executing the multiset of goals {g2(I,J,O), inv(O,J)} with the input substitution σ_3. ∎

Thus, the semantics presented here also circumvents Brock-Ackerman anomaly, and actually it is compositional with respect to the conjunction (AND-parallel execution) of goals. Namely, the guarded stream that represents the computation of a multiset of goals is obtained from the guarded streams that represent the computation of individual goals.

The domain of I/O histories has been introduced as a GHC counterpart of a Herbrand base in logic programming. However, each element of this domain is not a GHC clause syntactically. Recently, a syntactical extension of Flat GHC called NGHC (Nested Guarded Horn Clauses) was proposed [9]. An NGHC unit clause represents a possible computation, and the semantics of an NGHC program is defined with a set of unit clauses as in pure Horn-logic programs. The notion of NGHC unit clauses is essentially equivalent to the notion of I/O histories.

4. Conclusions and Future Work

The two semantics described above have been developed with different motivations and from different directions. The primary purpose of the first semantics (Section 2) was to define *observables* (i.e., entities that matters) that should be preserved by the program transformation rules we designed. The semantics is currently defined as an abstraction of an operational semantics; although the resulting semantics is quite intuitive, it is yet to be shown how the semantics of a program can be obtained compositionally from its components.

On the other hand, the primary purpose of the second semantics (Section 3) was to see how a semantics could be obtained in a compositional manner. Another difference is that the second semantics is based on a true concurrent model while the first semantics is based on interleaving. However, the second semantics as shown above does not deal with anomalous behavior and should therefore be combined with a semantics such as the one in [10]. Also, we must make certain that the resulting semantics coincides with our intuitive understanding of Flat GHC.

References

[1] Ueda, K. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. In *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, 1988, pp. 441–456.

[2] Milner, R. *A Calculus of Communicating Systems.* LNCS 92, Springer-Verlag, Berlin, 1980.

[3] Lloyd, J. W. *Foundations of Logic Programming (2nd ed.).* Springer-Verlag, Berlin, 1987.

[4] Ueda, K. and Furukawa, K. Transformation Rules for GHC Programs. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 582–591.

[5] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[6] Brock, J. D. and Ackerman, W. B. Scenarios: A Model of Non-determinate Computation. In *Formalization of Programming Concepts*, LNCS 107, Springer-Verlag, Berlin, 1981, pp. 252–259.

[7] Levi, G. A New Declarative Semantics of Flat Guarded Horn Clauses. ICOT Technical Report, TR-345, ICOT, 1988.

[8] Murakami, M. A New Declarative Semantics of Parallel Logic Programs with Perpetual Processes. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 374-381.

[9] Falaschi, M., Gabbrielli, M., Levi, G. and Murakami, M. Nested Guarded Horn Clauses: A Language Provided with a Complete Set of Unfolding Rules. To be presented at the Logic Programming Conf. '89, Tokyo, 1989.

[10] Murakami, M. A Failure Set Semantics of Guarded Horn Clauses Programs. In *Preprints of the 27th WGSF meeting*, IPS Japan, 1988.