Published in Lecture notes in computer science 1830, 144-153, 2000
which should be used for any reference to this work

The GIPSY Architecture

Joey Paquet[1] and Peter Kropf[2]

[1] Concordia University, Montreal, Canada H3G 1M8. paquet@cs.concordia.ca.
[2] Université de Montréal, Montréal, Canada H3C 3J7. kropf@iro.umontreal.ca

Abstract. Intensional Programming involves the programming of expressions placed in an inherent multidimensional context space. It is an emerging and highly dynamic domain of general application. The fast growing computer connectivity allows for more and more efficient implementation of distributed applications. The paradigm of intensionality inherently includes notions of parallelism at different levels. However, the currently available intensional programming software tools are becoming obsolete and do not enable us to further push forward practical investigations on the subject. Experience shows that the theoretical advancement of the field has come to acceptable maturity. Consequently, new powerful tools for intensional programming are required. In this paper, we present the design of a General Intensional Programming System (GIPSY). The design and implementation of the GIPSY reflect three main goals: generality, adaptability and efficiency.

1 Introduction

Intensional programming is a generalization of unidimensional contextual (a.k.a. modal logic) programming such as temporal programming, but where the context is multidimensional and implicit rather than unidimensional and explicit. Intensional programming is also called *multidimensional programming* because the expressions involved are allowed to vary in an arbitrary number of dimensions, the context of evaluation is thus a *multidimensional context*. For example, in intensional programming, one can very naturally represent complex physical phenomena such as plasma physics, which are in fact a set of charged particles placed in a space-time continuum that behaves according to a limited set of laws of intensional nature. This space-time continuum becomes the different dimensions of the context of evaluation, and the laws are expressed naturally using intensional definitions [7].

Lucid is a multidimensional intensional programming language whose semantics is based on the possible world semantics of intensional logic [1, 10]. It is a functional language in which expressions and their valuations are allowed to vary in an arbitrary number of dimensions. Intensional programming (in the sense of Lucid) has been successfully applied to resolve problems with a new perspective that enables a more natural understanding of problems of intensional nature. Such problems include topics as diverse as reactive programming, software configuration [8], tensor programming [7], distributed operating systems [6]. However, these projects have all been developed in isolation. GLU is the most general intensional programming tool presently available [4]. However, experience has shown that, while being very efficient, the GLU system suffers

from a lack of flexibility and adaptability [7]. Given that Lucid is evolving continually, there is an important need for the successor to GLU to be able to stand the heat of evolution.

We propose the design of a general intensional programming system (GIPSY). To cope with the fast evolution and generality of the intensional programming field, the design and implementation of all its subsystems is done towards *generality*, *flexibility* and *efficiency*.

2 Approach

The General Intensional Programming System (GIPSY) consists in three modular subsystems: The General Intensional Programming Language Compiler (GIPC) ; the General Eduction Engine (GEE), and the Intensional Run-time Programming Environment (RIPE). Although the theoretical basis of the language has been settled, the implementation of an efficient, general and adaptable programming system for this language raises many interrogations. The following sections outline the theoretical basis and architecture of the different components of the system. All these components are designed in a modular manner to permit the eventual replacement of each of its components—at compile-time or even at run-time—to improve the overall efficiency of the system.

2.1 General Intensional Programming Language Compiler (GIPC)

Like all functional programming languages, there are many variants of Lucid, depending on the basic set of types, constants and data operations, i.e. the basic algebra, made available to a user. Nevertheless, all variants of Lucid include function application, conditional expressions, intensional navigation and intensional query.

Language Syntax and Semantics

The language whose syntax and semantics is given in Figure 1 and explained below is capable of expressing all extensions to Lucid, proposed to this day.

