Comparing Proofs about I/O in Three Programming Paradigms

Andrew Butterfield, Glenn Strong

November 19, 2001*

Contents

1 Introduction **4**
- 1.1 Methodology . . . 5

2 The I/O Model **7**
- 2.1 The World and the File-System . . . 8
- 2.2 The Operations . . . 9
  - 2.2.1 The fopen Operation . . . 9
  - 2.2.2 The fclose Operation . . . 10
  - 2.2.3 The fwritei Operation . . . 11
  - 2.2.4 The freadi Operation . . . 11
- 2.3 I/O Model Signature Summary . . . 12
- 2.4 Connecting I/O Model to Abstracted Programs . . . 12

3 Abstract Syntaxes **13**
- 3.1 Common Syntax . . . 13
  - 3.1.1 Common Expressions . . . 13
  - 3.1.2 Functional Language Expressions . . . 13
- 3.2 C Abstract Syntax . . . 14
  - 3.2.1 C Statements . . . 14
  - 3.2.2 C Programs . . . 14
- 3.3 Clean Abstract Syntax . . . 14
  - 3.3.1 Clean Expressions . . . 14
  - 3.3.2 Clean Hash Elements . . . 14
  - 3.3.3 Clean Programs . . . 14
- 3.4 Haskell Abstract Syntax . . . 15
  - 3.4.1 Haskell Expressions . . . 15

*Updated version, January 18, 2002

3.4.2 Haskell Monadic Statements 15
3.4.3 Haskell Programs 15

4 Real Programs **15**
4.1 The real C program 15
4.2 The real Clean program 16
4.3 The real Haskell program 16

5 Abstracted Programs **16**
5.1 The IO abstraction 17
5.2 Concrete Programs using IO Abstraction 17
5.2.1 The abstracted C program 17
5.2.2 The abstracted Clean program 17
5.2.3 The abstracted Haskell program 18
5.3 Abstract Syntax Forms 18
5.3.1 Abstract Syntax for C Program 18
5.3.2 Abstract Syntax for Clean Program 19
5.3.3 Abstract Syntax for Haskell Program 20

6 Denotational Semantics **20**
6.1 Common Semantic Domains 20
6.1.1 Value Semantic Domain 20
6.1.2 Environments 21
6.1.3 Handles/References 21
6.1.4 Overall Environment 22
6.1.5 Denotation Functions 22
6.1.6 Note on Type-Correctness 22
6.2 C Denotational Semantics 22
6.2.1 C Program State 22
6.2.2 C Program Denotations 23
6.2.3 C Statement Denotations 23
6.2.4 C Expression Denotations 23
6.2.5 C Builtin I/O denotations 23
6.3 Clean Denotational Semantics 24
6.3.1 Clean Program State 24
6.3.2 Clean Program Denotations 24
6.3.3 Clean Expression Denotations 25
6.3.4 Clean Pattern Match 25
6.3.5 Clean Builtin Function Denotations 25
6.4 Haskell Denotational Semantics 25

7 Denotational Proofs **26**
7.1 The Property 26
7.2 Proof for C Program 26
7.2.1 C Program Labelled Syntax 26
7.2.2 The Proof 27
7.2.3 Lemma Cd.1 29
7.2.4 Lemma Cd.2 29
7.2.5 Lemma Cd.3 30
7.2.6 Lemma Cd.4 30
7.3 Proof for Clean Program 32
7.3.1 Clean Program Labelled Syntax 32
7.3.2 The Proof 33
7.3.3 Lemma Kd.1 33

8 Language-Based Semantics **34**
8.1 C Language Semantics 34
8.1.1 Hoare Triple Rules 34
8.1.2 wp-rules 34
8.1.3 C Program Language Semantics 34
8.1.4 I/O Model in Hoare Triple Form 35
8.1.5 IO Model in C Language form 36
8.2 Clean Language Semantics 40
8.3 IO Model in Clean Language Form 40
8.4 Haskell Language Semantics 41
8.5 IO Model in Haskell Language Form 42

9 Language-Based Proofs **42**
9.1 C Language Proof 43
9.1.1 Condition Annotated Program. 43
9.1.2 C Statement 1 43
9.1.3 C Statement 2 45
9.1.4 C Statement 3 46
9.1.5 C Statement 4 47
9.1.6 C Statement 5 48
9.1.7 C Statement 6 49
9.1.8 Finishing the Proof 50
9.1.9 Lemma C.1 51
9.1.10 Lemma C.2 52
9.2 Clean Language Proof 52
9.2.1 Lemma K.1 54

9.2.2 Lemma K.1.1 55
9.2.3 Lemma K.2 56
9.2.4 Lemma K.2.1 56
9.2.5 Lemma K.3 56
9.2.6 Lemma K.3.1 57
9.2.7 Lemma K.4 57
9.2.8 Lemma K.5 57
9.2.9 Lemma K.6 57
9.3 Haskell Language Proof 58

10 Lemmas for Haskell proof **61**
10.1 Lemma H.1 61
10.1.1 Lemma H.2 61
10.1.2 Lemma H.3 62
10.1.3 Lemma H.4 62
10.1.4 Lemma H.5 63

