Global Script String Code Specification

Jonathan Cast
<jcast@globalscript.org>

July 8, 2014

Contents

Chapter 1

Organization of a Program

Global Script programs are organized into a *prefix*, contained in some number of *prefix files*, and a *document*, contained in a single *document file*. Essentially, prefix files contain library code, whereas document files contain code specific to a single program. At the source level, each prefix file consists of a single *prefix generator*; a document file consists of an expression — the program's entry point or '`main`' — and an optional **where** clause. For various reasons, string code syntax flattens both prefix and document files out into a sequence of *items*.

Chapter 2

Layout of a String Code File

Every string code file must begin with the syntax, followed by blank line:

```
#calculus: gsdl.string-code 0.6
```

Older versions are also supported: versions 0.4 and 0.5. This document specifieds version 0.6, but it has notes on the changes from earlier versions.

Next after this is the declaration

```
.document
```

or

```
.prefix
```

declaring whether this particular file is a prefix or document file.

2.1 Sections

String code files are divided into *sections*: the data section, begun by the line

```
.data
```

containing declarations of the file's data items; the code section, begun by the line

```
.code
```

containing auxiliary *code items* which are part of the definition of the data items; the type section, begun by the line

```
.type
```

containing declarations of the file's type items; and the coercion section, begun by the line

```
.coercion
```

containing declarations of the file's coercion items.

Sections should normally be included in the above order; it is illegal to include multiple sections with the same name.

2.2 Lexical Syntax

- Each line consists of a sequence of fields separated by whitespace.
- Lines may optionally be terminated by the field value `--`, which begins a comment.
- String literals and rune literals are un-quoted and may contain the standard C escapes as well as `\s`, which represents an ASCII space. [1] [2]
- If a line begins with whitespace, field 0 of that line is the empty string; any run of linear whitespace is equivalent to a single space and whitespace at the end of the line is ignored, so empty fields are not possible except for field 0.
- Lines consisting exclusively of whitespace and/or comments are ignored.
- In general, field 0 of any line will be referred to as the *label*; field 1 will be variously referred to as the *directive* or the *op*. Labels of width 0 will be referred to as *missing*; labels may be required or forbidden, depending on the op. Any non-blank non-comment line must contain a field 1; additional fields may be required or permitted depending on the directive or op; they will be referred to as the *arguments*.

[1] `.ags-only:` String literals are only supported for simplifying hand-written string code, for bootstrapping.

[2] **Future direction:** We will add a syntax for writing arbitrary UTF coding units using ASCII characters.

Chapter 3

Data Items

Any data item has the form

```
label directive args
```

`label` is the name of the object variable being defined; `directive` declares the form of the declaration. The syntax of `args` will depend on `directive`.

The first data item in a document's data section is the *entry point*; the value of this data item is the value of the document. The label is optional on this data item, and only this data item.

3.1 .closure

```
label .closure <code label> <type label>?
```

This is the general form for top-level data declarations, used for functions, block statements, and thunks.

`<code label>` is the label of an `.expr` or `.impprog` item in the code section, containing the variable `label`'s definition.

3.2 .record

```
label .record (field value)*
```

This defines a simple object-member-only structure literal; the value of each field is given by the object variable following the field name.

3.3 .constr

```
label .constr type constr (field value)*
```

or

```
label .constr type constr arg
```

This defines a literal of a sum type; `type` is the exact sum type the constructor should include into, and `constr` is the constructor name to use.

Summands in Core or String Code sum types can have one of two forms:

- A regular boxed type, of any form.

  In this case, the second form will be used for `.constr`, with a simple variable for the tagged value.

- An un-boxed product.

  In this case, the first form will be used for `.constr`. The field name / value pairs are the same data used for a record literal, which will then be taken to be the tagged value.

3.4 `.rune`

```
label .rune r
```

This bind `label` to a simple (un-boxed, un-lifted, un-cast) rune literal. This will have type "$definedprim\ rune.prim\ rune$.

3.5 `.string`

[1]

```
label .string string
```

This binds `label` to a *lifted, boxed* string literal of type *list.t rune.t*. The byte-compiler will expand this definition to a chain of : constructors terminated by a *nil* constructor, with literal un-boxed runes for the elements.

3.6 `.list`

[2]

```
label .list type elem*
```

This binds `label` to a *lifted, boxed, cast* list literal of type *list.t type*. The byte-compiler will expand this definition to a chain of : constructors terminated by a *nil* constructor.

Alternative form:[3]

[1] **`.ags-only:`** String literals are only supported in hand-written .ags files; the string compiler never generates them.

[2] **`.ags-only:`** List literals are only supported in hand-written .ags files; the string compiler never generates them.

[3] **Unimplemented:** This isn't supported yet, and may never be, as it hasn't proven necessary yet.

```
label .list type elem* | tail
```

This binds `label` to a 'dotted list', where *tail* is used in place of the final *nil* constructor.

3.7 .regex

[4]

```
label .regex re arg*
```

This binds `label` to a *lifted, boxed, cast* regex literal of type *regex.t rune.t*. The byte-compiler will expand this definition to a tree of constructor literals; the character § can be used to insert another data item, taken from the `arg` list, into the generated tree.

3.8 .undefined

```
label .undefined type
```

This binds `label` to the least element of the lifted, boxed type `type`. This is used to translate undefined variables at the core or source level.