This syntax assumes that identifiers (*id*) can refer to constants, data operations, variables, functions or dimensions. This approach comes from the fact that function and dimension identifiers can be first-class values in our version of Lucid. The operational semantics of Lucid is given in a structural operational semantics style. Normally this would mean that semantic judgments would be of the general form $\mathcal{D} \vdash E : v$ i.e. under the definition environment $\mathcal{D}$, expression E would evaluate to value v. However, in Lucid, we must take into account the *context of evaluation* of expressions, so we need an additional entry to the left, hence

$$\mathcal{D}, \mathcal{P} \vdash E : v$$

which means that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}$ (sometimes also referred to as *point*), expression E evaluates to v. The definition environment $\mathcal{D}$ retains the definitions of all of the identifiers that appear in a Lucid program. It is therefore a partial function

$$\mathcal{D} : \mathbf{Id} \rightarrow \mathbf{IdEntry}$$

$$
\begin{aligned}
E ::= \ & \mathit{id} \\
& \mid E(E_1, \ldots, E_n) \\
& \mid \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \\
& \mid \#E \\
& \mid E \ @E' \ E'' \\
& \mid E \texttt{ where } Q \\
Q ::= \ & \texttt{dimension } \mathit{id} \\
& \mid \mathit{id} = E \\
& \mid \mathit{id}(\mathit{id}_1, \ldots, \mathit{id}_n) = E \\
& \mid Q \ Q
\end{aligned}
$$

Fig. 1. Syntax of the Lucid language

where $\mathbf{Id}$ is the set of all possible identifiers and $\mathbf{IdEntry}$ has five possible kinds of value, one for each of the kinds of identifier:

- *Dimensions* define the coordinates in which one can navigate. The $\mathbf{IdEntry}$ is simply $(\texttt{dim})$.
- *Constants* are entities that provide the same value in any context. The $\mathbf{IdEntry}$ is $(\texttt{const}, c)$, where c is the value of the constant.
- *Data operators* are entities that provide memoryless functions, e.g. arithmetic operations. The constants and data operators define the *basic algebra* of the language. The $\mathbf{IdEntry}$ is $(\texttt{op}, f)$, where f is the function itself.
- *Variables* carry the multidimensional streams. The $\mathbf{IdEntry}$ is $(\texttt{var}, E)$, where E is the expression defining the variable. Uniqueness of names is achieved by performing compile-time renaming or using a nesting level environment [7].
- *Functions* are user-defined functions. The $\mathbf{IdEntry}$ is $(\texttt{func}, id_i, E)$, where the id_i are the formal parameters to the function and E is the body of the function. The semantics for recursive functions could be easily added, but is discouraged by the nature of intensional programming.

The evaluation context $\mathcal{P}$, which is changed when the `@` operator or a `where` clause is encountered, associates a tag to each relevant dimension. It is therefore a partial function

$$\mathcal{P} : \mathbf{Id} \to \mathbf{N}$$

The operational semantics of Lucid programs is defined in Appendix A. Each type of identifier can only be used in the appropriate situations. Identifiers of type, `op`, `func` and `dim` evaluate to themselves. Constant identifiers (`const`) evaluate to the corresponding constant. Function calls, resolved by the $\mathbf{E_{fct}}$ rule, require the renaming of the formal parameters into the actual parameters (as represented by $E'[id_i \leftarrow E_i]$).

For example, the rule for the navigation operator, $\mathbf{E_{at}}$, which corresponds to the syntactic expression $E \ @E' \ E''$, evaluates E in context $[E' : E'']$, where E' evaluates to a dimension and E'' to a value corresponding to a tag in E'. The rule for the `where` clause, $\mathbf{E_w}$, which corresponds to the syntactic expression E `where` Q, evaluates E using the definitions (Q) therein.

The additions to the definition environment and context of evaluation made by the $\mathbf{Q}$ rules are local to the current `where` clause. This is represented by the fact that the $\mathbf{E_w}$ rule returns neither $\mathcal{D}$ nor $\mathcal{P}$. The $\mathbf{Q_{dim}}$ rule adds a dimension to the definition environment and, as a convention, adds this dimension to the context of evaluation with tag 0. The $\mathbf{Q_{id}}$ and $\mathbf{Q_{fid}}$ simply add variable and function identifiers along with their definition to the definition environment.

The initial definition $\mathcal{D}_0$ includes the predefined intensional operators, the constants and the data operators. Hence

$$\mathcal{D}_0, \mathcal{P}_0 \vdash E : v$$

where $\mathcal{P}_0$ defines a particular context of interest, represents the computation of any Lucid expression, where v is the result.

