other *Prelude* functions is defined by reference to the Haskell report, which provides a source language definition. We assume that the meaning of basic operations (such as addition on numbers) is obvious. To define this formally would be tedious in the extreme.

The initial environment contains the following functions (*BasVal*)

| | |
|---|---|
| PrimPlusInt : $\backslash a$ -> $\backslash b$ -> $a + b$ | $+$ is fixed-precision integer addition |
| PrimMulInt : $\backslash a$ -> $\backslash b$ -> $a \times b$ | $\times$ is fixed-precision integer multiplication |
| ... | |
| error : $\backslash s$ -> $\perp$ | the error function |

plus the standard constructors (*BasCon*):

0, 1, ..., 0.0, 0.1, ..., *True*, *False*, 'a', ..., (:), *Nil*,
*ReadFile*, *ReadBinFile*, *ReadChan*, *ReadBinChan*,
*WriteFile*, *WriteBinFile*,
*AppendFile*, *AppendBinFile*, *AppendChan*, *AppendBinChan*,
*StatusChan*, *StatusFile*, *DeleteFile*,
*GetEnv*, *SetEnv*, *GetProgName*, *GetArgs*, *Echo*,
*Success*, *Failure*, *Str*, *StrList*, *Bn*,
*ReadError*, *WriteError*, *SearchError*, *FormatError*, *OtherError*

The characters correspond to those defined by the ASCII character set. The mapping from syntactic variables to semantic constructors is the obvious one, that is, $\mathrm{E}_0$ (SetEnv) = SetEnv ....

## References


[HW90] Hudak, P and Wadler, PL (eds), "Report on the Programming Language Haskell, Version 1.0", Universities of Yale and Glasgow, April 1990.

[HJW91] Hudak, P, Peyton Jones SL, and Wadler, PL (eds), "Report on the Programming Language Haskell, Version 1.1", Universities of Yale and Glasgow, August 1991.

[HJW92] Hudak, P, Peyton Jones SL, and Wadler, PL (eds), "Report on the Programming Language Haskell, Version 1.2", *SIGPLAN Notices*, March 1992.

[JW92] Peyton Jones, SL and Wadler, PL, "The Static Semantics of Haskell", *Unpublished Technical Report*, Universitry of Glasgow, 1992.

[HMT91] Harper, Milner and Tofte, *The Definition of Standard ML*, MIT Press, 1991.

*noEcho* sets the flag in the state which causes further *Echo* requests to fail.

```
noEcho c S =
        c = stdin → < e, False > in (Echo of S); S
        where       < e, n > = (Echo of S)
```

# 8 Operating System Functions

In writing this semantics, we have assumed the existence of a number of operating system specific functions. These functions are listed in the following table.

| **Function** | **Result** |
|---|---|
| *OScanCreate* file State bin text | *True* if *file* can be created in *State* as a binary and/or text file (as determined by the *bin*/*text* predicates), otherwise *False*. |
| *OScreateStatus* file State bin text | A list of characters representing the initial status of a newly created file, as used by the *StatusFile* request. |
| *OSstatusFile* file State | A list of characters containing OS-specific status information about a file, as used by the *StatusFile* request. |
| *OSstatusChan* chan State | A list of characters containing OS-specific status information about the channel *chan* (for *stdout*, *stderr* and *stdecho* the initial segment of the string should be the maximum line length and the maximum page length allowed for the channel as two separate integers, separated by a single space). |
| *OSisDirectory* file State | *True* if *file* is a directory in state *State*, *False* otherwise. |
| *OSErr* string value | An OS-specific error message for error *string* with parameter *value*. |

# 9 The Initial Environment

The initial environment comprises definitions for all functions and constructors defined in the module *PreludeCore*. These values must be available in all Haskell programs. The meanings of

```
canWriteTextChan c S s  =  canRWChan c S True False True s
canWriteBinChan c S s   =  canRWChan c S True True False s
canReadTextChan c S s   =  canRWChan c S False False True s
canReadBinChan c S s    =  canRWChan c S False True False s
```

*statusString* constructs a string representing the status of a file from the filesystem information. *OSstatusFile* extends the standard string with operating-system specific information.

```
statusString f S  =
    f ∈ dom (File of S) →
        Str < (:) < fileType,
              (:) < readStatus,
              (:) < writeStatus,
              OSstatusFile f S >>>> ;
        searchError "file does not exist" f
    where
        < v, _ > = (File of S) f
        fileType = text v ∧ ¬bin v → 't'; bin v ∧ ¬text v → 'b';
                   text v ∨ bin v → 'f'; OSisDirectory f S → 'd'; 'u'
        readStatus = read v → 'r'; '-'
        writeStatus = write v → 'w'; '-'
```

*statusChanString* is similar, but returns status information about a channel. *OSStatusChan* is used to return the status information if the channel exists, otherwise *Failure* is returned.

```
statusChanString c S  =
        f ∈ dom (Channel of S) → Str (OSStatusChan c S);
        searchError "channel does not exist" c
```

*canDeleteFile* returns either *Success* if the named file can be deleted or a *Failure*.

```
canDeleteFile f S  =
        f ∈ dom (File of S) →
            delete v → Success; writeError "cannot delete file" f;
        searchError "file does not exist" f
        where < v, s > = (File of S) f
```

*doEcho* returns either *Success* and a new state if echoing is permissible or *Failure* and the existing state.

```
doEcho v S  =
    n → otherError "cannot set echo";
        Success, < v, False > in (Echo of S)
    where    < e, n > = (Echo of S)
```

the current contents of the file, otherwise *s'* replaces the current file contents in the filesystem. *OScanCreate* and *OScreateStatus* are operating-system dependent functions which determine whether the file can be created, and if so its initial status, respectively.

```
canRWFile f S w b t a s' =
    f ∈ dom (File of S) →
        < testReadWrite ∧ testBinText → Success; readWriteErrorMesg,
            < v, ifWriteThenTestAppend >> ;
        < w ∧ ¬a ∧ OScanCreate f S b t → Success; readCreateErrorMesg,
            < OScreateStatus f S b t, s' >>
    where
        < v, s > = (File of S) f
        testReadWrite = w → write v; read v
        testBinText = (b → bin v; True) ∧ (t → text v; True)
        readWriteErrorMesg = w → writeError "can't write file" f;
                                 readError "can't read file" f
        ifWriteThenTestAppend = w → (a → s ++ s'; s'); s
        readCreateErrorMesg = w → writeError "can't create file" f;
                                  readError "file does not exist" f
```

*canWriteTextFile*, *canWriteBinFile* etc. are packaged versions of *canRWFile* for the obvious cases.

```
canWriteTextFile f S a s = canRWFile f S True False True a s
canWriteBinFile f S a s  = canRWFile f S True True False a s
canReadTextFile f S a s  = canRWFile f S False False True a s
canReadBinFile f S a s   = canRWFile f S False True False a s
```

*canRWChan* is analogous to *canRWFile*, but for channels. Values written to channels are always appended to the current contents, and channels can never be created.

```
canRWChan c S w b t s' =
        c ∈ dom (Channel of S) →
                < (w →writec; readc) v ∧ bt v → Success;
                                (w →writeError "can't write channel" c;
                                    readError "can't read channel" c),
                        < v, w → s ++s';s >> ;
        searchError "channel does not exist" c
        where < v, s > = (Channel of S) c
                bt v = (b → binc v; True) ∧ (t → textc v; True)
```

*canWriteTextChan* etc. are packaged versions of *canRWChan* analagous to *canWriteTextFile* etc.

