Global Script Program Calculus

Jonathan Cast

February 19, 2019

Contents

Chapter 1

Introduction and General Remarks

This document specifies the *Global Script Program Calculus*, a purely-functional, completely machine-independent program calculus designed to support the Global Script Type-Setting System.

1.1 Excursus on the Term 'Programming Language'

The term 'programming language' has an odd, and in the opinion of the developer of this system deleterious, definition in normal usage. 'Programming language' clearly derives from the verb 'program', (rather than the noun); it clearly means 'language for the act of programming'. But the verb 'program' is a transitive verb: 'program the i686 processor', 'program the fuel injection system', 'program the accounting system', etc. To *program* a machine is to make it do what the programmer wishes; the act of programming makes no sense except in reference to a machine. So 'machine-independent programming language' is, properly speaking, a nonsensical concept.

Originally, this term simply meant 'programming language which targets multiple machines' (or, rather, usually, 'family of programming languages indexed by multiple machines'), which really means (if it means anything) 'programming language which targets a family of machines sharing similar capabilities'. Typically this class of machines encompases most of the machines typically encountered by programmers (but not most programmable machines in use!),[1] so the myth has grown up that typical languages 'support all computers'; and even that languages lacking I/O facilities are not Turing-complete![2]

[1] Most software is actually embedded software, which lacks access to the familiar keyboard / screen / mouse facilities assumed by the typical definition of 'general-purpose' language.

[2] The Intercal manual (implicitly) makes this claim.

Meanwhile, 'programming language' has come to mean something like 'algorithmic language', i.e., methods for algorithmic computation, variable naming, function definition, program partitioning, etc. I/O 'isn't part of the language' or 'shouldn't be part of the language' (!).[3] The problem with this position[4] is that it justifies the neglect of I/O that programming language designers have traditionally lavished on it. Almost every language has genuinely wretched I/O facilities, precisely because of the second-class status assigned to I/O by the modern concept of 'programming language'. Global Script thus includes I/O 'in the language', not only because that is the only way to give a denotation for I/O, but because I/O deserves the same careful design as any other aspect of the language.

The Global Script Program (not programming!) Calculus is a general syntax for writing programs, together with rules for manipulating them — consisting of a static semantics (type system), denotational semantics,[5] operational semantics (only for expression evaluation), and formal semantics (equational theory). The Global Script Program Calculus provides an 'algorithmic language' in its expression syntax, data types, and operational semantics; it also provides a module system, including support for abstract data types and for encapsulating business logic, allowing for extensions, libraries, and allowing the large-scale structure of a program to be expressed within the language. It thus provides everything a programming language needs, *except* the environment-specific features. In that sense, it is a genuinely environment-independent language. But not a *programming* language, because you can't program anything with it!

However, the Global Script Program Calculus can easily be extended with operations for programming a specific machine. To allow libraries to be universal, we use the classic pure-functional approach of adding a new type for programs in each environment, with operations that construct / return such programs, including so-called *combinators* that take parts of programs and combine them into larger programs. Then, the type system distinguishes between different environments, allowing libraries to be used cross-environment, but not allowing subprograms for one environment to be invoked incorrectly within another environment which doesn't provide the operations they need.

The type of programs for each environment, together with its operations, is called a 'programming language' in Global Script, and, in general, the term 'language' is used freely for the API for any specific area.

Currently, the major programming languages provided include:

- Global Script Log — a logical markup language;
- Global Script Set — a page description language;
- Global Script Type — ditto, but for Unix terminals;

[3] Thus excluding denotational semantics from 'the language'....

[4] Beyond the fact that you can't interpret I/O operations in an arbitrary model of a language that lacks them!

[5] Meaningful meaning...

- IBIO — a concurrent stream I/O language, suitable for writing Unix filters (and, although not designed for them, compilers);
- Cord — a relational database language;
- Dance — an FRP GUI language; and
- Guisen — a procedural GUI language.

1.2 Syntax

This document contains no formal syntax for the language; instead, the intention is that the mapping from the abstract syntax given here back to concrete syntax be simple enough to not need a repetitive specification.

Global Script has a handful of special symbols: , ; . and delimiters such as ( ) [ ] { } ⟨ ⟩ ⟪ ⟫; otherwise, most non-whitespace symbols are treated as being part of identifiers (or keywords).