The GIPSY Architecture and Program Compilation

GYPSY programs are compiled in a two-stage process (see Figure 2). First, the intensional part of the GYPSY program is translated into C, then the resulting C program is compiled in the standard way.

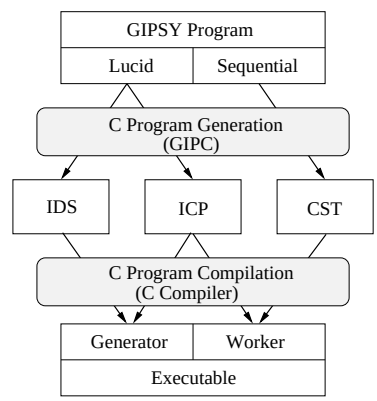

Fig. 2. GIPSY Program Compilation Process

The source code consists of two parts: the Lucid part that defines the intensional data dependencies between variables and the sequential part that defines the granular sequential computation units (usually written in C). The Lucid part is compiled into an intensional data dependency structure (IDS) describing the dependencies between each variable involved in the Lucid part. This structure is interpreted at run-time by the GEE (see Section 2.2) following the demand propagation mechanism. Data communication procedures used in a distributed evaluation of the program are also generated by the GIPC according to the data structures definitions written in the Lucid part, yielding a set of intensional communication procedures (ICP). These are generated following a given communication layer definition such as provided by IPC, CORBA or the WOS [2]. The

sequential functions defined in the second part of the GIPSY program are translated into C code using the second stage C compiler syntax, yielding C sequential threads (CST).

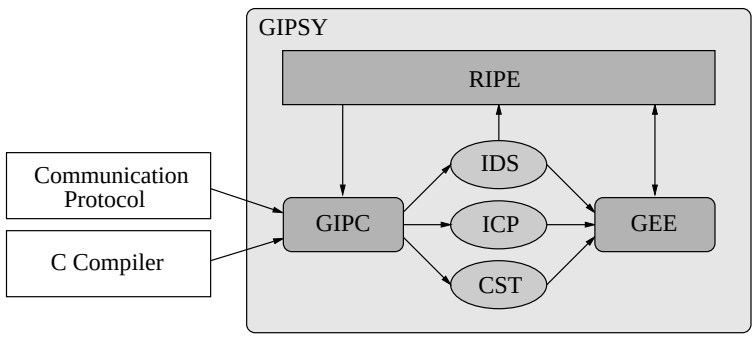

Fig. 3. GIPSY Software Architecture

Intensional function definitions, including higher order functions, will be flattened using a well-know efficient technique [9]. Because of the interactive nature of the RIPE (see Section 2.3), the GIPC is modularly designed to allow the individual on-the-fly compilation of either the IDS (by changing the Lucid code) ICP (by changing the communication protocol) or CST (by changing the sequential code). Such a modular design even allows sequential threads to be programs written in different languages.

2.2 General Eduction Engine (GEE)

The GIPSY uses a demand-driven model of computation, whose principle is that a computation takes effect only if there is an explicit demand for it. The GIPSY uses eduction, which is demand-driven computation in conjunction with an intelligent value cache called a warehouse. Every demand generates a procedure call, which is either computed locally or remotely, thus eventually in parallel with other procedure calls. Every computed value is placed in the warehouse, and every demand for an already-computed value is extracted from the warehouse rather than computed anew. Eduction thus reduces the overhead induced by the procedure calls needed for the computation of demands.

The GIPSY uses a generator-worker execution architecture. The IDS generated by the GIPC is interpreted by the generator following the eductive model of computation. The low-charge ripe sequential threads are evaluated locally by the generator. The higher-charge ripe sequential threads are evaluated on a remote worker.

As shown in Figure 4, the generator consists of two systems: the Intensional Demand Propagator (IDP) and the Intensional Value Warehouse (IVW). The IDP implements the demand generation and propagation mechanisms, and the IVW implements the warehouse. A set of semantic rules that outlines the theoretical aspects of the distributed demand propagation mechanism has been defined in [7]. The worker simply

consists of a “Ripe Function Executor” (RFE), responsible for the computation of the ripe sequential threads as demanded by the generator.