$$\frac{\text{s} \in \text{dom (Environ of S)} \quad \text{s'} = \text{(Environ of S) s}}{\text{S} \vdash \mathit{GetEnv} < \text{s} > \Rightarrow \mathit{Str} < \text{s'} >, \text{S}} \tag{80}$$

$$\frac{\text{s} \notin \text{dom (Environ of S)}}{\text{S} \vdash \mathit{GetEnv} < \text{s} > \Rightarrow \mathit{envError}\ \text{s},\ \text{S}} \tag{81}$$

$$\frac{\text{S'} = \text{(Environ of S)} \vec{\oplus} \{ \text{s} : \text{s'} \}}{\text{S} \vdash \mathit{SetEnv} < \text{s}, \text{s'} > \Rightarrow \mathit{Success}, \text{S} \vec{\oplus} \text{S'}} \tag{82}$$

$$\frac{S' = \mathit{doEcho}\ \text{v S}}{\text{S} \vdash \mathit{Echo} < \text{v} > \Rightarrow \mathit{Success}, \text{S'}} \tag{83}$$

## 7.1 Auxiliary File-System Functions

This section defines the auxiliary functions which are used when defining the semantics of filesystem requests. *read*, *write*, etc. determine whether a file is readable, writable etc. according to the status recorded within the filesystem. *readc*, *writec* perform similar operations for channels.

| | |
|---|---|
| read $< \text{v}, _, _, _, _ > = \text{v}$ | readc $< \text{v}, _, _, _ > = \text{v}$ |
| write $< _, \text{v}, _, _, _ > = \text{v}$ | writec $< _, \text{v}, _, _ > = \text{v}$ |
| delete $< _, _, \text{v}, _, _ > = \text{v}$ | |
| text $< _, _, _, \text{v}, _ > = \text{v}$ | textc $< _, _, \text{v}, _ > = \text{v}$ |
| bin $< _, _, _, _, \text{v} > = \text{v}$ | binc $< _, _, _, \text{v} > = \text{v}$ |

We also define some auxiliary functions which are used when returning error values. *OSErr* constructs an operating-system specific error message from a string and error value.

| | |
|---|---|
| *readError* s v = | *Failure* (*ReadError* (*OSErr* s v)) |
| *writeError* s v = | *Failure* (*WriteError* (*OSErr* s v)) |
| *searchError* s v = | *Failure* (*SearchError* (*OSErr* s v)) |
| *formatError* s v = | *Failure* (*FormatError* (*OSErr* s v)) |
| *otherError* s v = | *Failure* (*OtherError* (*OSErr* s v)) |
| *envError* v = | *searchError* "Environment variable does not exist" v |

*canRWFile* determines whether the file $f$ can be read or written (depending on the value of $w$) as a binary or text file (depending on the settings of $b$ and $t$). If $a$ is *True* then $s$' is appended to

$$\frac{\mathit{canWriteBinFile}\ \mathrm{f\ S\ False\ s}\ =\ <\mathit{Success}, \mathrm{v}> \qquad \mathrm{S'}\ =\ (\mathrm{File\ of\ S})\ \vec{\oplus}\ \{\mathrm{f} : \mathrm{v}\}}{\mathrm{S} \vdash\ \mathit{WriteBinFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Success},\ \mathrm{S}\ \vec{\oplus}\ \mathrm{S'}} \tag{72}$$

$$\frac{\mathit{canWriteBinFile}\ \mathrm{f\ S\ False\ s}\ =\ <\mathit{Failure}\ \mathrm{f}, \mathrm{v}>}{\mathrm{S} \vdash\ \mathit{WriteBinFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Failure}\ \mathrm{f}, \mathrm{S}} \tag{73}$$

Rules (74)–(83) handle miscellaneous requests.

$$\frac{\mathit{canDeleteFile}\ \mathrm{f\ S}\ =\ \mathit{Success}}{\mathrm{S} \vdash\ \mathit{DeleteFile}\ <\mathrm{f}> \Rightarrow\ \mathit{Success},\ ((\mathrm{File\ of\ S})\ \ominus\ \mathrm{f})\ \mathrm{in\ S}} \tag{74}$$

$$\frac{\mathit{canDeleteFile}\ \mathrm{f\ S}\ =\ \mathit{Failure}\ \mathrm{v}}{\mathrm{S} \vdash\ \mathit{DeleteFile}\ <\mathrm{f}> \Rightarrow\ \mathit{Failure}\ \mathrm{v}, \mathrm{S}} \tag{75}$$

$$\frac{\mathit{statusString}\ \mathrm{f\ S}\ =\ \mathrm{v}}{\mathrm{S} \vdash\ \mathit{StatusFile}\ <\mathrm{f}> \Rightarrow\ \mathrm{v}, \mathrm{S}} \tag{76}$$

$$\frac{\mathit{statusChanString}\ \mathrm{c\ S}\ =\ \mathrm{v}}{\mathrm{S} \vdash\ \mathit{StatusChan}\ <\mathrm{c}> \Rightarrow\ \mathrm{v\ S}} \tag{77}$$

$$\frac{}{\mathrm{S} \vdash\ \mathit{GetProgName} \Rightarrow\ \mathit{Str}\ <\mathrm{Prog\ of\ S}>, \mathrm{S}} \tag{78}$$

$$\frac{}{\mathrm{S} \vdash\ \mathit{GetArgs} \Rightarrow\ \mathit{StrList}\ <\mathrm{Args\ of\ S}>, \mathrm{S}} \tag{79}$$

$$\frac{\begin{array}{c} \mathit{canWriteBinChan}\ \mathrm{c}\ \mathrm{S}\ \mathrm{s}\ =\ <\mathit{Success}, \mathrm{v}> \\ \mathrm{S'}\ =\ (\mathrm{Channel\ of\ S})\ \vec{\oplus}\ \{\mathrm{c} : \mathrm{v}\} \end{array}}{\mathrm{S} \vdash\ \mathit{AppendBinChan}\ <\mathrm{c}, \mathrm{s}> \Rightarrow\ \mathit{Success},\ \mathrm{S}\ \vec{\oplus}\ \mathrm{S'}} \tag{64}$$

$$\frac{\mathit{canWriteBinChan}\ \mathrm{c}\ \mathrm{S}\ \mathrm{True}\ \mathrm{s}\ =\ <\mathit{Failure}\ \mathrm{f}, \mathrm{v}>}{\mathrm{S} \vdash\ \mathit{AppendBinChan}\ <\mathrm{c}, \mathrm{s}> \Rightarrow\ \mathit{Failure}\ \mathrm{f}, \mathrm{S}} \tag{65}$$

$$\frac{\begin{array}{c} \mathit{canWriteTextFile}\ \mathrm{f}\ \mathrm{S}\ \mathrm{True}\ \mathrm{s}\ =\ <\mathit{Success}, \mathrm{v}> \\ \mathrm{S'}\ =\ (\mathrm{File\ of\ S})\ \vec{\oplus}\ \{\mathrm{f} : \mathrm{v}\} \end{array}}{\mathrm{S} \vdash\ \mathit{AppendFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Success},\ \mathrm{S}\ \vec{\oplus}\ \mathrm{S'}} \tag{66}$$

$$\frac{\mathit{canWriteTextFile}\ \mathrm{f}\ \mathrm{S}\ \mathrm{True}\ \mathrm{s}\ =\ <\mathit{Failure}\ \mathrm{v}, \mathrm{v'}>}{\mathrm{S} \vdash\ \mathit{AppendFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Failure}\ \mathrm{v}, \mathrm{S}} \tag{67}$$