An identifier consists of a sequence of one or more components separated by . characters (with no intervening whitespace); the three types of components are

- *alphanumeric* components, which consist of a lower-case letter followed by 0 or more alphanumeric letters;
- *numeric* components, which consist of a sequence of 1 or more decimal digits (two numeric components with the same value, e.g. 01 and 1, are considered identical); and
- *symbolic* components, which consist of a sequence of 1 or more symbols, other than , ; . or a delimiter.

A symbolic component may only be the final component in an identifier.

1.3 Type System

Global Script has a standard natural deduction-based type system, based on *judgment*s of the form

$$\Gamma \vdash \text{conclusion}$$

The context Γ maps several classes of variables (type variables, object variables, views, etcw.) to various sorts of meta-data (kinds, types, implicit arguments, macro properties, etc.). Nevertheless, the context will generally be treated as a single object, except in the denotational semantics, where it will be split into a kind context Δ and a type context Γ.

1.4 Models and Denotations

Note: 'semantics' means 'meaning'; specifically, the kind of detailed meaning that might be found in a language specification. 'Denotation' means 'meaning'; specifically, the kind of formal meaning that might be found in a language specification. So 'denotational semantics' is redundant. End note.

A *model* of Global Script is a directed-complete partial order-enriched category $\mathcal{C}$ with:

- Finite products indexed by sets of Global Script identifiers (note that this implies in particular the existence of a terminal object $\star$);
- Finite coproducts indexed by sets of Global Script identifiers (note that this implies in particular the existence of an initial object ϵ);
- Exponentials;
- An existential quantifier, which is a functor $\exists : \mathcal{C}^\kappa \to \mathcal{C}$, which is the left adjoint of the constant functor $K : \mathcal{C} \to \mathcal{C}^\kappa$, for κ at least the denotations of the kind environments as defined below; and
- A universal quantifier, which is a functor $\forall : \mathcal{C}^\kappa \to \mathcal{C}$ which is the right adjoint of of the constant functor $K : \mathcal{C} \to \mathcal{C}^\kappa$, for κ at least the denotations of the kind environments as defined below;

together with a designated *lifting monad*, which is a monad $\lfloor — \rfloor$ on $\mathcal{C}$ together with a natural transformation $\perp : K\star \to \lfloor — \rfloor$ so that $\perp_\alpha \circ \star_\beta$ is the least homomorphism $: \beta \to \lfloor \alpha \rfloor$ for all objects α, β.

Note: we need a couple of other things:

- A 'strength' $: \exists(K\alpha \times F) \to \alpha \times \exists F$;
- A way to 'lift' $\exists$ and $\forall$ to functors with more than one argument (eliminating only the last one).

These seem like they should be definable in any category, though. TODO: Will need to confirm / deny that.

For the purpose of the denotational semantics, contexts will be divided into two pieces: a kind environment Δ and a type environment Γ. The kind environment Δ can be thought of as a mapping from type variables to kinds; the denotation of a kind is a category, and the denotation of a type environment Δ, written $Den[\![\Delta]\!]$, is

$$\prod_{t \in Dom\Delta} \mathbb{K}[\![\Delta(t)]\!]$$

The denotation of the type environment Γ, which will be written $Den[\![\lambda\Delta.\Gamma]\!]$, will be a functor $F : Den[\![\Delta]\!] \to \mathcal{C}$.

1.5 Equational Theory

The equational theory of Global Script, like the type system, consists of judgments of the form

$$\Gamma \vdash P \tag{1.1}$$

where, however, Γ is a sequence of type assignments and *equational propositions* and P is an equational proposition. If you like, you can read this as 'P is deducible from Γ'.

An equational proposition is formed from the propositional connective $\wedge$, universal quantifiers $\forall$ (over Global Script type assignments, so $\forall\, 'x :: \tau$ where x is a Global Script variable and τ its type), and seven forms of atoms:

- *Equations* of the form $e_0 = e_1$, between two expressions of the same type;
- *Pattern match success assertions* of the form $p \propto e$, read 'p *matches* e';
- *Pattern match failure assertions* of the form $p \propto e \downarrow$, read 'p *fails to match* e';
- *Pattern match divergence assertions* of the form $p \propto e \uparrow$, read 'the match of p against e *diverges*';
- *Generator success assertions* of the form g, read 'g *succeeds*';
- *Generator failure assertions* of the form $g \downarrow$, read 'g *fails*'; and
- *Generator divergence assertions* of the form $g \uparrow$, read 'g *diverges*'.