3.9 .cast

```
label .undefined x co
```

This binds `label` to the same value as `x` (a data item), but with a new type obtained by coercing it through `co` (a coercion item). This is useful for casting data items created by `.constr` or `.record` generators into abstract types, although it will be used more generally.

[4] **.ags-only:** Regex literals are only supported in hand-written .ags files; the string compiler never generates them.

Chapter 4

Code Items

Code items are substantially more complicated than data items.

4.1 `.expr`

```
label .expr
ops
```

This defines a code item which represents a discrete expression, or a lambda term whose body is an expression.

The contents of a `.expr` code item will be:

- A list of the items (global variables) this expression depends on, comprised of:
  - A sequence of global type variable declarations (section 5.3);
  - A sequence of sub-code declarations (section 4.6);
  - A sequence of global coercion variable declarations (section 4.7); and
  - A sequence of global object variable declarations (section 4.8).
- A list of the (non-global) free variables of this expression, comprised of:
  - A sequence of free type variable declarations (section 4.9);
  - A sequence of type lets using the free type variables decared above (section 5.7);
  - A sequence of free object variable declarations (section 4.10), using the type variables declared above;
- A sequence, intermixed in the appropriate order, of:
  - Type argument declarations (section 4.11);

  - Type lets using the type arguments declared above (and other type variables in scope at this point) (section 5.7); and
  - Object argument declarations, using the type variables declared above (section 4.13).

- A list of local declarations within this expression, intermixed in the appropriate order,[1] consisting of
  - Thunk allocations (section 4.16); and
  - Value allocations (section 4.17).
- A list of evaluation contexts, in the appropriate order (section 4.20). And, finally,
- A terminal operator (section 4.21).

4.2 `.forcecont`

```
label .forcecont
```

This defines a code item which represents an evaluation context of the form **for** $\lfloor x_0 \rfloor \propto \bullet.\ e1$. This represents a continuation for a lifted expression, which receives the unlifted value as an argument.

The contents of a `.forcecont` code item will be:

- A list of the items (global variables) this evaluation context depends on, comprised of:
  - A sequence of global type variable declarations (section 5.3);
  - A sequence of sub-code declarations (section 4.6);
  - A sequence of global coercion variable declarations (section 4.7); and
  - A sequence of global object variable declarations (section 4.8).
- A list of the (non-global) free variables of this evaluation context, comprised of:
  - A sequence of free type variable declarations (section 4.9);
  - A sequence of type lets using the free type variables decared above (section 5.7);
  - A sequence of free object variable declarations (section 4.10), using the type variables declared above;
- A further sequence of type lets (section 5.7).

[1] **Future direction:** We plan to support recursion at this point, at which point the only restriction will be that forward references will have to be to variables with type signatures.

- A single continuation argument (section 4.14).
- A list of local declarations within this evaluation context, intermixed in the appropriate order,[2] consisting of
  - Thunk allocations (section 4.16); and
  - Value allocations (section 4.17).
- A list of evaluation contexts, in the appropriate order (section 4.20). And, finally,
- A terminal operator (section 4.21).

4.3 `.strictcont`

`label .strictcont`

This defines a code item which represents an evaluation context of the form **for** $!x = \bullet.\ e1$. This represents a continuation for a lifted expression, which receives the unlifted value as an argument.

The contents of a `.strictcont` code item will be:

- A list of the items (global variables) this evaluation context depends on, comprised of:
  - A sequence of global type variable declarations (section 5.3);
  - A sequence of sub-code declarations (section 4.6);
  - A sequence of global coercion variable declarations (section 4.7); and
  - A sequence of global object variable declarations (section 4.8).
- A list of the (non-global) free variables of this evaluation context, comprised of:
  - A sequence of free type variable declarations (section 4.9);
  - A sequence of type lets using the free type variables decared above (section 5.7);[3]
  - A sequence of free object variable declarations (section 4.10), using the type variables declared above;
- A further sequence of type lets (section 5.7).
- A single continuation argument (section 4.14).

[2] **Future direction:** We plan to support recursion, at which point the only restriction will be that forward references will have to be to variables with type signatures.

[3] **Unimplemented:** This hasn't been implemented yet (`.strictcont`s are little used); yell if you need it.

- A list of local declarations within this evaluation context, intermixed in the appropriate order,[4] consisting of
  - Thunk allocations (section 4.16);[5] and
  - Value allocations (section 4.17).
- A list of evaluation contexts, in the appropriate order (section 4.20). And, finally,
- A terminal operator (section 4.21).

4.4 `.ubcasecont`

`label .ubcasecont`

The elimination context for an un-boxed sum type has the form

analyze • . **case** c_0 x_0. e_0 ... **case** c_{n-1} x_{n-1}. e_{n-1}

. GSDL Global Script uses a *vectored return* [1] to implement this construct, where the evaluation context is stored on the stack as a vector of `.ubcasecont`s corresponding to the individual cases, together with a single common vector of free variable bindings. This allows an un-boxed constructor (or primitive) to pop the appropriate case off the stack and branch to it directly. A `.ubcasecont` code item thus corresponds to a single **case** in the above construct.