$$\frac{\begin{array}{c} \mathit{canWriteBinFile}\ \mathrm{f}\ \mathrm{S}\ \mathrm{True}\ \mathrm{s}\ =\ <\mathit{Success}, \mathrm{v}> \\ \mathrm{S'}\ =\ (\mathrm{File\ of\ S})\ \vec{\oplus}\ \{\mathrm{f} : \mathrm{v}\} \end{array}}{\mathrm{S} \vdash\ \mathit{AppendBinFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Success},\ \mathrm{S}\ \vec{\oplus}\ \mathrm{S'}} \tag{68}$$

$$\frac{\mathit{canWriteBinFile}\ \mathrm{f}\ \mathrm{S}\ \mathrm{True}\ \mathrm{s}\ =\ <\mathit{Failure}\ \mathrm{v}, \mathrm{v'}>}{\mathrm{S} \vdash\ \mathit{AppendBinFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Failure}\ \mathrm{v}, \mathrm{S}} \tag{69}$$

$$\frac{\begin{array}{c} \mathit{canWriteTextFile}\ \mathrm{f}\ \mathrm{S}\ \mathrm{False}\ \mathrm{s}\ =\ <\mathit{Success}, \mathrm{v}> \\ \mathrm{S'}\ =\ (\mathrm{File\ of\ S})\ \vec{\oplus}\ \{\mathrm{f} : \mathrm{v}\} \end{array}}{\mathrm{S} \vdash\ \mathit{WriteFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Success},\ \mathrm{S}\ \vec{\oplus}\ \mathrm{S'}} \tag{70}$$

$$\frac{\mathit{canWriteTextFile}\ \mathrm{f}\ \mathrm{S}\ \mathrm{False}\ \mathrm{s}\ =\ <\mathit{Failure}\ \mathrm{v}, \mathrm{v'}>}{\mathrm{S} \vdash\ \mathit{WriteFile}\ <\mathrm{f}, \mathrm{s}> \Rightarrow\ \mathit{Failure}\ \mathrm{v}, \mathrm{S}} \tag{71}$$

$$\frac{\begin{array}{c} \mathit{canReadBinChan}\ \mathrm{c}\ \mathrm{S}\ =\ <\mathit{Success}, \mathrm{b}> \\ \mathrm{S'}\ =\ (\mathrm{Channel}\ \mathit{of}\ \mathrm{S}\ \ominus\ \mathrm{c})\ \mathrm{in}\ (\mathit{noEcho}\ \mathrm{c}\ \mathrm{S}) \end{array}}{\mathrm{S} \vdash\ \mathit{ReadBinChan}\ <\mathrm{c}> \Rightarrow\ \mathit{Bn}\ <\mathrm{b}>, \mathrm{S'}} \tag{56}$$

$$\frac{\begin{array}{c} \mathit{canReadBinChan}\ \mathrm{c}\ \mathrm{S}\ =\ <\mathit{Failure}\ \mathrm{f}, \mathrm{b}> \\ \mathrm{S'}\ =\ (\mathrm{Channel}\ \mathit{of}\ \mathrm{S}\ \ominus\ \mathrm{c})\ \mathrm{in}\ (\mathit{noEcho}\ \mathrm{c}\ \mathrm{S}) \end{array}}{\mathrm{S} \vdash\ \mathit{ReadBinChan}\ <\mathrm{c}> \Rightarrow\ \mathit{Failure}\ \mathrm{f}, \mathrm{S'}} \tag{57}$$

$$\frac{\mathit{canReadTextFile}\ \mathrm{f}\ \mathrm{S}\ =\ <\mathit{Success}, \mathrm{s}>}{\mathrm{S} \vdash\ \mathit{ReadFile}\ <\mathrm{f}> \Rightarrow\ \mathit{Str}\ <\mathrm{s}>, \mathrm{S}} \tag{58}$$

$$\frac{\mathit{canReadTextFile}\ \mathrm{f}\ \mathrm{S}\ =\ <\mathit{Failure}\ \mathrm{v}, \mathrm{s}>}{\mathrm{S} \vdash\ \mathit{ReadFile}\ <\mathrm{f}> \Rightarrow\ \mathit{Failure}\ \mathrm{v}, \mathrm{S}} \tag{59}$$

$$\frac{\mathit{canReadBinFile}\ \mathrm{f}\ \mathrm{S}\ =\ <\mathit{Success}, \mathrm{b}>}{\mathrm{S} \vdash\ \mathit{ReadBinFile}\ <\mathrm{f}> \Rightarrow\ \mathit{Bn}\ <\mathrm{b}>, \mathrm{S}} \tag{60}$$

$$\frac{\mathit{canReadBinFile}\ \mathrm{f}\ \mathrm{S}\ =\ <\mathit{Failure}\ \mathrm{v}, \mathrm{b}>}{\mathrm{S} \vdash\ \mathit{ReadBinFile}\ <\mathrm{f}> \Rightarrow\ \mathit{Failure}\ \mathrm{v}, \mathrm{S}} \tag{61}$$

Rules (62)–(73) define the semantics of writing or appending values to files or channels.

$$\frac{\begin{array}{c} \mathit{canWriteTextChan}\ \mathrm{c}\ \mathrm{S}\ \mathrm{s}\ =\ <\mathit{Success}, \mathrm{v}> \\ \mathrm{S'}\ =\ (\mathrm{Channel\ of\ S})\ \vec{\oplus}\ \{\mathrm{c} : \mathrm{v}\} \end{array}}{\mathrm{S} \vdash\ \mathit{AppendChan}\ <\mathrm{c}, \mathrm{s}> \Rightarrow\ \mathit{Success},\ \mathrm{S}\ \vec{\oplus}\ \mathrm{S'}} \tag{62}$$

$$\frac{\mathit{canWriteTextChan}\ \mathrm{c}\ \mathrm{S}\ \mathrm{s}\ =\ <\mathit{Failure}\ \mathrm{f}, \mathrm{v}>}{\mathrm{S} \vdash\ \mathit{AppendChan}\ <\mathrm{c}, \mathrm{s}> \Rightarrow\ \mathit{Failure}\ \mathrm{f}, \mathrm{S}} \tag{63}$$

state $S'''$. In a single-tasking system, we might expect this transformation to be the identity function.

$$\frac{\overset{\text{OS}}{\vdash} \text{S} \Rightarrow \text{S'}}{\ll \text{match, E} \gg, \text{S} \mid\!= \bot \Rightarrow \bot, \text{S'}} \tag{51}$$

$$\frac{\overset{\text{OS}}{\vdash} \text{S} \Rightarrow \text{S'}}{\ll \text{match, E} \gg, \text{S} \mid\!= \texttt{Nil} <> \Rightarrow \texttt{Nil} <>, \text{S'}} \tag{52}$$

$$\frac{\begin{array}{c}\text{S} \vdash \text{v} \Rightarrow \text{v'}, \text{S'} \\ \text{E}, \text{v'} \mid\!= \text{match} \Rightarrow \text{v"} \\ \text{v"} \;=\; \texttt{(:)} < \text{v}, \text{vs'} > \\ \ll \text{match, E} \gg, \text{S'} \mid\!= \text{vs'} \Rightarrow \text{vs"}, \text{S"} \\ \overset{\text{OS}}{\vdash} \text{S"} \Rightarrow \text{S"'}\end{array}}{\ll \text{match, E} \gg, \text{S} \mid\!= \texttt{(:)} < \text{v}, \text{vs} > \Rightarrow \texttt{(:)} < \text{v"}, \text{vs"} >, \text{S"}} \tag{53}$$

The remaining rules construct a response and state transition from a request and an initial state, $S$. To each remaining rule is attached the implicit side condition $\bot \notin \text{v}_i$, where the $\text{v}_i$ are the parameters of the request. Together with rule (51), this ensures that requests (though not responses) are hyper-strict, as required by the Haskell report. The majority of requests have two rules: one used when the request succeeds, and another used when the request fails.