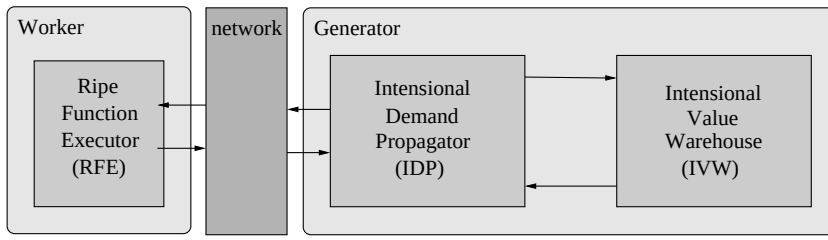

Fig. 4. Generator-Worker Execution Architecture

Intensional Demand Propagator (IDP) The IDP generates and propagates demands according to the data dependence structure (IDS) generated by the GIPC. If a demand requires some computation, the result can be calculated either locally or on a remote computing unit. In the latter case, the communication procedures (ICP) generated by the GIPC are used by the GEE to send the demand to the worker. When a demand is made, it is placed in a demand queue, to be removed only when the demand has been successfully computed. This way of doing provides a highly fault-tolerant system. One of the weaknesses of GLU is its inability to optimize the overhead induced by demand-propagation. The IDP will remedy to this weakness by implementing various optimization techniques:

- Data blocking techniques used to aggregate similar demands at run time, which will also be used at compile-time in the GIPC for automatic granularization of data and functions for data-parallel applications.
- The performance-critical parts (IDP and IVW) are designed as replaceable modules to enable run-time replacements by more efficient versions adapted to specific computation-intensive applications.
- Certain demand paths identified (at compile-time or run-time) as critical will be compiled to reduce their demand propagation overhead.
- Extensive compile-time and run-time rank analysis (analysis of the dimensionality of variables) [3]. This will be one of the major research topics during the implementation phase.

Intensional Value Warehouse (IVW) The second part of the GEE is the intensional value warehouse (IVW or “ivy” warehouse), which is simply implemented as a cache. The GEE uses the dataflow’s context tags to build a store of values that already have been computed (the IVW). One of the key concerns when using caches is the use of a garbage collecting algorithm adapted to the current situation. The use of a garbage collector configured or adapted to the current situation is of prime importance to obtain high performance.

Eduction has many guises. For example, eduction can be a simple one-time process, as in computing in Lucid, or a two-stage show, as in spreadsheet calculations. Also, in software versioning, some kinds of version selection correspond more to an aggregation process than to a selection process: All versions that correspond to the version description are chosen, and these are all coalesced into a single version. This has been implemented in the Lemur system using an intensional versioning technique [8].

A highly modular design and complete specification of generic software interfaces enables the GEE to accept other garbage collecting algorithms with minimal programming cost and replacement overhead. A thorough analysis of the different requirements of each intended application of intensional programming will enable us to identify a minimal set of garbage collecting techniques to be implemented and tested. We will also investigate the possibility of using multi-level warehouses enabling faster access to values accessed regularly and allowing out-of-date computed results to be stored on the file system. Rank analysis also greatly reduces the number of values stored in the warehouse by preventing the storage of values outside the dimensionality of the variables.

2.3 Run-time Interactive Programming Environment (RIPE)

The RIPE is a visual run-time programming environment enabling the visualization of a dataflow diagram corresponding to the Lucid part of the GIPSY program. The user can interact with the RIPE at run-time in the following ways:

- dynamically inspect the IVW;
- change the input/output channels of the program;
- recompile sequential threads;
- change the communication protocol;
- change parts of the GIPSY itself (e.g. garbage collector).

A graphical formalism to visually represent Lucid programs as multidimensional dataflow graphs had been devised in [7]. For example, consider the Hamming problem that consists of generating the stream of all numbers of the form $2^i 3^j 5^k$ in increasing order and without repetition. The following Lucid program solving this problem can be translated into a dataflow diagram, as shown in Figure 5:

```
H
where
  H = 1 fby merge(merge(2*H,3*H),5*H);
  merge(x,y)= if (xx<=yy) then xx else yy
  where
    xx = x upon (xx<=yy);
    yy = y upon (yy<=xx);
  end;
end;
```

Figure 6 represents the dataflow diagram defining the `merge` function. Such nested definitions will be implemented in the RIPE by allowing the user to expand or reduce sub-graphs, thus allowing the visualization of large scale Lucid definitions.