The contents of a `.ubcasecont` code item will be:

- A list of the items (global variables) this evaluation context depends on, comprised of:
  - A sequence of global type variable declarations (section 5.3);
  - A sequence of sub-code declarations (section 4.6);
  - A sequence of global coercion variable declarations (section 4.7); and
  - A sequence of global object variable declarations (section 4.8).
- A list of the (non-global) free variables of this evaluation context, comprised of:
  - A sequence of free type variable declarations (section 4.9);
  - A sequence of type lets using the free type variables decared above (section 5.7);[6]
  - A sequence of free object variable declarations (section 4.10), using the type variables declared above;

[4] **Future direction:** We plan to support recursion, at which point the only restriction will be that forward references will have to be to variables with type signatures.

[5] **Unimplemented:** This hasn't been implemented yet, either.

[6] **Unimplemented:** This hasn't been implemented yet; yell if you need it.

- A further sequence of type lets (section 5.7).
- A continuation argument, declared either as
  - A single continuation argument (section 4.14), or
  - A list of continuation argument fields (section 4.15).
- A list of local declarations within this evaluation context, intermixed in the appropriate order,[7] consisting of
  - Thunk allocations (section 4.16);[8] and
  - Value allocations (section 4.17).
- A list of evaluation contexts, in the appropriate order (section 4.20). And, finally,
- A terminal operator (section 4.21).

4.5 `.impprog`

`label .impprog primset primtype`

This defines a closure of the form **for** $@m\ \overline{g_{ii}}.\ e$.[9] These are special values, since they need to be executed on an API thread rather than being part of regular evaluation.

Closures with `.impprog` bodies are considered 'boxed', which makes them un-lifted, unless you specifically lift them using a `.lift` evaluation context or give them arguments.[10]

The contents of an `.impprog` code item will be:

- A list of the items (global variables) this evaluation context depends on, comprised of:
  - A sequence of global type variable declarations (section 5.3);
  - A sequence of sub-code declarations (section 4.6);
  - A sequence of global coercion variable declarations (section 4.7);[11] and
  - A sequence of global object variable declarations (section 4.8).[12]

[7] **Future direction:** We plan to support recursion, at which point the only restriction will be that forward references will have to be to variables with type signatures.

[8] **Unimplemented:** This hasn't been implemented yet, either.

[9] **Future direction:** Or **for rec** $@m\ \overline{g_{ii}}.\ e$, once we add support for recursion.

[10] **New in version 0.6**; previously, these were considered lifted if and only if the `.body` expression was lifted, and they were required to be lifted if the expression on the right-hand side of the first `.bind` expression was lifted.

[11] **Unimplemented:** This isn't implemented yet, since no implemented generator or body form would support it yet

[12] **Unimplemented:** This isn't implemented yet, since no implemented generator or body form would support it yet

- A list of the (non-global) free variables of this evaluation context, comprised of:
  - A sequence of free type variable declarations (section 4.9);
  - A sequence of type lets using the free type variables decared above (section 5.7);[13]
  - A sequence of free object variable declarations (section 4.10), using the type variables declared above;
- A further sequence of type lets (section 5.7).[14]
- A sequence, intermixed in the appropriate order,[15] of:
  - Type argument declarations (section 4.11);
  - Type lets using the type arguments declared above (and other type variables in scope at this point) (section 5.7); and
  - Object argument declarations, using the type variables declared above (section 4.13).
- A list of local declarations within this evaluation context, intermixed in the appropriate order,[16] consisting of
  - Thunk allocations (section 4.16); and
  - Bind generators (section 4.18).

  And, finally,
- A body expression (section 4.19).

4.6 Sub-Code Items

These declare dependencies on code items which sub-expressions or evaluation contexts within this expression were compiled to.

4.6.1 `.subcode`

```
c .subcode
```

This declares that the code item `c` is a sub-expression or evaluation context of the current code item.

[13] **Unimplemented:** This hasn't been implemented yet; yell if you need it.
[14] **Unimplemented:** This hasn't been implemented yet, either.
[15] **Unimplemented:** The arbitrary inter-mixing isn't supported yet; currently, the sequence must be type arguments, then type lets, then value arguments.
[16] **Future direction:** We plan to support recursion. Obviously, we will still require that bind generators be placed in the order they will be executed in, but at that point allocations can be listed in any order.

4.7 Global Coercion Variables

These are type variables with definitions that are global in scope, as opposed to 'free' coercion variables,[17] which are lambda-bound at a higher level or defined in an intermediate enclosing scope.

4.7.1 `.cogvar`

```
co .cogvar
```

This declares a dependency on the global coercion variable `co`.

4.8 Global Data Variables

These are value variables (generally, object variables) with definitions that are global in scope, as opposed to 'free' value variables, which are lambda-bound at a higher level or defined in an intermediate enclosing scope.

4.8.1 `.gvar`

```
x .gvar
```

This declares a dependency on the global variable `x`.

4.8.2 `.rune`

```
r .rune ch
```

This binds `r` to the rune literal `ch`. Technically, this is a generator, but is kept with the global variables so we have a good storage location for `ch`.

4.8.3 `.natural`

```
x .natural n
```

This binds `x` to the decimal natural number literal `n`. `n` should be small enough to be stored un-boxed, which makes this syntax somewhat machine-dependent. This is technically a generator, but is kept with the global variables so we have a good storage location for `n`.

[17] **Unimplemented:** we don't support coercion abstractions or definitions, yet.