Section 7.1 defines the auxiliary functions *read*, *write*, *canReadTextChan* etc. which are used here.

Rules (54)–(61) define the semantics of reading values from files or channels.

$$\frac{\begin{array}{c}\mathit{canReadTextChan}\ \text{c S} \;=\; < \mathit{Success}, \text{s} > \\ \text{S'} \;=\; (\text{Channel } \mathit{of}\ \text{S} \ominus \text{c}) \text{ in } (\mathit{noEcho}\ \text{c S})\end{array}}{\text{S} \vdash \mathit{ReadChan} < \text{c} > \Rightarrow \mathit{Str} < \text{s} >, \text{S'}} \tag{54}$$

$$\frac{\begin{array}{c}\mathit{canReadTextChan}\ \text{c S} \;=\; < \mathit{Failure}\ \text{f}, \text{s} > \\ \text{S'} \;=\; (\text{Channel } \mathit{of}\ \text{S} \ominus \text{c}) \text{ in } (\mathit{noEcho}\ \text{c S})\end{array}}{\text{S} \vdash \mathit{ReadChan} < \text{c} > \Rightarrow \mathit{Failure}\ \text{f}, \text{S'}} \tag{55}$$

The semantics does not consider the possibility of *FormatError* when writing the standard channels *stdout*, *stderr* or *stdecho*. Issues of file and channel interdependence are also ignored. All these issues are operating-system dependent. We consider only the required filesystem requests here, optional requests are ignored.

The meaning of a program is defined with respect to $S_0$, the initial state of the filesystem and $E_0$, the initial environment. The initial value of the "echo-processed" flag in $S_0$ should be *False*; that is no *Echo* request has been issued initially. This flag is set to *True* after the first echo request or read from the distinguished channel *stdin*.

The meaning of a set of modules is a closure, which when applied to a list of responses yields a list of requests. The request is applied to the state to determine a response and a new state. Rule (48) defines the meaning of a program. The meaning of the modules comprising the program, *mods*, will be a closure, *v*. This closure is evaluated in the context of the initial state $S_0$ to give the value result of the program, *v'* and a new state $S'$.

$$\frac{\mathrm{E}_0 \vdash \mathrm{mods} \Rightarrow \mathrm{v} \qquad \mathrm{S}_0 \vdash \mathrm{v} \Rightarrow \mathrm{v'}, \mathrm{S'}}{\mathrm{E}_0, \mathrm{S}_0 \vdash \mathrm{mods} \Rightarrow \mathrm{v'}, \mathrm{S'}} \tag{48}$$

Rules (49) and (50) define the meaning of the program if the main function returns $\perp$ or a closure, respectively. In the latter case, *match* is matched against $\perp$ to give a list of requests *v*. This list of requests is then evaluated in the context of the state to give a list of responses *v'* and a new state $S'$.

$$\frac{}{\mathrm{S} \vdash \perp \Rightarrow \perp} \tag{49}$$

$$\frac{\mathrm{E}, \perp \models \mathrm{match} \Rightarrow \mathrm{v} \qquad \ll \mathrm{match},\ \mathrm{E} \gg, \mathrm{S} \models \mathrm{v} \Rightarrow \mathrm{v'}, \mathrm{S'}}{\mathrm{S} \vdash \ll \mathrm{match},\ \mathrm{E} \gg \Rightarrow \mathrm{v'}, \mathrm{S'}} \tag{50}$$

Rules (51)–(53) map lists of requests to lists of responses. We use the same semantic list constructor (:) here as elsewhere in this report. Since any component of the list may be $\perp$, there are three rules.

The third rule (for (:)) is the most interesting. The first request is evaluated in the context of *S* to give a response *v'* and a new state $S'$. The response is then matched against *match* to return a list of requests *v"*. The first of these requests must be the the same as *v*. The remaining requests are evalauted in the context of the new state to give a corresponding list of responses, and the associated state, $S''$. The operating system is free to transform this state to give a final

$$\frac{\begin{array}{c}\text{pat grhs} \neq \text{var} = \text{exp} \\ \forall i.\ 1 \leq i \leq \#\text{fv(pat)},\ \ \text{var}_i = \text{fv(pat)} \\ \text{E} \vdash \text{var}_i = \texttt{case}\ \text{exp}\ \texttt{of}\ \{\text{pat} \mid \text{gd} = \text{var}_i\} \Rightarrow \text{E}_i\end{array}}{\text{E} \vdash \text{pat} \mid \text{gd} = \text{exp} \Rightarrow \bigoplus (\forall i.\ 1 \leq i \leq \#\text{fv(pat)},\ \text{E}_i)} \tag{44}$$

$$\frac{\text{E} \vdash \text{exp} \Rightarrow \text{v}}{\text{E} \vdash \text{var} = \text{exp} \Rightarrow \{\text{var} : \text{v}\}} \tag{45}$$

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n,\ \ \text{E} \oplus \text{E'} \vdash \text{def}_i \Rightarrow \text{E}_i \\ \text{E'} = \bigoplus (\forall j.\ 1 \leq j \leq n,\ \text{E}_j)\end{array}}{\text{E} \vdash \{\ \text{def}_1, \ldots, \text{def}_n\} \Rightarrow \text{E'}} \tag{46}$$

# 6 Modules

The meaning of a group of modules is the meaning of the function *main* from the module *Main*. This defines a function from *responses* to *requests* which is evaluated lazily in the context of an initial operating system state. The result of the evaluation is a series of operating system state transitions which collectively define the visible effect of executing the Haskell program.

All definitions are potentially mutually recursive, so we define their meaning in the value environment which comprises all definitions in the set of modules, plus the initial environment. Note that the use of $\oplus$ rather than $\vec{\oplus}$ ensures that aliasing of original names cannot occur. We assume that modules are consistent: this is enforced by the static semantics.