There is a technical ambiguity wether the propositions $p \propto e$, $p \propto e \downarrow$, and $p \propto e \uparrow$ are about the pattern p and expression e or the generator $p \propto e$, but since the meaning is the same in either case the ambiguity does not matter, and will be clarified in those cases where which is meant is not immediate.

1.6 Operational Model

1.7 Implementations

An *implementation* of a Global Script program is a program or machine that stores the program internally. On request, it will evaluate that program to WHNF and, if successful, reports the value; if the value is a function it will further permit Global Script expressions of the correct type to be supplied as arguments, and will calculate and report their values. If the value is a data value, of sum or product type, it will allow the components to be evaluated in the same way.

Note that an implementation is necessarily interactive and works with a client; implementations on general-purpose computers will generally be libraries rather than complete programs. Nothing in this chapter is intended to disallow

the creation of interpreters that take programs, evaluate them to normal form, and print the results; however, such a program, because it attaches additional semantics beyond that of the pure calculus, is properly an implementation of a language based on the calculus and not of the Global Script program calculus itself. Similar considerations apply to an implementation that assigns semantics to Global Script expressions to encode I/O, database access, web servers, GUIs, or type-setting and allow those to be programmed in Global Script.

A *correct* implementation of the Global Script program calculus is an implementation that, given a program or component thereof, exhibits the following behavior:

- Correctly reports that the expression has a WHNF if it has one, and provides correct access to its components or gives the value if it is a primitive;
- Correctly reports that the expression has no WHNF if it lacks one; or
- Runs forever if the expression has no WHNF.

Note that it is impossible to realize a correct implementation of the Global Script program calculus as a physical object, or to implement one on a general-purpose computer. We thus say that Global Script is *un-implementable*. Any two expressions of the same type that lack a WHNF are equivalent according to the calculus; however, two such expressions may exhibit different behavior when run by such an implementation, if it is able to determine that the one expression lacks a WHNF but is unable to make that determination for the other.

A *partially correct* implementation of the Global Script program calculus is an implementation that, given a program or component thereof, exhibits the following behavior:

- Correctly reports that the expression has a WHNF if it has one, and provides partially correct access to its components or gives the value if it is a primitive;
- Correctly reports that the expression has no WHNF if it lacks one;
- Runs forever if the expression has no WHNF; or
- Reports that it is unable to find a WHNF for the given expression.

Note that two expressions that are equivalent according to the calculus may exhibit different behavior when run by such an implementation, even if they have a WHNF, because the implementation may be able to find a WHNF for one program but unable to do so for the other.

Chapter 2

Object Expressions

The judgment form for an expression e in context Γ is

$$\Gamma \vdash e :: \tau \tag{2.1}$$

for some type $\Gamma \vdash \tau :: \kappa$.

The denotation of an expression will be a natural transformation $\mathbb{E}[\![\Delta, \Gamma \vdash e :: \tau]\!] : Den[\![\Delta \vdash \Gamma \text{ context}]\!] \to \mathbb{T}[\![\Delta \vdash \tau :: \kappa]\!]$.

2.1 Variables

The judgment form for a variable v in context Γ is

$$\Gamma \vdash v \ \overline{\tau_{0,i} :: \kappa_i}^{\,n}_{i=0} \ \overline{a_i :: \tau_{1,i}}^{\,m}_{i=0} :: \tau, \tag{2.2}$$

where

- $\overline{\tau_{0,i} :: \kappa_i}^{\,n}_{i=0}$ is a sequence of *implicit type arguments* and their kinds,
- $\overline{a_i :: \tau_{1,i}}^{\,m}_{i=0}$ is a sequence of *implicit value arguments* and their types, and
- τ is the (monomorphic) type of the variable term.

The meta-variable v can stand for a simple identifier (this case will be written x, x_i, etc.), or for an application (see below).

A variable has no denotation, but see section 2.1.5 below.