4.9 Free Type Variables

These are type variables that are lambda-bound in an enclosing expression. These should only be lambda-bound variables, and not let-defined variables, because string code is typed bottom-up, which means each code item is typed (in terms of its free type variables) first, then that type is used to type-check the enclosing context. This means that free type variables will be treated as lambda-bound regardless of how they are actually defined.

4.9.1 `.tyfv`

```
t .tyfv ki
```

This declares `t` to be a free type variable of kind `ki` (see chapter 8).

4.10 Free Variables

These are free object or value variables that are bound in an intermediate scope between this expression or evaluation context and the global level of the program.

4.10.1 `.fv`

```
x .fv tyf tyx0 ... tyxn-1
```

This declares `x` to be a free variable of type `tyf` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$.

4.10.2 `.efv`

```
x .efv tyf tyx0 ... tyxn-1
```

This is the same as `.fv`, but warns that `x`, which is of un-lifted type, may nevertheless be bound to a (fulfilled) promise. As such, when this expression is entered, `x` should first have all indirections removed before the value is used.

This is required for any (unlifted) free variable which is bind-bound in the enclosing scope, as well in some cases involving mutual recursion between lifted and unlifted variables.

4.11 Type Arguments

These are abstractions over type variables.

4.11.1 `.tyarg`

```
t .tyarg ki
```

This declares `t` to be a type variable with kind `ki` (see chapter 8).

4.12 Type Continuation Arguments

4.12.1 .exkarg

```
t .exkarg ki
```

This declares that the argument to the continuation has an existential quantifier, with quantified type variable `t` of kind `ki` (see chapter 8).

4.13 Value Arguments

These are abstractions over object or value variables.

4.13.1 .larg

```
x .larg tyf tyx_0 ... tyx_n-1
```

This does two jobs: it declares `x` to be an argument of type `tyf` tyx_0 ... tyx_{n-1}, and it lifts the resulting lambda term.

4.13.2 .arg

```
x .arg tyf tyx_0 ... tyx_n-1
```

This only declares `x` to be an argument of type `tyf` tyx_0 ... tyx_{n-1}; the resulting expression is un-lifted.

4.14 (Boxed) Continuation Arguments

This declares the (single) value argument to a continuation for a boxed type.

4.14.1 .karg

```
x .karg tyf tyx0 ... tyxn-1
```

This declares `x` to be the continuation's argument, of type `tyf` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$.

4.15 (Un-boxed) Continuation Arguments

This declares the (single) value argument to a continuation for a boxed type.

4.15.1 .karg

```
x .fkarg f tyf tyx0 ... tyxn-1
```

This declares `x` to be the field `f` of the continuation's (un-boxed product) argument, of type `tyf` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$.

4.16 Thunk (Closure) Allocations

These are actually just the allocations that are supported in both expressions and in imperative block statements.

4.16.1 `.closure`

```
label .closure <code label>
```

`<code label>` should be either an `.expr` or a `.impprog` code item; this construct binds `label` to the expression defined by `<code label>`.

Legally, the label can be omitted with a warning (since allocation operations are side-effect-free).

4.16.2 `.impprim`

[18] [19]

```
x .impprim primset primtype prim ty tyx0 ... tyxn-1 | x0 ... xn-1
```

This defines `x` to be the image of `prim`, from `primset`, at type arguments $\texttt{tyx}_0$, ..., $\texttt{tyx}_{n-1}$ and value arguments $\texttt{x}_0$, ..., $\texttt{x}_{n-1}$. `ty` should be a type synonym that expands to the actual, polymorphic, type of `prim`; for primsets known to the interpreter, it will be checked against the actual type, and for primsets unknown to the interpreter, it will be blindly believed and used to type-check the arguments and assign a type to `x`. The result type must be an appliction of `primtype` from the same primitive set, which must be an imperative primitive type.

4.16.3 `.lfield`

[20] [21]

```
x .lfield f r
```

This binds `x` to the field `f` of `r`, which must be a lifted record with a field `f`, which must in turn have lifted type.

[18] **New in version 0.5**; previously, in imperative block statements, this had to be put out in a `.closure` and the result returned.

[19] **Unimplemented:** This is un-supported in imperative block statements as yet.

[20] **New in version 0.5**; previously, in imperative block statements, this had to be put out in a `.closure` and the result returned.

[21] **Unimplemented:** This is un-supported in imperative block statements as yet.

4.16.4 `.undefined`

22 23

```
x .undefined tyf tyx0 ... tyxn-1
```

This binds `x` to the 'undefined' value of type `tyf` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$, which must be a lifted type.

4.16.5 `.lifted`

24 25

```
y .lifted x
```

This binds `y` to the image of `x` under the unit of the lifting monad.

4.16.6 `.cast`

26 27

```
y .cast x cof tyx0 ... tyxn-1
```

This binds `x` to the image of `x` under the application of the coercion `cof` to the type variables $\texttt{tyx}_0$, ..., $\texttt{tyx}_{n-1}$.

4.16.7 `.apply`

28 29

```
y .cast f tyx0 ... tyxn-1 | x0 ... xm-1
```

This binds `y` to the image of `f` a type arguments $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$ and value arguments $\texttt{x}_0$ ... $\texttt{x}_{m-1}$.

4.17 Value Allocations

These are the allocations that are not legal in imperative block statements.