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n,\ \ \text{E'} \vdash \text{mod}_i \Rightarrow \text{E}_i \\ \text{E'} = \text{E} \oplus (\bigoplus (\forall j.\ 1 \leq j \leq n, \text{E}_i)) \\ \text{E'} \vdash \textit{originalname}\ (\text{Main}, \text{main}) \Rightarrow \text{v}\end{array}}{\text{E} \vdash \{\text{mod}_1, \ldots, \text{mod}_n\} \Rightarrow \text{v}}\ \ n \geq 0 \tag{47}$$

# 7 Programs and Request Semantics

This section defines the meaning of programs and their interaction with the file-system by means of the Haskell request/response mechanism.

$$\frac{\mathrm{v} \;=\; \bot}{\mathrm{E,v} \models \{() \;=\; \mathrm{exp}\,[]\,_ \;=\; \mathrm{exp'}\} \Rightarrow \bot} \tag{39}$$

$$\frac{\begin{array}{c}\forall i,j.\ 0 < i \leq n, 0 < j \leq n, \\ \mathrm{var}_i \notin \left(\bigcup ((\mathrm{fv}(\mathrm{pat}_j)) \cup \mathrm{fv}(\mathrm{exp})\right) \\ \mathrm{E,v} \models \left\{\begin{array}{l}(\mathrm{var}_1,\ldots,\mathrm{var}_n) \;= \\ \quad \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{\mathrm{pat}_1 \;=\; \ldots \\ \quad \texttt{case}\ \mathrm{var}_n\ \texttt{of}\ \{\mathrm{pat}_n \;=\; \mathrm{exp}\,[]\,_ \;=\; \mathrm{exp'}\}\ldots\}\end{array}\right\} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{(\mathrm{pat}_1,\ldots,\mathrm{pat}_n) \;=\; \mathrm{exp}\,[]\,_ \;=\; \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{40}$$

$$\frac{\begin{array}{c}\mathrm{v} \;=\; <\mathrm{v}_1,\ldots,\mathrm{v}_n> \\ \forall i.\ 0 < i \leq n,\ \ \mathrm{E} \oplus \{\mathrm{var}_i : \mathrm{v}_i\} \vdash \mathrm{exp} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{(\mathrm{var}_1,\ldots,\mathrm{var}_n) \;=\; \mathrm{exp}\,[]\,_ \;=\; \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{41}$$

$$\frac{\mathrm{v} \;=\; \bot}{\mathrm{E,v} \models \{(\mathrm{var}_1,\ldots,\mathrm{var}_n) \;=\; \mathrm{exp}\,[]\,_ \;=\; \mathrm{exp'}\} \Rightarrow \bot} \tag{42}$$

# 5 Definitions

This section presents the semantics for definitions. These rules are used to derive value environments. Rule (46) defines the semantics for a set of bindings. Since all bindings are recursive, the resulting environment sum can be used to derive the meanings of each individual binding. This is exploited in rule (43), where the environment sum resulting from a set of recursive pattern bindings is used to define the meaning of those bindings.

$$\frac{}{\mathrm{E} \vdash \mathrm{var\ match} \Rightarrow \{\mathrm{var} :\ \ll \mathrm{match},\ \mathrm{E} \gg\}} \tag{43}$$

$$\frac{\mathrm{E} \oplus \{\mathrm{var} : \mathrm{v}\} \vdash \mathrm{exp} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{var} = \mathrm{exp}\} \Rightarrow \mathrm{v'}} \tag{33}$$

Rules (34)–(42) define the semantics of matches against constructor patterns. Rules (34) and (40) are simplification rules, simplifying general constructor matches and tuple matches, respectively; the remaining rules define the matching semantics. The simplification rules are used to simplify deep pattern matches (such as [1,2]) into single-level matches. Note the explicit use of $\perp$ in the matching rules.

$$\forall i, j.\ 1 \leq i \leq n, 0 < j \leq n, \quad \mathrm{var}_i \notin \bigcup (\mathrm{fv}(\mathrm{pat}_j) \cup \mathrm{fv}(\mathrm{exp}))$$

$$\frac{\mathrm{E}, \mathrm{v} \models \left\{ \begin{array}{l} \mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n\ = \\ \quad \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{\mathrm{pat}_1\ =\ \ldots \\ \quad \texttt{case}\ \mathrm{var}_n\ \texttt{of}\ \{\mathrm{pat}_n\ =\ \mathrm{exp} \,[]\, _\ =\ \mathrm{exp'}\} \ldots\} \end{array} \right\} \Rightarrow \mathrm{v'}}{\mathrm{E}, \mathrm{v} \models \{\mathrm{con}\ \mathrm{pat}_1 \ldots \mathrm{pat}_n\ =\ exp \,[]\, _\ =\ \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{34}$$

$$\frac{\begin{array}{c} \mathrm{v} = \mathrm{con} < \mathrm{v}_1, \ldots, \mathrm{v}_n > \\ \forall i.\ 1 \leq i \leq n, \quad \mathrm{E} \stackrel{\rightarrow}{\oplus} \{\mathrm{var}_i : \mathrm{v}_i\} \vdash \mathrm{exp} \Rightarrow \mathrm{v'} \end{array}}{\mathrm{E,v} \models \{\mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n\ =\ \mathrm{exp} \,[]\, _\ =\ \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{35}$$

$$\frac{\begin{array}{c} \mathrm{v} \neq \mathrm{con} < \mathrm{v}_1, \ldots, \mathrm{v}_n > \ \wedge\ \mathrm{v} \neq \perp \\ \mathrm{E} \vdash \mathrm{exp'} \Rightarrow \mathrm{v'} \end{array}}{\mathrm{E,v} \models \{\mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n\ =\ \mathrm{exp} \,[]\, _\ =\ \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{36}$$

$$\frac{\mathrm{v} = \perp}{\mathrm{E,v} \models \{\mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n\ =\ \mathrm{exp} \,[]\, _\ =\ \mathrm{exp'}\} \Rightarrow \perp} \tag{37}$$

$$\frac{\begin{array}{c} \mathrm{v} = <> \\ \mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v'} \end{array}}{\mathrm{E,v} \models \{() = \mathrm{exp} \,[]\, _\ =\ \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{38}$$

$$\frac{\mathrm{E,v} \models \left\{ \begin{array}{ll} \mathrm{pat} = & \texttt{if}\ \mathrm{gd}_1\ \texttt{then}\ \mathrm{exp}_1\ \texttt{else}\ \ldots \\ & \texttt{if}\ \mathrm{gd}_n\ \texttt{then}\ \mathrm{exp}_n\ \texttt{else}\ \mathrm{exp'} \\ [] _ = \mathrm{exp'} & \end{array} \right\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{pat} \mid \mathrm{gd}_1 = \mathrm{exp}_1 \mid \ldots \mid \mathrm{gd}_n = \mathrm{exp}_n\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{26}$$

Rules (27)–(33) define the semantics of variable matches, lazy pattern matches and as-patterns.

$$\mathrm{x}_0 \notin \mathrm{fv}(\mathrm{exp})$$
$$\forall i.\ 1 \leq i \leq \#\mathrm{fv}(\mathrm{pat}),\quad \mathrm{x}_i \in \mathrm{fv}(\mathrm{pat})$$

$$\frac{\mathrm{E,v} \models \{\mathrm{x}_0 = \texttt{let}\ \{\mathrm{x}_i = \texttt{case}\ \mathrm{x}_0\ \texttt{of}\ \{\mathrm{pat} = \mathrm{x}_i\}\}\ \texttt{in}\ \mathrm{exp}\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\sim \mathrm{pat} = \mathrm{exp}\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{27}$$

$$\frac{\mathrm{E,v} \models \{\mathrm{var} = \texttt{case}\ \mathrm{var}\ \texttt{of}\ \{\mathrm{pat} = \mathrm{exp}\}\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{var@pat} = \mathrm{exp}\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{28}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{_ = \mathrm{exp}\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{29}$$

$$\frac{\mathrm{E,v} \models \{\mathrm{var} = \mathrm{exp}\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{var} : \tau = \mathrm{exp}\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{30}$$

$$\frac{\mathrm{E,v} \models \{\mathrm{var} = \mathrm{exp}\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{var} : \kappa\ \tau = \mathrm{exp}\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{31}$$

$$\frac{\mathrm{E,v} \models \{\mathrm{var} = \mathrm{exp}\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{var} = \mathrm{exp}\ [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{32}$$

a less direct semantics than that of, e.g., Standard ML, but we feel this approach is justified because of the extra complications of lazy pattern matching (introduced by $\sim$ patterns) and guarded patterns.