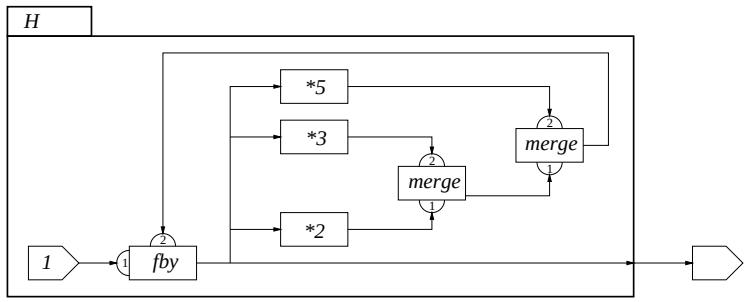

Fig. 5. Dataflow graph for the Hamming problem

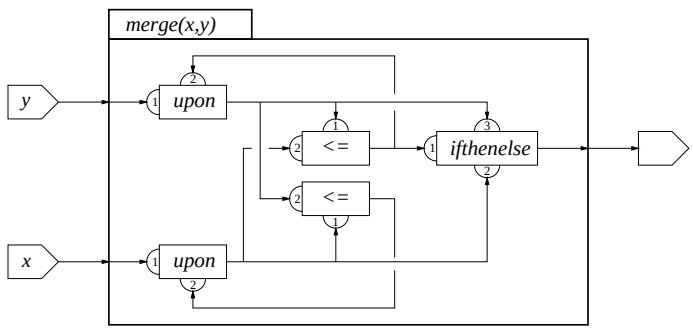

Fig. 6. Dataflow graph for the `merge` function

Using this visual technique, the RIPE will even enable the graphic development of Lucid programs, translating the graphic version of the program into a textual version that can then be compiled into an operational version. However, the development of this facility for graphical programming poses many problems whose solution is not yet settled. An extensive and general requirements analysis will be undertaken, as this interface will have to be suited to many different types of applications. There is also the possibility to have a kernel run-time interface on top of which we can plug-in different types of interfaces adapted to different applications.

3 Summary

It already has been proven that intensional programming can be used to solve critical problems such as tensor programming (TensorLucid [7]), distributed operating systems (the Web Operating System [6]) and software versioning (Lemur [8]). The solutions proposed by these three systems use intensional programming implemented in a demand-driven computation framework. This intensional programming framework has proven to provide a superior solution (in terms of expressiveness and inherent distributed computation possibilities) compared to all other techniques currently used to solve these problems. A plethora of other problems of intensional nature can be solved

within this framework. Although it has already theoretically proved its usefulness, Intensional programming is still in its infancy, mainly due to a lack of visibility.

The implementation of the GIPSY will enable us to realize the afore mentioned solutions in a unified framework and show the usefulness of the approach. A first prototype implementation of the GIPC and GEE is under way, whereas the implementation of the RIPE will be started after the experimental evaluation of the other two subsystems.

References

1. E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional, Declarative Programming*. Oxford University Press, London, 1995.
2. G. Babin, P.G. Kropf and H. Unger. A Two-Level Communication Protocol for a Web Operating System (WOS). In *Proceedings of IEEE Euromicro Workshop on Network Computing*, pp. 934–944, Västerås, Sweden, 1998.
3. C. Dodd. *Intensional Programming I*, chapter Rank analysis in the GLU compiler, pages 76–82. World Scientific, Singapore, 1996.
4. R. Jagannathan and C. Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.
5. R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. *Concurrency: Practice and Experience*, (1):63–83, 1997.
6. P.G. Kropf. Overview of the Web Operating System (WOS) project. In *Proceedings of the 1999 Advanced Simulation Technologies Conference (ASTC 1999)*, pages 350–356, San Diego, California, april 1999.
7. J. Paquet. *Scientific Intensional Programming*. Ph.D thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.
8. J. Plaice and W. W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 3(19):268–276, 1993.
9. P. Rondogiannis. *Higher-Order Functional Languages and Intensional Logic*. PhD thesis, Department of Computer Science, University of Victoria, Victoria, Canada, 1994.
10. W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