[22] **New in version 0.5**; previously, in imperative block statements, a `.undef` had to be put out in a `.closure`.

[23] **Unimplemented:** This is un-supported in imperative block statements as yet.

[24] **New in version 0.5**; previously, in imperative block statements, a `.lift` had to be put out in a `.closure`.

[25] **Unimplemented:** This is un-supported in imperative block statements as yet.

[26] **New in version 0.5**; previously, in imperative block statements, a `.coerce` had to be put out in a `.closure`.

[27] **Unimplemented:** This is un-supported in imperative block statements as yet.

[28] **New in version 0.5**; previously, in imperative block statements, a `.app` had to be put out in a `.closure`.

[29] **Unimplemented:** This is un-supported in imperative block statements as yet.

4.17.1 .prim

```
x .prim primset prim type tyx0 ... tyxn-1 | x0 ... xn-1
```

This defines `x` to be the image of `prim`, from `primset`, at type arguments $\texttt{tyx}_0$, ..., $\texttt{tyx}_{n-1}$ and value arguments $\texttt{x}_0$, ..., $\texttt{x}_{n-1}$. `type` should be a type synonym that expands to the actual, polymorphic, type of `prim`; for primsets known to the interpreter, it will be checked against the actual type, and for primsets unknown to the interpreter, it will be blindly believed and used to type-check the arguments and assign a type to `x`.

4.17.2 .constr

```
x .constr t c <args>
```

`<args>` should either be a single variable,[30] or a list of field name / variable pairs, which will be treated as an un-boxed record literal. `t` should be a sum type with a constructor `c`, whose argument type is the type of `<args>`, however interpreted. The construct as a whole binds `x` to the image of `<args>` under the injection defined by `c` into `t`.

4.17.3 .exconstr

```
x .exconstr t c <type args> | <args>
```

`<args>` should either be a single variable,[31] or a list of field name / variable pairs, which will be treated as an un-boxed record literal. `t` should be a sum type with a constructor `c`, whose argument type has as many existential quantifiers as there are type arguments, and whose body, after substituting the values of the type arguments for those quantified variables, is the type of `<args>`, however interpreted. The construct as a whole binds `x` to the image of `<args>` under the injection defined by `c` into `t`.

4.17.4 .record, .lrecord

```
x .record <field args>
x .record <field args> | tyf tyx0 ... tyxn-1
x .lrecord <field args>
x .lrecord <field args> | tyf tyx0 ... tyxn-1
```

`<field args>` is a sequence of field name, variable pairs, giving the values of the fields of the record. `.record` binds `x` to the so-defined record; `.lrecord` binds `x` to its image under the unit of the lifting monad. The clause `| tyf` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$, if present, is a type signature; it gives the type of `x`, so in the `.lrecord` case it must itself be lifted. Every field in the record must be given a value explicitly; there is no support for undefined fields.

[30] **Unimplemented:** only the record form is supported currently
[31] **Unimplemented:** only the record form is supported currently

4.17.5 `.field`

```
x .field f r
```

This binds `x` to the field `f` of `r`, which must be an unlifted record with a field `f`.

4.18 Bind Generators

These are used in imperative block statements to execute sub-programs and name their results.

4.18.1 `.bind`

[32]

```
x .bind <alloc op>
```

This declares `x` to be the result of executing the subprogram denoted by the allocation operator. The allocation operator can be any specified in section 4.16, although since it will need to be of imperative type, not all allocations will be useful.[33]

4.19 Block Statement Bodies

These are similar to bind generators (section 4.18), but they define the last sub-program in a block statement. The result of these sub-programs is returned directly from the block statement, rather than being captured and used inside it.

4.19.1 `.body`

[34]

```
.body <alloc op>
```

This declares the result of the block statement to be the result of executing the subprogram denoted by the allocation operator. As with bind generators, the allocation operator can be any specified in section 4.16, although since it will need to be of imperative type, not all allocations will be useful.[35]

[32] **New in version 0.5**; previously, these were combined into one token; e.g., what is now written `.bind .closure` was `.bind.closure`.

[33] **Unimplemented:** Only `.closure` is currently implemented.

[34] **New in version 0.5**; previously, these were combined into one token; e.g., what is now written `.body .closure` was `.body.closure`.

[35] **Unimplemented:** Only `.closure` is currently implemented.

4.20 Evaluation Contexts

4.20.1 `.lift`

`.lift`

Corresponds to the evaluation context $\lfloor \bullet \rfloor$, which returns the image of the following expression under the unit of the lifting monad.

4.20.2 `.coerce`

`.coerce cof tyx0 ... tyxn-1`

Corresponds to the evaluation context $\bullet$ " $\triangleright$ *cof* (**type** tyx_0) ... (**type** tyx_{n-1}), which returns the image of the following expression under the given coercion.

4.20.3 `.app`

`.app x0 ... xn-1`

Corresponds to the evalution context $\bullet$ x_0 ... $x_n - 1$, which returns the image of the given arguments under the following expression.

4.20.4 `.force`

`.force k`

Corresponds to the evaluation context denoted by the code item labeled by `k`, which must be a `.forcecont`.

4.20.5 `.strict`

`.strict k`

Corresponds to the evaluation context denoted by the code item labeled by `k`, which must be a `.strictcont`.