Rule (23) simplifies multi-argument matches to single-argument matches.

$$\frac{\begin{array}{l}\forall i,j.\ 0 < i \le m,\ 0 < j \le n,\quad x_i \notin (\mathrm{fv}(\mathrm{pat}_{ij}) \cup \mathrm{fv}(\mathrm{grhs}_i)) \\ \mathrm{E,v} \models \left\{ \begin{array}{l} \mathrm{x}_1 = \backslash \mathrm{x}_2 = \ldots \backslash \mathrm{x}_n = \\ \quad \texttt{case}\ (\mathrm{x}_1, \ldots, \mathrm{x}_n)\ \texttt{of} \\ \qquad \{\ (\mathrm{pat}_{11}, \ldots, \mathrm{pat}_{1n})\ \mathrm{grhs}_1 \ [] \ \ldots \ [] \\ \qquad\quad (\mathrm{pat}_{m1}, \ldots, \mathrm{pat}_{mn})\ \mathrm{grhs}_m \} \end{array} \right\} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{\mathrm{pat}_{11} \ldots \mathrm{pat}_{1n}\ \mathrm{grhs}_1 \ [] \ \ldots \ [] \ \mathrm{pat}_{m1} \ldots \mathrm{pat}_{mn}\ \mathrm{grhs}_m\} \Rightarrow \mathrm{v'}} \quad m \ge 1, n \ge 2 \tag{23}$$

Rule (24) simplifies matches into matches of the form $\{\mathrm{pat} = \mathrm{exp}\ [] _ = \mathrm{exp'}\}$.

$$\frac{\begin{array}{c}\forall i.\ 1 \le i \le n,\quad x_i \notin \bigcup (\mathrm{fv}(\mathrm{pat}_i) \cup \mathrm{fv}(\mathrm{grhs}_i)) \\ \mathrm{pat}_n\ \mathrm{grhs}_n \ne _ = \mathrm{exp}_n \\ \mathrm{E,v} \models \left\{ \begin{array}{l} \mathrm{pat}_1\ \mathrm{grhs}_1 \\ []\ \mathrm{x}_1 = \texttt{case}\ \mathrm{x}_1\ \texttt{of}\ \{\mathrm{pat}_2\ \mathrm{grhs}_2 \\ []\ \mathrm{x}_2 = \texttt{case}\ \mathrm{x}_2\ \texttt{of}\ \{\mathrm{pat}_3\ \mathrm{grhs}_3 \ [] \ \ldots [] _ = \mathit{caseError}\}\ \} \end{array} \right\} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{\mathrm{pat}_1\ \mathrm{grhs}_1 \ [] \ldots [] \ \mathrm{pat}_n \mathrm{grhs}_n\} \Rightarrow \mathrm{v'}} \quad n \ge 1 \tag{24}$$

Rules (25)–(26) simplify guarded matches. Guards are translated into conditional expressions.

$$\frac{\mathrm{E,v} \models \left\{ \begin{array}{ll} \mathrm{pat} = & \texttt{if}\ \mathrm{gd}_1\ \texttt{then}\ \mathrm{exp}_1\ \texttt{else}\ \ldots \\ & \texttt{if}\ \mathrm{gd}_n\ \texttt{then}\ \mathrm{exp}_n\ \texttt{else}\ \mathrm{exp} \\ [] _ = \mathrm{exp'} & \end{array} \right\} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\mathrm{pat} \mid \mathrm{gd}_1 = \mathrm{exp}_1 \mid \ldots \mid \mathrm{gd}_n = \mathrm{exp}_n = \mathrm{exp} [] _ = \mathrm{exp'}\} \Rightarrow \mathrm{v'}} \tag{25}$$

Rules (18)–(20) give the semantics of let-expressions. Rule (18) applies when there are multiple bindings in a let. Rules (19) and (20) handle non-recursive and recursive bindings, respectively.

$$\frac{\mathrm{E} \vdash \texttt{let}\ \{(\sim \mathrm{pat}_1, \ldots, \sim \mathrm{pat}_n)\ =\ (\mathrm{exp}_1, \ldots, \mathrm{exp}_n)\}\ \texttt{in}\ \mathrm{exp}\ \Rightarrow\ \mathrm{v}}{\mathrm{E} \vdash \texttt{let}\ \{\mathrm{pat}_1\ =\ \mathrm{exp}_1, \ldots, \mathrm{pat}_n\ =\ \mathrm{exp}_n\}\ \texttt{in}\ \mathrm{exp}\ \Rightarrow\ \mathrm{v}}\ n > 1 \tag{18}$$

$$\frac{\begin{array}{c}\mathrm{fv}(\mathrm{pat}) \cap \mathrm{fv}(\mathrm{exp})\ =\ \emptyset \\ \mathrm{E} \vdash \texttt{case}\ \mathrm{exp'}\ \texttt{of}\ \{\sim \mathrm{pat} = \mathrm{exp}\}\ \Rightarrow\ \mathrm{v}\end{array}}{\mathrm{E} \vdash \texttt{let}\ \{\mathrm{pat} = \mathrm{exp'}\}\ \texttt{in}\ \mathrm{exp}\ \Rightarrow\ \mathrm{v}} \tag{19}$$

$$\frac{\begin{array}{c}\mathrm{fv}(\mathrm{pat}) \cap \mathrm{fv}(\mathrm{exp}) \neq \emptyset \\ \mathrm{E} \vdash \texttt{case}\ \mathrm{exp'}\ \texttt{of}\ \{\sim \mathrm{pat}\ =\ \mathit{fix}\ (\backslash \sim \mathrm{pat}\ =\ \mathrm{exp})\}\ \Rightarrow\ \mathrm{v}\end{array}}{\mathrm{E} \vdash \texttt{let}\ \{\mathrm{pat}\ =\ \mathrm{exp'}\}\ \texttt{in}\ \mathrm{exp}\ \Rightarrow\ \mathrm{v}} \tag{20}$$

Rules (21) and (22) handle the second-order polymorphic $\lambda$-calculus forms introduced by the dictionary translations in the static semantics. These terms are used only to convey type information: neither type applications (rule 22) nor type abstractions (rule 21) have any meaning in this value-oriented semantics.

$$\frac{\mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \mathrm{v}}{\mathrm{E} \vdash \Lambda\ \alpha_1 \ldots \alpha_n \texttt{->}\ \mathrm{exp}\ \Rightarrow\ \mathrm{v}} \tag{21}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \mathrm{v}}{\mathrm{E} \vdash \mathrm{exp}\ \tau\ \Rightarrow\ \mathrm{v}} \tag{22}$$

# 4 Matches

For clarity, we use a different kind of turnstile ($|=$) for match inference rules. E,v $|=$ e $\Rightarrow$ v' defines the meaning of *match* with respect to a single matched value *e*. The semantics for definitions and applications ensures that matches are curried appropriately.