2.1.1 Variables Without implicit Declarations

The derivation rule for simple identifiers without **implicit** declarations is

$$\frac{'x :: \overline{\forall\, '\alpha_i :: \kappa_i .}^{\,n}_{i=0} \tau_1[\,\overline{\alpha_i}^{\,n}_{i=0}\,] \in \Gamma \qquad \textbf{implicit}\ 'x \notin \Gamma \qquad \overline{\Gamma \vdash \tau_{0,i} :: \kappa_i}^{\,n}_{i=0}}{\Gamma \vdash x\ \overline{\tau_i :: \kappa_i}^{\,n}_{i=0} :: \tau_1[\,\overline{\tau_{0,i}}^{\,n}_{i=0}\,]}$$

2.1.2 Variables With implicit Declarations

The derivation rule for simple identifiers with **implicit** declarations is

$$\frac{{}'x :: \overline{\forall {}'\alpha_i :: \kappa_i.}^{\,n}_{i=0}\ \overline{\tau_{1,i}\big[\overline{\alpha_j}^{\,n}_{j=0}\big] \to}^{\,m}_{i=0}\ \tau_2\big[\overline{\alpha_i}^{\,n}_{i=0}\big] \in \Gamma \qquad \textbf{implicit}\ {}'x = \overline{e_i}^{\,m}_{i=0} \in \Gamma \qquad \overline{\Gamma \vdash \tau_{0,i} :: \kappa_i}^{\,n}_{i=0}}{\Gamma \vdash x\ \overline{\tau_{0,i} :: \kappa_i}^{\,n}_{i=0}\ \overline{e_i :: \tau_{1,i}\big[\overline{\tau_{0,j}}^{\,n}_{j=0}\big]}^{\,m}_{i=0} :: \tau_2\big[\overline{\tau_{0,i}}^{\,n}_{i=0}\big]}$$

2.1.3 Explicit Type Applications

Explicit type arguments only serve as documentation.

$$\frac{\Gamma \vdash v\ \tau_{0,0} :: \kappa_0,\ \overline{\tau_{0,i} :: \kappa_i}^{\,n}_{i=1}\ \overline{e_i :: \tau_{1,i}}^{\,m}_{i=0} :: \tau}{\Gamma \vdash v\_(\textbf{type}\ \tau_{0,0})\ \overline{\tau_{0,i} :: \kappa_i}^{\,n}_{i=1}\ \overline{e_j :: \tau_{1,i}}^{\,m}_{i=0} :: \tau}$$

2.1.4 Explicit Applications

Explicit expression applications replace the implicit value arguments, negating them and removing them from the expression.

$$\frac{\Gamma \vdash v\ \overline{\tau_{0,i} :: \kappa_i}^{\,n-1}_{i=0}\ e_0 :: \tau_{1,0},\ \overline{e_i :: \tau_{1,i}}^{\,m-1}_{i=1} :: \tau \qquad \Gamma \vdash e :: \tau_{1,0}}{\Gamma \vdash v\_e\ \overline{\tau_{0,i} :: \kappa_i}^{\,n-1}_{i=0}\ \overline{e_i :: \tau_{1,i}}^{\,m-1}_{i=1} :: \tau}$$

2.1.5 Variables Used as Expressions

If the implicit arguments to a variable all type-check, the variable can be used as an expression:

$$\frac{\Gamma \vdash v\ \overline{\tau_i :: \kappa_i}\ \overline{e_i :: \tau_{1,i}}^{\,m-1}_{i=0} :: \tau \qquad \overline{\Gamma \vdash e_i :: \tau_{1,i}}^{\,m-1}_{i=0}}{\Gamma \vdash v :: \tau}$$

2.2 Functions

2.2.1 Function Literals

Lambda terms have the rule:[1]

$$\frac{\Gamma \vdash p :: \tau_1 \triangleright \Gamma' \qquad \Gamma, \Gamma', \vdash e :: \tau_2}{\Gamma \vdash \lambda p.e :: \tau_1 \to \tau_2}$$

The pattern p must be a lax or strict (not monoidal) pattern; see Section 3.2, Classes of Patterns. BUG: λ takes *multiple* patterns (0 or more). End BUG.

[1] See Chapter 3, Patterns

2.2.2 Applications

Applications have the rule:

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$$

2.3 for Expressions

for expressions are used to make local definitions within an expression. The syntax is **for** ⟨generators⟩. ⟨body expression⟩. The generators in a **for** are non-recursive, but cannot shadow variables defined in the same **for**.