4.20.6 `.ubanalyze`

`.ubanalyze c0 k0 ... cn-1 kn-1`

Corresponds to an evaluation context of the form

analyze $\bullet$. **case** c_0 x_0. e_0 ... **case** c_{n-1} x_{n-1}. e_{n-1}

, where the scrutinee has un-boxed sum type or lifted un-boxed sum type, the constructors are the arguments $\texttt{c}_0$, ..., $\texttt{c}_{n-1}$, and the corresponding cases are given by the code items labeled by $\texttt{k}_0$, ..., $\texttt{k}_{n-1}$.

4.21 Terminal Operators

4.21.1 .undef

```
.undef tyf tyx0 ... tyxn-1
```

This allocates and returns the least element of type `tyf` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$, which must be a lifted type.

4.21.2 .yield

```
.yield x tyx0 ... tyxn-1
```

This returns the value of `x` applied to the type arguments $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$. `x` must be of unlifted type, and hence bound to an actual value.

4.21.3 .enter

```
.enter x tyx0 ... tyxn-1
```

This expression has the value of `x` applied to the type arguments $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$. `x` must be of lifted type, and so may be bound to a value or to a thunk. If bound to a value, the `.enter` instruction will return that value; if bound to a thunk, the `.enter` instruction will enter that thunk.

4.21.4 .ubprim

```
.ubprim primset prim ty tyx0 ... tyxn-1 | x0 ... xn-1
```

This returns the image of the primitive `prim` from primset `primset` at type arguments $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$ and value arguments $\texttt{x}_0$ ... $\texttt{x}_{n-1}$, which should be a saturated application. `ty` should be the actual (possibly polymorphic) type of the primitive, and the type of the whole application should be a (possibly lifted) un-boxed sum or un-boxed product.

XXX Do lifted un-boxed primitives use this or `.lprim`?

4.21.5 .lprim

```
.lprim primset prim ty tyx0 ... tyxn-1 | x0 ... xn-1
```

This returns the image of the primitive `prim` from primset `primset` at type arguments $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$ and value arguments $\texttt{x}_0$ ... $\texttt{x}_{n-1}$, which should be a saturated application. `ty` should be the actual (possibly polymorphic) type of the primitive, and the type of the whole application should be lifted.

(Non-imperative) primitives are actually divided at the implementation level into three classes:

- Simple primitives, which run in a bounded amount of time (and are uninterruptible) and return a boxed un-lifted value;

- Un-boxed primitives, which return an un-boxed value and so need a vector of continuations to return to; and
- Lifted primitives, which may block (and so need to be interrupted), may never return, and return a lifted value.

The different primitive instructions correspond to this breakdown.

4.21.6 `.analyze`

This isn't really a 'terminal' operator, but it's legal only where we allow terminal operators.

```
.analyze x c0 ... cn-1
```

This branches on `x`, which must be of (un-lifted) sum type with constructors $\texttt{c}_0$, ..., $\texttt{c}_{n-1}$. The constructors must be given in unibetical order by printed name, as used in the source code. `x` may not have empty sum type.

The `.analyze` statement must be followed by a sequence of `case` ops (see section 4.22), in order, one per constructor, defining the behavior for the various branches.

4.21.7 `.danalyze`

This isn't really a 'terminal' operator, but it's legal only where we allow terminal operators.

```
.danalyze x c0 ... cn-1
```

This branches on `x`, which must be of (un-lifted) sum type which has constructors $\texttt{c}_0$, ..., $\texttt{c}_{n-1}$, among others. The constructors must be given in unibetical order by printed name, as used in the source code. There must be at least one constructor listed, but the type of `x` may have other constructors not listed. In that case, if `x` is the image of one of those constructors, evaluation will proceed with the default case.

The `.analyze` statement must be followed by a `default` op (see section 4.22), then a sequence of `case` ops (see section 4.22), in order, one per constructor, defining the behavior for the various branches.

4.22 `.case` and `.default`

```
.case c
ops
```

or

```
.default
ops
```

This defines a specific case, or the default case of a `.danalyze`.
The contents of a `.case` will be:

- A sequence of:
  - Type argument declarations (section 4.12).
- A sequence of:
  - Type lets using the type arguments declared above (and other type variables in scope at this point) (section 5.7).
- A continuation argument, declared either as
  - A single continuation argument (section 4.14), or
  - A list of continuation argument fields (section 4.15).
- A list of local declarations within this expression, intermixed in the appropriate order,[36] consisting of
  - Thunk allocations (section 4.16); and
  - Value allocations (section 4.17).
- A list of evaluation contexts, in the appropriate order (section 4.20). And, finally,
- A terminal operator (section 4.21).

A `.default` will have the same contents, but with the arguments and type lets omitted.

[36] **Future direction:** We plan to support recursion at this point, at which point the only restriction will be that forward references will have to be to variables with type signatures.

Chapter 5

Types

5.1 Type Items

5.1.1 `.tyexpr`

```
ty .tyexpr ki
```

The kind (see chapter 8) is optional. This defines a type synonym `ty`, of kind `ki` if that is present. The arity of the synonym is determined by the number of `.tylambda` declarations following the `.tyexpr`, and the definition is given by the type expression following the `.tyexpr` (see section 5.2).