The semantics for pattern-matching is derived from that presented in the Haskell report for **case** expressions (where the semantics was defined as a translation into a Haskell kernel). This gives

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v} \qquad \mathrm{E,v} \models \mathrm{match} \Rightarrow \mathrm{v'}}{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}\ \texttt{of}\ \mathrm{match} \Rightarrow \mathrm{v'}} \tag{9}$$

$$\frac{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}_1\ \texttt{of}\ \{\mathrm{True} = \mathrm{exp}_2 \,[]\, \mathrm{False} = \mathrm{exp}_3\} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \texttt{if}\ \mathrm{exp}_1\ \texttt{then}\ \mathrm{exp}_2\ \texttt{else}\ \mathrm{exp}_3 \Rightarrow \mathrm{v}} \tag{10}$$

$$\frac{\mathrm{E} \vdash (:)\,\mathrm{exp}_1\,(\ldots((:)\mathrm{exp}_n\,[\,])\ldots) \Rightarrow \mathrm{v}}{\mathrm{E} \vdash [\mathrm{exp}_1, \ldots, \mathrm{exp}_n] \Rightarrow \mathrm{v}}\ \mathrm{n} \geq 1 \tag{11}$$

$$\frac{}{\mathrm{E} \vdash [\,] \Rightarrow \texttt{Nil}\ <>} \tag{12}$$

$$\frac{}{\mathrm{E} \vdash () \Rightarrow <>} \tag{13}$$

$$\frac{\forall i.\ 1 < i \leq n, \quad \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \mathrm{v}_i}{\mathrm{E} \vdash (\mathrm{exp}_1, \ldots, \mathrm{exp}_n) \Rightarrow < v_1, \ldots, v_n >} \tag{14}$$

Rules (15)–(17) define the semantics of list comprehensions, as specified in the Haskell report.

$$\frac{\mathrm{E} \vdash \mathit{concat}\,[\,[\,\mathrm{exp} \mid \mathrm{qual}_2\,]\mid \mathrm{qual}_1\,] \Rightarrow \mathrm{v}}{\mathrm{E} \vdash [\mathrm{exp} \mid \mathrm{qual}_1, \mathrm{qual}_2\,] \Rightarrow \mathrm{v}} \tag{15}$$

$$\frac{\mathrm{E} \vdash \mathit{map}\ (\backslash \mathrm{pat} = \mathrm{exp})\ \mathrm{exp'} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash [\mathrm{exp} \mid \mathrm{pat}\ \texttt{->}\ \mathrm{exp'}] \Rightarrow \mathrm{v}} \tag{16}$$

$$\frac{\mathrm{E} \vdash \texttt{if}\ \mathrm{exp'}\ \texttt{then}\ [\mathrm{exp}]\ \texttt{else}\ [\,] \Rightarrow \mathrm{v}}{\mathrm{E} \vdash [\mathrm{exp} \mid \mathrm{exp'}] \Rightarrow \mathrm{v}} \tag{17}$$

$$\frac{\mathrm{E}\ (\mathrm{var})\ =\ \mathrm{v}}{\mathrm{E} \vdash \mathrm{var} \Rightarrow \mathrm{v}} \tag{1}$$

$$\frac{\begin{array}{c}\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v} \\ \mathrm{E}_0\ (\mathrm{b})\ =\ \mathrm{exp}\end{array}}{\mathrm{E} \vdash \mathrm{b} \Rightarrow \mathrm{v}} \tag{2}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{con} \Rightarrow \mathrm{con}\ <>} \tag{3}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{char} \Rightarrow \mathrm{char}} \tag{4}$$

$$\frac{\mathit{sval}\,(\mathrm{string})\ =\ \mathrm{v}}{\mathrm{E} \vdash \mathrm{string} \Rightarrow \mathrm{v}} \tag{5}$$

$$\frac{\mathrm{E'} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \ll \mathrm{exp},\ \mathrm{E'} \gg \Rightarrow \mathrm{v}} \tag{6}$$

$$\frac{\begin{array}{c}\mathrm{E} \vdash \mathrm{exp} \Rightarrow \ll \mathrm{match},\ \mathrm{E'} \gg \\ \mathrm{E} \vec{\oplus} \mathrm{E'},\ \ll \mathrm{exp'},\ \mathrm{E} \gg \models \mathrm{match} \Rightarrow \mathrm{v}\end{array}}{\mathrm{E} \vdash \mathrm{exp\ exp'} \Rightarrow \mathrm{v}} \tag{7}$$

$$\frac{\begin{array}{c}\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{con}\ < \mathrm{v}_1 \ldots, \mathrm{v}_n > \\ \mathrm{E} \vdash \mathrm{exp'} \Rightarrow \mathrm{v}\end{array}}{\mathrm{E} \vdash \mathrm{exp\ exp'} \Rightarrow \mathrm{con}\ < \mathrm{v}_1, \ldots, \mathrm{v}_n, \mathrm{v} >} \tag{8}$$

| | | | |
|---|---|---|---|
| $b \in$ | BasVal = | $\{\text{PrimPlusInt}, \text{PrimEqInt}, \ldots\}$ | Basic Values |
| | BasCon = | $\{(:), \text{Nil}, \text{True}, \text{False}, \ldots\}$ | Basic Constructors |
| | Con = | BasCon + con | |
| | Closure = | $\ll$ match, Env $\gg$ | Closures |
| $c, f, v \in$ | SemVal = | BasVal + Con SemVal$^*$ + SemVal$^*$ + Closure | Semantic Values |
| E $\in$ | Env = | $\{$var $\mapsto$ SemVal+ BasVal $\mapsto$ exp $\}$ | Environments |
| $S \in$ | State = | Channel $\times$ File $\times$ Environ$\times$ Prog $\times$ Args $\times$ Echo | States |
| | Channel = | $\{\text{SemVal} \mapsto \text{SemVal}\}$ | Filesystem channels |
| | File = | $\{\text{SemVal} \mapsto \text{SemVal}\}$ | Files: name $\mapsto$ contents |
| | Environ = | $\{\text{SemVal} \mapsto \text{SemVal}\}$ | Program environment |
| | Prog = | SemVal | Program Name |
| | Args = | SemVal$^*$ | Program arguments |
| | Echo = | $<$ SemVal, SemVal $>$ | Echoing on/off; Echo allowed |

$m_1 \oplus m_2$ operator updates an environment mapping $m_1$ by the new mapping $m_2$. $m_1 \vec{\oplus} m_2$ is similar, but allows values in $m_1$ to be "shadowed" by those in $m_2$. $e \ominus m$ removes the mapping $m$ from an environment $e$.

We use two judgement forms $\vdash$ and $\models$. The latter is used for judgements on matches, of the form E,v $\models$ match $\Rightarrow$ v'; the former is used in all other cases. We use a special annotated version of the former judgement $\overset{\text{os}}{\vdash}$ S $\Rightarrow$ S' to indicate operating system transformations of the state. For obvious reasons, we do not define the effect of these transformations in this semantics.

# 3 Expressions

This section defines the semantic rules for expressions. The semantic function *sval* maps a semantic string into the semantic list of characters comprising the elements of that string. *fix* is the least fixpoint operator. *map* and *concat* are defined in the obvious way, ++ is used for list concatenation. *caseError* is used to define the semantics of match failure. It uses the *error* function from the Haskell prelude.