In the simplest case, the generators in a **for** are all let and (non-monoidal) match generators.

$$\frac{\Gamma \vdash \overline{g_i} \triangleright \Gamma' \qquad \Gamma, \Gamma', \vdash e :: \tau}{\Gamma \vdash (\textbf{for}\ \overline{g_i}.\ e) :: \tau}$$

for can also be used to name intermediate values inside a monad. E.g. **for** $'x \leftarrow e_0.\ e_1$. The type rule for this is

$$\frac{\Gamma \vdash \textbf{module}\ monad.c\ m \qquad \Gamma \vdash \overline{g_i} \triangleright_m \Gamma' \qquad \Gamma, \Gamma', \vdash e :: m\ \tau}{\Gamma \vdash \textbf{for}\ \overline{g_i}.\ e :: m\ \tau}$$

As might be expected, the monad structure to use in de-sugaring this syntax can be over-ridden:

$$\frac{\Gamma \vdash mon :: monad.t\ m \qquad \Gamma \vdash \overline{g_i} \triangleright_m \Gamma' \qquad \Gamma, \Gamma', \vdash e :: m\ \tau}{\Gamma \vdash \textbf{for}\ @mon\ (\overline{g_i})\ e :: m\ \tau}$$

The sequence of generators inside the () in a **for** can either be a single generator or a sequence of 0 or more generators terminated by semi-colons. In the special case of a single **rec** generator, the syntax **for** (**rec**($\overline{g_i}$)) can be abreviated to **for rec** ($\overline{g_i}$).

2.4 Branch Expressions

An **analyze** expression, Syntactically, **analyze** $\overline{expr_i}$ $\overline{\textbf{case}\ \overline{p_{ji}}.\ body_j}$, takes a sequence of scrutinee expressions and matches them against the patterns in the cases, top-to-bottom and left-to-right. If matching a scrutinee against any pattern diverges or no case matches evaluation diverges; otherwise the **analyze** expression evaluates to the body of the first matching case. Note: if a case matches and matching the scrutinees against the patterns in a previous case diverges, evaluation diverges. If a case matches and matching the scrutinees against a subsequent case diverges, the evaluation proceeds with the body of the matching case.

This syntax has a 'dangling else' problem when a **case** ends with an **analyze** expression; this is resolved as usual by associating each **case** to the nearest preceding **analyze** expression. This resolution can be over-ridden by putting parentheses around the inner **analyze** expression or its enclosing **case**.

2.4.1 Cases

The judgment form for **case** terms is:

$$\Gamma; \vdash \textbf{case}\ \overline{p_i}.\ e :: \overline{\tau_i} \to \tau$$

Single cases type thus:

$$\frac{\Gamma \vdash \tau :: *\qquad \Gamma \vdash e :: \tau}{\Gamma \vdash \textbf{case}.\ e :: \to\ \tau}$$

$$\frac{\Gamma \vdash \tau :: *\qquad \Gamma \vdash p_0 :: \tau_0 \triangleright \Gamma'\qquad \Gamma, \Gamma', \vdash \textbf{case}\ \overline{p_i}.\ e :: \overline{\tau_i} \to \tau}{\Gamma \vdash \textbf{case}\ p_0,\ \overline{p_i},.\ e :: \tau_0,\ \overline{\tau_i}, \to \tau}$$

NB: p_0, $\overline{p_i}$ should define disjoint sets of variables. This has implications. The principle is — hrm. So we have a can-be-bound-by set for a pattern — in a context — and then an is-bound-by function for sequences/sets of patterns. Usually these are the same. However, certain patterns do *not* bind variables that can be bound by other patterns (whether they are or not). But that all goes in the Patterns chapter (Chapter 3).

Chapter 3

Patterns

The judgment form for patterns is

$$\Gamma \vdash \overline{p_i :: \tau_i} \triangleright \Gamma' \tag{3.1}$$

3.1 Concatenating Sequences of Patterns

$$\frac{\Gamma \vdash \overline{p_i :: \tau_i} \triangleright \Gamma'_1 \qquad \Gamma \vdash \overline{p_j :: \tau_j} \triangleright \Gamma'_2}{\Gamma \vdash \overline{p_i :: \tau_i}, \overline{p_j :: \tau_j}, \triangleright \Gamma'_1, \Gamma'_2,}$$

$(i \in [0, n),\ j \in [n, n + m))$. The bound variables of the patterns must all be distinct. TODO: need a way to handle module patterns **module** $'m$, for which the bound variables are selected to be distinct from any other patterns in the same sequence. The rule is basically: calculate the set of variables which *could* be bound by any given pattern. Then each **module** $'m$ pattern brings into scope those variables which could be bound by it, and which cannot be bound by any other pattern. This only applies to **module** patterns with no export list, or whose export list contain .. . See Chapter 6, Modules. End TODO.