5.1.2 `.tyabstract`

```
ty .tyabstract ki
```

This defines a type constant `ty` of kind `ki` (see chapter 8), defined by the type expression following the `.tyabstract` (see section 5.2).

5.1.3 `.tydefinedprim`

```
ty .tydefinedprim ps pn ki
```

This declares `ty` as a name for the primtype `pn` from primset `ps`, of kind `ki` (see chapter 8).

A 'defined' primitive type is one that has an explicit denotation as part of the Global Script semantics, and can be implemented in language-independent way on any Turing-complete machine. As such, they are part of every GSPC-based language, which means this declaration is only allowed for primsets that are known to the current implementation. It is illegal to supply an incorrect kind signature as part of such a declaration, or to mention a non-existent primset or primtype in the declaration.

5.1.4 `.tyintrprim`

```
ty .tyintrprim ps pn ki
```

This declares `ty` as a name for the introduction primtype `pn` from primset `ps`, of kind `ki` (see chapter 8).

An 'introduction' primitive type is one that has only primitives for building objects of the type, and none for eliminating them. As such, any implementation to which its primset is unknown can treat it as a singleton type. However, interpreters that do know the primset will reject library definitions with invalid primtype declarations in that primset, or with incorrect kinds on known primtypes.

5.1.5 `.tyelimprim`

```
ty .tyelimprim ps pn ki
```

This declares `ty` as a name for the elimination primtype `pn` from primset `ps`, of kind `ki` (see chapter 8).

An 'elimination' primitive type is one that has only primitives objects of the type as arguments, and none for introducing them. As such, any implementation to which its primset is unknown can treat it as an empty type. However, interpreters that do know the primset will reject library definitions with invalid primtype declarations in that primset, or with incorrect kinds on known primtypes.

5.1.6 `.tyimpprim`

```
ty .tyimpprim ps pn ki
```

This declares `ty` as a name for the imperative primtype `pn` from primset `ps`, of kind `ki` (see chapter 8).

An 'imperative' primitive type is an introduction primtype with the special syntax `.impprog` (see section 4.5) (or **for** $@m\ x \leftarrow e_0.\ e_1$ at the Core level), which allows for the creation and execution of sequential-execution block statements. However, any implementation to which its primset is unknown can still treat it as a singleton type.[1] However, interpreters that do know the primset will reject library definitions with invalid primtype declarations in that primset, or with incorrect kinds on known primtypes.

5.2 Type Expressions

A type expression, the body of a `.tyexpr` or a `.tyabstract`, consists of

- A sequence of type global variable declarations (see section 5.3),

[1] **Future direction:** We plan to add a special syntax for unit, as well, which *can* be branched on; at that point, `m` α will need to be treated as $1 + \alpha$.

- A sequence of type arguments (see section 5.4),
- A sequence of universal quantifiers (see section 5.5),
- A sequence of existential quantifiers (see section 5.6),
- A sequence of type lets (see section 5.7),
- A sequence of type contexts (see section 5.8), and finally
- A terminal type (see section 5.9).

5.3 Global Type Variables

These are type variables with definitions that are global in scope, as opposed to 'free' type variables, which are defined (actually, lambda-bound) in an intermediate enclosing scope.

5.3.1 `.tygvar`

```
t .tygvar
```

This declares (a dependency on) the global type variable `t`, which is any type label defined at the top level, either a type synonym or an abstract type (= type constant).

5.3.2 `.tyextabstype`

```
t .tyextabstype ki
```

This declares `t` to be specifically the type constant `t`, which must have kind `ki` (see chapter 8). This construct allows the string compiler to produce output files that are relatively independent of each other.

5.3.3 `.tyextelimprim`

```
t .tyextelimprim ps pn ki
```

This declares `t` to be the elimination primitive type `pn` from primitive set `ps`, with kind `ki` (see chapter 8). This isn't really an external dependency; it's more a kind of type literal, that simplifies the implementation of the string compiler.

5.4 Type Arguments

5.4.1 `.tylambda`

```
t .tylambda ki
```

This translates to a type parameter `t` of kind `ki` (see chapter 8).

5.5 Universal Quantifiers

5.5.1 `.tyforall`

```
t .tyforall ki
```

This translates to a universal quantifier for the type variable `t` of kind `ki` (see chapter 8).

5.6 Existential Quantifiers

5.6.1 `.tyexists`

```
t .tyexists ki
```

This translates to an existential quantifier for the type variable `t` of kind `ki` (see chapter 8).

5.7 Local Type Definitions

These allow for location type variable definitions, mainly for cases where a complex type expression is need but the syntax requires a simple variable.

5.7.1 `.tylet`

```
t .tylet tyf tyx_0 ... tyx_n
```

This defines `t` to be the result of applying `tyf` to the type variables `tyx_0` ... `tyx_n`.

Technically, this only allows type applications to be named; more complicated type expressions can be dealt with via lambda-lifting.

5.8 Type Environments

5.8.1 `.tylift`

```
.tylift
```

This represents the type environment $\lfloor \bullet \rfloor$, which maps the following type to its image under the lifting monad.

5.8.2 `.tyfun`

```
.tyfun tyf tyx0 ... tyxn-1
```

This represents the type environment $tyf\ tyx_0\ \ldots\ tyx_{n-1} \rightarrow \bullet$, which defines a function type from `tyf` $\texttt{tyx}_0\ \ldots\ \texttt{tyx}_{n-1}$ to the following type.