*map* f [ ] = [ ]
*map* f ((:) x xs) = (:) (f x) (*map* f xs)

*concat* [ ] = [ ]
*concat* ((:) x xs) = (++) x (*concat* xs)

*caseError* = *error* "Unexpected Match"

The first few rules handle the semantics for simple expressions. Note that the semantics for conditional expressions (rule 10) is defined in terms of the semantics for case-expressions (rule 9).

| | | | | |
|---|---|---|---|---|
| $exp$ | $\rightarrow$ | $var$ | | Variable |
| | $\mid$ | $con$ | | Constructor |
| | $\mid$ | $integer$ | | Integer |
| | $\mid$ | $float$ | | Real number |
| | $\mid$ | $char$ | | ASCII character |
| | $\mid$ | $string$ | | Literal string |
| | $\mid$ | \ $mrule$ | | Lambda abstraction |
| | $\mid$ | $exp\ exp'$ | | Function application |
| | $\mid$ | `case` $exp$ `of` $match$ | | Case expression |
| | $\mid$ | `if` $exp_1$ `then` $exp_2$ `else` $exp_3$ | | Conditional |
| | $\mid$ | `[`$exp$`\|`$quals$`]` | | List comprehension |
| | $\mid$ | `let` $binds$ `in` $exp$ | | Local definition |
| | $\mid$ | `[`$exp_1$`,` $\ldots$`,`$exp_n$`]` | $(n \geq 0)$ | List |
| | $\mid$ | `(`$exp_1$`,` $\ldots$`,`$exp_n$`)` | $(n = 0\ or\ n \geq 2)$ | Tuple |
| | $\mid$ | $\Lambda\alpha_1 \ldots \alpha_n$`->` $exp$ | | Type abstraction |
| | $\mid$ | $exp\ \tau_1 \ldots \tau_n$ | | Type application |
| $quals$ | $\rightarrow$ | $quals$`,`$quals$ | | Qualifiers |
| | $\mid$ | $pat$`<-`$exp$ | | Generator |
| | $\mid$ | $guard$ | | Filter |

Figure 2: Abstract syntax: Expressions

| | | | | |
|---|---|---|---|---|
| Monotypes $\tau$ | $\rightarrow$ | $\alpha$ | | Type variable |
| | $\mid$ | $\chi\ \tau_1 \ldots \tau$ | $(k \geq 0)$ | Structured type |
| | $\mid$ | $\tau_1 \rightarrow \tau_2$ | | Function type |
| | $\mid$ | $[\tau]$ | | List type |
| | $\mid$ | $(\tau_1, \ldots, \tau_k)$ | $(k = 0\ or\ k \geq 2)$ | Tuple type |
| | $\mid$ | `Bool` | | Boolean type |

Figure 3: Syntax of semantic types

| | | | | |
|---|---|---|---|---|
| $program$ | $\rightarrow$ | $\{mod_1, \ldots, mod_n\}$ | $(n > 0)$ | |
| $mod$ | $\rightarrow$ | $binds$ | | |
| $binds$ | $\rightarrow$ | $\{bind_1, \ldots, bind_n\}$ | $(n \geq 0)$ | Set of bindings |
| $bind$ | $\rightarrow$ | $pat\ grhs$ | | A pattern binding |
| | $\mid$ | $var\ match$ | | A function binding |
| $match$ | $\rightarrow$ | $match_1$ `[]` $match_2$ | | Case alternatives |
| | $\mid$ | $mrule$ | | |
| $mrule$ | $\rightarrow$ | $grhs$ | | |
| | $\mid$ | $pat\ mrule$ | | |
| $grhs$ | $\rightarrow$ | $grhs_1$ `[]` $grhs_2$ | | Guarded right-hand side |
| | $\mid$ | `\|` $guard$ `=` $exp$ | | |
| | $\mid$ | `=` $exp$ | | |
| $guard$ | $\rightarrow$ | $exp$ | | |
| $pat$ | $\rightarrow$ | $var$ | | Variable |
| | $\mid$ | `_` | | Wildcard |
| | $\mid$ | $con\ pat_1 \ldots pat_n$ | | Compound pattern |
| | $\mid$ | `~`$pat$ | | Lazy pattern |
| | $\mid$ | $var$`@`$pat$ | | As-pattern |
| | $\mid$ | `[`$pat_1, \ldots, pat_n$`]` | $(n \geq 0)$ | List pattern |
| | $\mid$ | `(`$pat_1, \ldots, pat_n$`)` | $(n \geq 2)$ | Tuple pattern |
| | $\mid$ | `()` | | Unit pattern |
| | $\mid$ | $var : \tau$ | | Translated patterns |
| | $\mid$ | $var : \kappa\ \tau$ | | Dictionary variables |

Figure 1: Abstract syntax: Definitions and Patterns

as evaluation order, or sharing. These details are, in our opinion, best covered by an operational semantics for a particular machine model rather than the general language semantics.

## 2 Notation

Except where noted, we use the same notation as the definition of Standard ML [HMT91]. The syntax of our source language is as defined for the static semantics of Haskell [JW92], but with the extension of patterns to include simple variables (*var*) in addition to typed variables (*var* : $\tau$). For clarity, this abstract syntax is summarised in Figures 1–3. A program comprises a set of modules, each of which is a set of bindings.

We assume these bindings may be mutually recursive. A small function translates the static semantics *binds* and *monobinds* into this simpler form.

$$
\begin{array}{lcl}
[\![\texttt{rec}\ m]\!] & = & [\![m]\!] \\
[\![b_1\ \texttt{then}\ b_2]\!] & = & \{[\![b_1]\!]\} \cup \{[\![b_2]\!]\} \\
 & & \\
[\![m_1\ ;\ m_2]\!] & = & \{[\![m_1]\!]\} \cup \{[\![m_2]\!]\} \\
[\![\{\}]\!] & = & \{\} \\
[\![m]\!] & = & \{m\}
\end{array}
$$

Our semantics is given in terms of the semantic domain *SemVal* defined below. A *Closure* comprises a syntactic *expression* object plus an environment, $\ll$ exp, E $\gg$ . This is used to define a non-strict semantics for function application. We use $\langle \ldots \rangle$ to enclose semantic tuples in the *SemVal* domain. This avoids confusion with the syntactic tuple domains, and allows the direct representation of 1-tuples where necessary. $D^*$ defines the domain of all tuples of $D$: $\langle\rangle, \langle D\rangle, \langle D, D\rangle \ldots$

Every semantic domain $D$ incorporates an implicit bottom element $\perp_D$. *BasVal* and *BasCon* are fully defined in Section 9.

# A Dynamic Semantics for Haskell (Draft)

Kevin Hammond and Cordelia Hall*

April 14, 1992

**Abstract**

This paper defines a dynamic semantics for the functional programming language Haskell. The semantics defines the meaning of expressions, modules and programs, including all non-optional I/O requests. All constructs in the Haskell language are considered. We use the same natural semantics style as the definition of Standard ML.

For brevity, we assume that the translations specified by the Wadler/Peyton Jones static semantics have been performed, and base our source language on the target language of the static semantics.

## 1 Introduction

This paper defines the dynamic semantics of the full Haskell language [HW90, HJW91, HJW92]. The dynamic semantics is complementary to the existing static semantics for Haskell [JW92] in two respects: firstly it defines the meaning of the second-order polymorphic lambda-calculus which is synthesised by that semantics; secondly it is defined using a natural semantic form, where meaning is expressed through judgement rules. This corresponds to the type inference judgements used in the static semantics.

The paper is designed as a short reference work, to be read in conjunction with the Haskell report. Consequently, only minimal explanatory text is provided here. We expect that other papers will provide a full commentary on the semantics, and take a more introductory slant.

We have not attempted to give an explicit semantics of graph-reduction. In the context of a language definition, such an approach seems both excessively low-level and unnecessarily restrictive. Rather, we have captured the essence of non-strict evaluation through the notion of "closures": syntactic values paired with an environment.

This semantics thus defines the meaning of a Haskell program by specifying its resulting value (in terms of state-transitions on an initial state), but leaves unspecified such operational details

*Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, UK. Tel: +44-41-339-8855 ext {5619,4984}. Email: {kh,cvh}@dcs.glasgow.ac.uk