3.2 Classes of Patterns

- *Lax* patterns are precisely these:
  - Pattern variables $'x$;
  - Wildcard patterns _;
  - Module patterns where:
    - there is no export list, e.g. **module** $'m$,
    - there is a simple export list, e.g. **module** $'m.(x,\ y,\ z,)$,

    - there is a complex export list, e.g. $\textbf{module}\ 'm.\begin{pmatrix} x &= p_0, \\ y &= p_1, \\ z &= p_2, \end{pmatrix}$, where all of the patterns given for the members are themselves lax;
  - Lazy patterns $\sim p$; and
  - Parallel patterns $\parallel p$.
- *Strict* patterns are precisely those patterns of the form $!p$.
- *Monoidal* patterns are precisely those patterns that are neither lax nor strict.

3.3 Specific patterns

Pattern variables Pattern variables have the syntactic form $'x$, the ASCII apostrophe followed by a variable name. Pattern variables where the variable is an infix operator need to be enclosed in parentheses, e.g., $('+)$

$$\frac{\Gamma \vdash \tau :: *}{\Gamma \vdash\ 'x :: \tau \triangleright x :: \tau}$$

Wildcard Patterns

$$\frac{\Gamma \vdash \tau :: *}{\Gamma \vdash \_ :: \tau \triangleright}$$

Views The typing judgment for views has the form $\Gamma \vdash \textbf{view}\ v :: \overline{\tau_i}; \tau$, for $i \in [0, n)$. This means:

- v is a view,
- v has arrity n,
- v's arguments have types τ_i, respectively, and
- v's result has type τ.

Note that, if v is both a view and a function, *it need not have the same type in both cases*. Nor, if it does have the same type, need its definition as a view and as a function have any specific relation. Nevertheless giving them the same type and relating the definitions in some (documented) way is strongly recommended. Global Script does not draw a distinction between constructors and views.

$$\frac{\Gamma \vdash \textbf{view}\ v :: \overline{\tau_i}; \tau \qquad \Gamma \vdash \overline{p_i :: \tau_i} \triangleright \Gamma'}{\Gamma \vdash v\ \overline{p_i} :: \tau \triangleright \Gamma'}$$

TODO: How do you declare views? How do views and constructors interract, exactly? End TODO.

TODO: Existential types. End TODO.

Language TODO: Expression arguments to views (e.g., $n + m$ patterns). End TODO.

3.4 Reasoning at the Source Level

Source-level reasoning about programs uses three predicates:

- $p \propto e \uparrow$, read "matching pattern p against expression e diverges".
- $p \propto e \downarrow$, read "matching pattern p against expression e fails".
- $p \propto e$, read "matching pattern p against expression e succeeds".

Note that, for a given pattern p and expression e, precisely one of these predicates will hold.

If $p \propto e$, the syntax $\mathbb{P}[\![p \propto e]\!]$ will be used to denote the value environment induced by the pattern match; $\mathbb{P}[\![p \propto e]\!]e1$ will be used to denote $e1$ with the values bound by p:

$$\mathbb{E}[\![\mathbb{P}[\![p \propto e]\!]e1]\!]\eta = \mathbb{E}[\![e1]\!]\{\eta; \mathbb{P}[\![p \propto e]\!]\}$$

If $p \propto e \uparrow$ or $p \propto e \downarrow$, $\mathbb{P}[\![p \propto e]\!]$ and $\mathbb{P}[\![p \propto e]\!]e1$ will be undefined.

We have a few basic cases:

$$\frac{}{'x \propto e}$$

For a basic view pattern $v\ \overline{p_i}$, when matching against an expression e, use a pattern matching algorithm as follows:

- First consider the expression $'d = \mathit{match}\ v\ \mathit{bool.false}\ (\lambda\ \overline{'x_i}.\mathit{bool.true})\ e$.
  - If $d = \bot$, $v\ \overline{p_i} \propto e \uparrow$.
  - If $d = \mathit{bool.false}$, $v\ \overline{p_i} \propto e \downarrow$.

  Otherwise, $d = \mathit{bool.true}$.
- Second, if $d = \mathit{bool.true}$, let $t_0 = \mathit{match}\ v\ \mathbf{error}\ (\lambda\ \overline{'x_i}.(\!|\ \overline{x_i}, |\!|)\ e$. Let $\overline{x_i = \#i\ t_0}$ be the components of t_0.
  - If, for any $'k \in [0, n)$, $\forall\ 'i \in [0, k).\ p_i \propto x_i$ and $p_k \propto x_k \uparrow$, $v\ \overline{p_i} \propto e \uparrow$.
  - If, for any $'k \in [0, n)$, $\forall\ 'i \in [0, k).\ p_i \propto x_i$ and $p_k \propto x_k \downarrow$, $v\ \overline{p_i} \propto e \downarrow$.
  - Otherwise, for all $'i \in [0, n)$, $p_i \propto x_i$. In this case, $v\ \overline{p_i} \propto e$ and $\mathbb{P}[\![v\ \overline{p_i} \propto e]\!]$ is the concatenation of $\overline{\mathbb{P}[\![p_i \propto x_i]\!]}$.