5.9 Terminal Type Operators

5.9.1 `.tyref`

`.tyref tyf tyx0 ... tyxn-1`

This represents the type `tyf tyx`$_0$ `...` `tyx`$_{n-1}$.

5.9.2 `.tysum`

`.tysum f0 ty0 ... fn-1 tyn-1`

This represents the boxed un-lifted sum "$\sum \left\langle \overline{f_i\ ty_i};_{i=0}^{n-1} \right\rangle$.

5.9.3 `.tyubsum`

`.tyubsum f0 ty0 ... fn-1 tyn-1`

This represents the un-boxed un-lifted sum "$u \sum \left\langle \overline{f_i\ ty_i};_{i=0}^{n-1} \right\rangle$.

5.9.4 `.typroduct`

`.typroduct f0 ty0 ... fn-1 tyn-1`

This represents the boxed un-lifted product "$\prod \left\langle \overline{f_i\ ::\ ty_i};_{i=0}^{n-1} \right\rangle$.

5.9.5 `.tyubproduct`

`.tyubproduct f0 ty0 ... fn-1 tyn-1`

This represents the un-boxed un-lifted product "$u \prod \left\langle \overline{f_i\ ::\ ty_i};_{i=0}^{n-1} \right\rangle$.

Chapter 6

Coercion Items

6.1 `.tycoercion`

```
co .tycoercion
```

This binds `co` to the coercion defined by the following code, which consists of:

- A sequence of type global variable declarations (see section 5.3),
- A sequence of type arguments (see section 5.4),
- A sequence of coercion contexts (see section 6.2), and finally
- A terminal coercion (see section 6.3).

6.2 Coercion Contexts

6.2.1 `.tyinvert`

```
.tyinvert
```

This builds a coercion which is the inverse of the following coercion expression.

6.3 Terminal Coercions

6.3.1 `.tydefinition`

```
.tydefinition tyc tyx0 ... tyxn-1
```

This denotes a coercion which maps the definition of `tyc` $\texttt{tyx}_0$ ... $\texttt{tyx}_{n-1}$ to that type.

Chapter 7

Notes on the Type System (Semi-Obsolete)

Free variables are handled by having type λ nodes, and allowing

```
tv .tylet tf x y z
```

in type items, with arguments implemented by doing β-reduction directly on type trees.

In certain kinds of type declarations, type applications are also permitted:

```
x .fv t alpha beta gamma
x .arg t alpha beta gamma
```

`t` can be an abstract type or primitive type or a type synonym.

In both cases, the argument is declared in the type item using `.tylambda`:[1]

```
t .tylambda *
```

[1] Well, sort of.

Chapter 8

Kinds

Kinds in string code are written in one token, in reverse polish notation. The operators are `u`, an atom which denotes the kind of unlifted types, `*`, an atom which denotes the kind of lifted types, and `^`, a binary operator which denotes the exponential. $\kappa_0\kappa_1\hat{}$ is the kind of type functions from kind κ_1 to kind κ_0. Kinds of functions of multiple arguments can be written very simply: $\kappa_r\kappa_0\hat{}\ldots\kappa_{n-1}\hat{}$.

Chapter 9

API Block Statements

These are similar to expressions; in fact, the data items use `.closure` like expressions do; only the code items are different.

In Core, a block statement looks like

for @⟨type arg⟩ ⟨gens⟩. ⟨body⟩

or

for rec @⟨type arg⟩ ⟨gens⟩. ⟨body⟩

The generators look like

⟨var⟩ = ⟨expr⟩; — Let
⌊⟨var⟩⌋ ∝ ⟨expr⟩; — Force
⟨var⟩ ← ⟨expr⟩; — Bind

; the body is an expression.

String code only permits 'let' generators at lifted types and 'bind' generators in its 'imperative block statement' construct; the others are compiled to `.expr`s.[1] [2]

A block statement gets compiled to an `.impprog` code item.

The syntax is

```
label .impprog primset primtype
```

That is followed by global variables, free variables, and arguments as for an `.expr`.

'Let' generators get compiled to

[1] Not so! Ignores the issue of RRF RHSs!

[2] Unlifted 'let' generators and 'force' generators bind variables of unlifted types, and so cannot participate in recursion, *including forward references to bind generators.* NB: That's not entirely true — variables of unlifted but pointed type, like unlifted function types, *can* be recursively defined — need to think about that more. Update: I've thought about this; it's a *long* story.

`var .closure code-label`

'Bind' generators get compiled to[3]

`var .bind .closure code-label`

The body gets compiled to[4]

`.body .closure code-label`

[3] **New in version 0.5**; previously, this was one token, `.bind.closure`.
[4] **New in version 0.5**; previously, this was one token, `.body.closure`.

Chapter 10

`.regex` Literals

[1]

We have interpolation support in string code regex literals. To use, put § (with no name) in the literaland then follow the literal with the variable to use for each interpolation.

You can interpolate any top-level data item, not just `.regex` literals.

`.regex` won't support any grouping operator, so interpolation will be the only way to override the default precedence order. Interpolations aren't supported inside classes yet but they will be.

[1] **`.ags-only:`** Regex literals are only supported for simplifying hand-written string code, for bootstrapping.

Bibliography

[1] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201, New York, NY, USA, 1989. ACM.