Global Script Program Calculus

Jonathan Cast

April 30, 2013

Contents

Chapter 1

Object Expressions

1.1 Variables

The judgment form for variables is

$$\Gamma \vdash v :: \overline{\tau_i :: \kappa_i}; \overline{a_j :: \tau_j}; \tau \tag{1.1}$$

($i \in [0, n)$, $j \in [n, n + m)$).

- Γ is the type environment,
- v is the variable term in question,
- $\overline{\tau_i :: \kappa_i}$ is a sequence of *default type arguments* and their kinds,
- $\overline{a_j :: \tau_j}$ is a sequence of *default arguments* and their types, and
- τ is the (monomorphic) type of the variable term.

A *variable term* is a simple variable followed by a sequence of *explicit type arguments* of the form @(**type** τ) followed by a sequence of *explicit arguments* of the form @e.

1.1.1 Variables Without implicit Declarations

$$\frac{\Gamma \vdash \overline{\tau_i :: \kappa_i}}{\Gamma, x :: \forall\, \overline{'\alpha_i :: \kappa_i}.\tau \vdash x :: \overline{\tau_i :: \kappa_i}; ; \tau'}$$

- $i \in [0, n)$;
- Γ does not contain an **implicit** declaration for x;
- τ' means $[\overline{\alpha_i \to \tau_i}]\tau$, the result of substituting the type arguments for the $\forall$-bound variables in τ.

1.1.2 Variables With implicit Declarations

$$\frac{\Gamma \vdash \overline{\tau_i :: \kappa_i} \qquad \Gamma \vdash \overline{e_j :: \tau'_j}}{\Gamma, x :: \forall\, \overline{'\alpha_i :: \kappa_i}.\overline{\tau_j} \to \tau, \textbf{implicit}\ x\ \overline{e_j} \vdash x :: \overline{\tau_i :: \kappa_i}; \overline{e_j :: \tau'_j}; \tau'}$$

- $i \in [0, n)$, $j \in [n, n + m)$;
- τ'_j means $[\overline{\alpha_i \to \tau_i}]\tau_j$, the result of substituting the type arguments for the $\forall$-bound variables in τ_j;
- τ' means $[\overline{\alpha_i \to \tau_i}]\tau$, the result of substituting the type arguments for the $\forall$-bound variables in τ.

1.1.3 Explicit Type Applications

$$\frac{\Gamma \vdash x :: \tau_0 :: \kappa_0, \overline{\tau_i :: \kappa_i}; \overline{e_j :: \tau_j}; \tau}{\Gamma \vdash x\ @(\textbf{type}\ \tau_0) :: \overline{\tau_i :: \kappa_i}; \overline{e_j :: \tau_j}; \tau}$$

$(i \in [1, n)$, $j \in [n, n + m))$.

1.1.4 Explicit Applications

$$\frac{\Gamma \vdash x :: \overline{\tau_i :: \kappa_i}; e_n :: \tau_n, \overline{e_j :: \tau_j}; \tau \qquad \Gamma \vdash e :: \tau_n}{\Gamma \vdash x\ @e :: \overline{\tau_i :: \kappa_i}; \overline{e_j :: \tau_j}; \tau}$$

$(i \in [1, n)$, $j \in [n + 1, n + m))$.

1.1.5 Variables Used as Expressions

$$\frac{\Gamma \vdash x :: \overline{\tau_i :: \kappa_i}; \overline{e_j :: \tau_j}; \tau}{\Gamma \vdash x :: \tau}$$

$(i \in [1, n)$, $j \in [n, n + m))$.

1.1.6 TODOs

- Really shouldn't require explicit arguments to be well-typed unless they end up being used.

1.2 Functions

1.2.1 Function Literals

Lambda terms have the rule:[1]

[1] See Chapter 2, Patterns

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

The pattern p must be a lax or strict (not monoidal) pattern; see Section 2.2, Classes of Patterns. BUG: λ takes *multiple* patterns (0 or more). End BUG.

1.2.2 Applications

Applications have the rule:

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

1.3 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**.

1.3.1 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:

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

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

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 2).

Chapter 2

Patterns

The judgment form for patterns is

$$\Gamma \vdash \overline{p_i :: \tau_i} \rhd \Gamma' \tag{2.1}$$

2.1 Concatenating Sequences of Patterns

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

$(i \in [0, n),\ j \in [n, n + m))$. 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 4, Modules. End TODO.

2.2 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.

2.3 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., $('+)$

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

Wildcard Patterns

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

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.

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

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.

Chapter 3

Types

Chapter 4

Modules

Chapter 5

Overloading

Chapter 6

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 8).[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 $module\ 'm.(\langle \text{export list} \rangle) \propto e$).

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

- 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.

The inference rule from programs is

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

Chapter 7

Documents

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

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

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 (**where** $\overline{g_i}$) is equivalent to the expression **for rec** ($\overline{g_i}$) e, as might be expected. (Except that more generators are legal in a **for** than in a **where**.)

Chapter 8

Standard Library

Chapter 9

Examples

9.1 `gstype` Hello, World

qq{`Hello, world!\n`}

9.2 `gsdraw` Hello, World

text str{Hello, world!}

9.3 Dance Hello, World

text str{Hello, world!}

9.4 IBIO `echo`

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

9.5 IBIO `cat`

(**for** (*'as* ← *env.args.get*) **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;
)
```