Chapter 4

Generators

The judgment form for generators is:

$$\Gamma \vdash g \triangleright \Gamma'$$

g can be a single generator, or a sequence of (semicolon-terminated) generators. This is read "In type environment Γ, the generator(s) g is (are) well-typed and induce(s) the type environment Γ'.

Monoidal generators can fail at run time, which is indicated by using a judgment of the form

$$\Gamma \vdash g \triangleright^{?} \Gamma'$$

Monadic generators run in a particular monad, which is indicated by using a judgment of the form

$$\Gamma \vdash g \triangleright_m \Gamma'$$

And, finally, there are *monadic monoidal* generators:

$$\Gamma \vdash g \triangleright_m^{?} \Gamma'$$

Some simple generators are:

$$\frac{\Gamma \vdash p :: \tau \triangleright \Gamma' \qquad \Gamma \vdash e :: \tau}{\Gamma \vdash p \propto e \triangleright \Gamma'}$$

$$\frac{\Gamma \vdash p :: \tau \triangleright^{?} \Gamma' \qquad \Gamma \vdash e :: \tau}{\Gamma \vdash p \propto e \triangleright^{?} \Gamma'}$$

Chapter 5

Types

Chapter 6

Modules

Chapter 7

Overloading

Chapter 8

IDMC

IDMC is more complicated than other aspects of the language, because each QLO can define its own syntax and type system for that syntax.

So the judgement form for QLO bits is:

$$\Gamma \vdash \langle \text{text} \rangle :\!:_{qlo} \tau$$

$\langle$text$\rangle$ is a bit of marked-up text; *qlo* is a QLO.

The master inference rule for QLOs then is:

$$\frac{\Gamma \vdash \langle \text{text} \rangle :\!:_{qlo} \tau}{\Gamma \vdash qlo\{\langle \text{text} \rangle\} :: \tau}$$

The curly braces {} can be replaced with any delimiters you want (only in this construct).

Expression interpolations:

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \S(e) :\!:_{qlo} \tau}$$

Text interpolations:

$$\frac{\Gamma \vdash \langle \text{text} \rangle :\!:_{qlo} \tau}{\Gamma \vdash \S\{\langle \text{text} \rangle\} :\!:_{qlo} \tau}$$

Variable/macro interpolations Macro interpolations have the form

$$\S x @(\textbf{type}\, \tau_0) \ldots @(\textbf{type}\, \tau_{n-1}) @ arg_0 \ldots @ arg_{m-1}\, arg_m \ldots arg_{m+n-1}$$

where each *arg*, whether optional or required, has the form {$\langle$text$\rangle$} or (*exp*).

The judgment form for a macro interpolation is

$$\Gamma \vdash v :: \overline{\tau_i :: \kappa_i};\ \overline{a_j :: \tau_j};\ \overline{qlo_k};\ \tau$$

where each qlo_k is either a QLO name or _. The empty sequence at the end of the sequence $\overline{qlo_k}$ is equivalent to an infinite sequence of _s.

If Γ contains no **macro** declaration for x:

$$\frac{\Gamma \vdash x :: \overline{\tau_i :: \kappa_i}; \overline{a_j :: \tau_j}; \tau \qquad \overline{\Gamma \vdash \tau_i :: \kappa_i}}{\Gamma \vdash \S x ::_{qlo} \overline{\tau_i :: \kappa_i}; \overline{a_j :: \tau_j}; \tau}$$

Otherwise:

$$\frac{\textbf{macro}\ x\ \overline{qlo_k} \in \Gamma \qquad \Gamma \vdash x :: \overline{\tau_i :: \kappa_i}; \overline{a_j :: \tau_j}; \tau \qquad \overline{\Gamma \vdash \tau_i :: \kappa_i}}{\Gamma \vdash \S x ::_{qlo} \overline{\tau_i :: \kappa_i}; \overline{a_j :: \tau_j}; \overline{qlo_k}; \tau}$$

Chapter 9

Programs

The judgment form for whole programs is

$$\vdash \langle \overline{g_i};\ d \rangle :: \langle \Gamma;\ \tau \rangle$$

The type environment must be empty because whole programs include the entire library and so can have no free variables.