1 Introduction

An often cited advantage of functional programming languages is that they are supposed to be easier to reason about than imperative languages [BW88, p1],[PJ87, p1],[Bd87, p23],[BJLM91, p17],[Hen87, pp6–7],[Dav92, p5] with the property of *referential transparency* getting a prominent mention and the notion of *side-effect* being deprecated all round. For a long time, a major disadvantage of functional programming languages was their inability to adequately handle features where side-effects are an intrinsic component, such as file or other I/O operations [BJLM91, p139],[Gor94, p-xi]. However, two methodologies have emerged in the last decade to combine the side-effect world of I/O with the referentially transparent world of functional programming, namely the *uniqueness type system* of the programming language Clean [BS00] and the use of *monads* in the Haskell language [Gor94][Bir98, Chp 10, pp326–359].

However, as a consequence of these developments, functional programs written in these languages now look very like imperative programs — as evidenced by sample programs appearing later in this paper. This immediately raises concerns about the *relative* ease of reasoning about such programs, when compared to similar programs done in an imperative style.

Question: Has the technical machinery necessary to handle I/O in pure functional languages, led to a situation where correctness proofs have the same difficulty as those found in imperative programs ?

Question: Can these same technical developments be applied to imperative programs in order to make it easier to reason about them ?

In other words, have we ended up in a situation where there is little to choose between functional and imperative languages when it comes to reasoning about "real-world" programs that interact with the environment in an effective manner ?

A second issue concerns the relative ease of reasoning when using either of the two technical alternatives, namely uniqueness typing and/or monads. The uniqueness typing approach uses the type-system to ensure that the external "world" is accessed in an single-threaded fashion, so that an underlying implementation can safely implement operations on the world using side-effects, while still maintaining referential transparency. From the programmer's perspective nothing changes in the program, except that it must satisfy the type-checker. The monadic approach uses an abstract datatype which enforces single-threaded use of world resources, but which also requires the programmer to explicitly make use of this datatype and its operations. In effect, the monad acts as a wrapper around the potentially dangerous operations.

Question: Does the explicit monadic wrapper and its laws make the monadic I/O program harder to reason about when compared to a similar uniquely typed program ?

1.1 Methodology

The key aim of this work is to establish the effect the choice of paradigm has on the ease of reasoning. In particular we wish to avoid differences introduced by idiosyncrasies associated with real world instances of these paradigms. The paradigms under study, and well-known real world instances are:

Imperative: explicit side-effects with sequencing and assignment (C [KR88]).

Uniquely-Typed: referentially transparent with side-effects guaranteed single-threaded by a type-system dealing with uniqueness (Clean [PvE98]).

Monadic: referentially transparent with side-effects guaranteed single-threaded by embedding them within monads (Haskell [PH+99]).

The C programming language and Unix operating system have led to a fairly standardised set of I/O system calls, most of which are found with similar names, signature and behaviour in the Clean I/O system. However, the Haskell I/O system has some differences in both names and signatures with consequent differences in behaviour. The Clean I/O system also has system calls which have no counterpart in C, but which facilitate the use of the uniqueness type system. In order to factor out these differences, we needed to work with modified versions of each language to make the I/O system appear as uniform as possible.

The case study involved the following steps:

1. Choose the task to be performed by the program
2. Write and run real programs as a check
3. Develop a standardised I/O model
4. Rework the programming languages to make them uniform
5. Re-write the programs to conform to the reworked languages and I/O model

6. Develop formal denotational semantics for the languages

7. State property to be proved and attempt proofs.

8. Develop non-denotational semantics for the languages

9. State and prove properties

Task Choice

We wanted a small case-study to start, in order that we did not get swamped in too much messy detail. The key requirement was that the program performed some I/O and that the desired property would refer both to the external world and to some property of the data involved. We chose a simple task which involved opening a file with a fixed filename ("a"), reading an integer from it, closing it, re-opening it, and writing the square of that integer back. The property to be checked was: given the existence of such a file with at least one integer, that that file would end up with only one integer value, being the square of the original value.

Real Programs

Real programs were written in C, Clean and and Haskell, compiled and run. This step was particularly important for the Clean program as a key issue (discussed in more detail later) is that we can rely on the uniqueness typing to ensure single-threaded use of the I/O functions. So we needed to use a real program to be certain that we did have the required uniqueness typing. Similarly, with Haskell, we ensured that the IO monad usage was correctly typed. The Haskell program also made use of an auxiliary function definition so that it would have the same overall structure as the other two programs.

I/O Model

As a common background to the three cases, we developed a uniform model of file I/O to be used in all proofs. This model captures the notion of a "file-system" and the behaviour of the required file manipulation functions.

Reworked Languages

The programming languages were re-designed to minimise the differences between them, apart form the paradigm difference under study. In particular, the C-like language was assumed to have the same expression syntax and value space as those available in the Clean- and and Haskell-like languages. The re-worked languages were kept small, only covering the features needed for the case-study. The re-working also ensured that the overall structure of each program would not be changed, in order to avoid the risk of introducing type errors.