A Semantic definition of Lucid Programs

$$\mathbf{E_{cid}}: \frac{\mathcal{D}(id) = (\mathtt{const}, c)}{\mathcal{D}, \mathcal{P} \vdash id : c} \qquad \mathbf{E_{did}}: \frac{\mathcal{D}(id) = (\mathtt{dim})}{\mathcal{D}, \mathcal{P} \vdash id : id}$$

$$\mathbf{E_{opid}}: \frac{\mathcal{D}(id) = (\mathtt{op}, f)}{\mathcal{D}, \mathcal{P} \vdash id : id} \qquad \mathbf{E_{fid}}: \frac{\mathcal{D}(id) = (\mathtt{func}, id_i, E)}{\mathcal{D}, \mathcal{P} \vdash id : id}$$

$$\mathbf{E_{vid}}: \frac{\mathcal{D}(id) = (\mathtt{var}, E) \qquad \mathcal{D}, \mathcal{P} \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash id : v}$$

$$\mathbf{E_{op}}: \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{op}, f) \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)}$$

$$\mathbf{E_{fct}}: \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{func}, id_i, E') \qquad \mathcal{D}, \mathcal{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : v}$$

$$\mathbf{E_{c_T}}: \frac{\mathcal{D},\mathcal{P} \vdash E : \mathit{true} \qquad \mathcal{D},\mathcal{P} \vdash E' : v'}{\mathcal{D},\mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v'}$$

$$\mathbf{E_{c_F}}: \frac{\mathcal{D},\mathcal{P} \vdash E : \mathit{false} \qquad \mathcal{D},\mathcal{P} \vdash E'' : v''}{\mathcal{D},\mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v''}$$

$$\mathbf{E_{tag}}: \frac{\mathcal{D},\mathcal{P} \vdash E : \mathit{id} \qquad \mathcal{D}(\mathit{id}) = (\texttt{dim})}{\mathcal{D},\mathcal{P} \vdash \#E : \mathcal{P}(\mathit{id})}$$

$$\mathbf{E_{at}}: \frac{\mathcal{D},\mathcal{P} \vdash E' : \mathit{id} \qquad \mathcal{D}(\mathit{id}) = (\texttt{dim}) \qquad \mathcal{D},\mathcal{P} \vdash E'' : v'' \qquad \mathcal{D},\mathcal{P}\dagger[\mathit{id} \mapsto v''] \vdash E : v}{\mathcal{D},\mathcal{P} \vdash E\ @E'\ E'' : v}$$

$$\mathbf{E_{w}}: \frac{\mathcal{D},\mathcal{P} \vdash Q\ :\ \mathcal{D}',\mathcal{P}' \qquad \mathcal{D}',\mathcal{P}' \vdash E : v}{\mathcal{D},\mathcal{P} \vdash E\ \texttt{where}\ Q : v}$$

$$\mathbf{Q_{dim}}: \frac{}{\mathcal{D},\mathcal{P} \vdash \texttt{dimension}\ \mathit{id}\ :\ \mathcal{D}\dagger[\mathit{id} \mapsto (\texttt{dim})], \mathcal{P}\dagger[\mathit{id} \mapsto 0]}$$

$$\mathbf{Q_{id}}: \frac{}{\mathcal{D},\mathcal{P} \vdash \mathit{id} = E\ :\ \mathcal{D}\dagger[\mathit{id} \mapsto (\texttt{var}, E)], \mathcal{P}}$$

$$\mathbf{Q_{fid}}: \frac{}{\mathcal{D},\mathcal{P} \vdash \mathit{id}(\mathit{id}_1, \ldots, \mathit{id}_n) = E\ :\ \mathcal{D}\dagger[\mathit{id} \mapsto (\texttt{func}, \mathit{id}_i, E)], \mathcal{P}}$$

$$\mathbf{QQ}: \frac{\mathcal{D},\mathcal{P} \vdash Q\ :\ \mathcal{D}',\mathcal{P}' \qquad \mathcal{D}',\mathcal{P}' \vdash Q' : \mathcal{D}'',\mathcal{P}''}{\mathcal{D},\mathcal{P} \vdash Q\ Q'\ :\ \mathcal{D}'',\mathcal{P}''}$$