A program consists of a sequence of generators and a §defndocument; ideally, Global Script implementations should assemble programs from separately maintained components, including the §defnstandard library (see Chapter 11).[1] A program's value is the value of its document.

TODO: The standard library and the standard libraries of languages like IBIO and CORD are quite magic in bringing things into scope there's no way to define in the language. End TODO.

The generators in a program (at the top level) must be:

- Match generators $p \propto e$ where p is lax;
- Let generators $'f\ \overline{p_i} = e$;
- Type signatures $x :: \sigma$; and
- Recursive groups **rec** $(\overline{g_i})$ where all the g_i are otherwise legal at the top level.

Implementations may and are likely to further restrict accepted top-level generators; they must support these cases:

- Match generators $p \propto e$ defining variables in a single module. (When defining multiple variables, the best way to ensure this is to use a generator of the form $module\,'m.(\langle\text{export list}\rangle) \propto e$).
- Functions defined by a single let equation $'f\ \overline{p_i} = e$. Note that using monoidal patterns in such an equation must be legal, but is sub-optimal programming practice as it must be an error for any pattern in the head to fail.

[1] So including multiple components in the program isn't really optional.

The inference rule from programs is

$$\frac{\vdash \overline{g_i} \triangleright \Gamma \qquad \Gamma \vdash d :: \tau}{\vdash \langle \overline{g_i}; d \rangle :: \langle \Gamma; \tau \rangle}$$

Chapter 10

Documents

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash e :: \tau}$$

$$\frac{\Gamma, \Gamma' \vdash \overline{g_i} \triangleright \Gamma' \qquad \Gamma, \Gamma' \vdash e :: \tau}{\Gamma \vdash e\ (\textbf{where}\ \overline{g_i}) :: \tau}$$

The parentheses around the **where** clause can be omitted when there is only one generator, in which case that generator must not have a trailing ;, as usual. The standard limits on the generators legal in a **where** clause apply

TODO: Have we said what the 'standard limits' on **where** clauses are yet? End TODO.

A §defndocument is an expression together with an optional **where** clause. The document $e\ (\textbf{where}\ \overline{g_i})$ is equivalent to the expression $\textbf{for rec}\ (\overline{g_i})\ e$, as might be expected. (Except that more generators are legal in a **for** than in a **where**.)

Chapter 11

Standard Library

Chapter 12

Examples

12.1 Fibionacci Numbers

$'fibs \propto 0 : 1 : map_2\ @by.zip\ (+)\ fibs\ (drop\ 1\ fibs);$

12.2 Prime Numbers

$'primes = w\ 2\ (repeat\ true)$ (**where**
$'w\ !'n\ (\text{false} : 'bs) :\!- w\ (n + 1)\ bs;$
$'w\ !'p\ (\text{true} : 'bs) :\!- p : w\ (p + 1)\ (set\ (chunksof\ p \circ elems \circ last)\ false\ bs);$
);

12.3 `gstype` Hello, World

qq{`Hello, world!\n`}

12.4 `gsdraw` Hello, World

text str{Hello, world!}

12.5 Dance Hello, World

text str{Hello, world!}

12.6 IBIO `echo`

```
for 'as ← getM env.args. send $ concat (intersperse qq{ } as) <> qq{\n}
```

12.7 IBIO `cat`

```
for 'as ← getM env.args. analyze as.
  case nil. cat
  (case _. foreachM $ λ 'a.
    for 'eif ← file.open o/r/ $ file.name.in a. analyze eif.
          case left 'e. abend e
          case right 'if. cat << if
  )
where
  'cat = for 's ← receive (many symbol). send s
```

12.8 IBIO `wc`

```
'wc = for 'ls ← receive (many line). unit !(foldl.! accum init ls) (where
  'init ∝ ⟨
    'lines ∝ 0;
    'words ∝ 0;
    'runes ∝ 0;
  ⟩;
  'accum 'as ('l :: vector.t rune.t) = ⟨
        !'lines ∝ as #lines + 1;
        !'words ∝ as #words + (l =~ split m/\s+/) #length;
        !'runes ∝ as #runes + l #length;
  ⟩;
);
```