Reworked Programs

The programs were then translated into their re-worked languages, which in the main involved the renaming of the file operation functions, and some re-ordering of arguments.

Denotational Semantics

Initially it was decided to develop a denotational semantics [Sch88] for the three re-worked languages, largely because we were used to this approach and felt happiest about getting the semantic model correct. Denotational semantics were produced for the C-like and Clean-like languages, but not for the Haskell-like language (this would be almost identical to that for the Clean-like language, in any case).

Denotational Proofs

Proofs based on the denotational semantics were then attempted for the C-like and Clean-like programs. However, these proofs rapidly became unwieldy, largely due to the environment information being handed around. After a short struggle it was decided to abandon these proofs in favour of more tractable techniques. The partial proofs are shown in section 7. for reference. However, some of the domains developed for the denotational semantics did prove very useful in the later semantic models, so this effort was not entirely wasted.

non-Denotational Semantics

It was decided to develop semantics that would support proofs at the program text level, with use being made of so-called "laws of programming" or source-language transformation rules. For the C-like language we explored the use of Hoare triples [HJ98] and weakest precondition [Mor94], and finally settled on the Hoare triples as a proof methodology.

For the functional languages, we simply built a collection of re-write rules neccessary to perform the proofs, rather than giving a complete set.

In all three cases, we integrated the I/O model with the semantics being developed. Interestingly, both the C-like and Haskell-like semantics required additional machinery to be introduced.

non-Denotational Proofs

For each paradigm, we stated in the property to be proved in the appropriate manner. We then proceeded to do the proofs, ensuring that the proofs were complete that all necessary lemmas were handled, and paying particular attention to the pre-conditions of the operations.

2 The I/O Model

We develop an IO model to suit the case-study.

2.1 The World and the File-System

We posit a 'world' where everything of interest happens:

$$
\begin{aligned}
\mathcal{W} \in \mathit{World} &\;\widehat{=}\; \mathit{FS} \times \mathit{Events} \times \mathit{WWW} \times \cdots \\
\mathit{Events} &\;\widehat{=}\; \ldots \\
\mathit{WWW} &\;\widehat{=}\; \ldots
\end{aligned}
$$

The world contains interesting sub-systems such as the file-system of the local machine, GUI event queues, internet access, up to and including the World Wide Web. We shall only be interested in the file system component (FS).

The file system maps filenames to files:

$$
\begin{aligned}
\Phi \in \mathit{FS} &\;\widehat{=}\; \mathit{FName} \xrightarrow{m} \mathit{File} \\
n \in \mathit{FName} &\;\widehat{=}\; \mathbb{A}^\star \\
f \in \mathit{File} &\;\widehat{=}\; \mathit{FState} \times \mathit{FData}
\end{aligned}
$$

The file includes the file's data contents, as well as the file state. For present purposes, we shall simply view the file data as being sequences of integers

$$
\delta \in \mathit{FData} \;\widehat{=}\; \mathbb{Z}^\star
$$

We shall adopt the principle for this exercise, that a file can be opened many times for reading, but only once for writing. Also it cannot simultaneously be opened for both reading or writing. The file state ensures sensible patterns of access, by maintaining information about files which are opened for reading or writing, ensuring that only one writer exists at any point, and keeping track of the number of readers.

$$
\begin{aligned}
\Sigma \in \mathit{FState} &\;\widehat{=}\; \textsc{Closed} \\
&\;\;|\;\; \textsc{Write} \\
&\;\;|\;\; \textsc{Read}\ \mathbb{N}
\end{aligned}
$$

Once a file is opened, we use a file status block, which tracks the state of the open file.

$$
\begin{aligned}
f \in \mathit{FStatus} &\;\widehat{=}\; \textsc{HWrite}\ \mathit{FName}\ \mathit{FData} \\
&\;\;|\;\; \textsc{HRead}\ \mathit{FName}\ \mathit{FData}\ \mathit{FData}
\end{aligned}
$$

We split read data into two portions, that already read, and that remaining to be read, in order to simulate the motion of a read-head. The read status:

$$
\textsc{HRead}\ n\ \delta_r\ \delta_w
$$

denotes a file where portion δ_r has been read (r), while section δ_w is still waiting (w). We put the filename into the file status block, to facilitate the process of file closing (it is a sort of back-link into the filesystem).

We need to define a file mode in order to be able to specify what kind of file status is required:

$$
m \in \mathit{FMode} \;\widehat{=}\; \{\textsc{FRead}, \textsc{FWrite}\}
$$

2.2 The Operations

We now give definitions of all the operations. We shall adopt a standard framework in order that the semantics definitions can be kept uniform. In general an I/O operation takes some control or input data as a first argument, the world (or a relevant portion) as a second argument, and returns a tuple consisting of a result value and the modified world:

$$\text{InputOutputOp} : \mathit{Val} \rightarrow \mathit{World} \rightarrow \mathit{Val} \times \mathit{World}$$

Here we assume *Val* includes all possible program values. If there is no *Val* input or result, we omit that component.

For file operations, we restrict ourselves to the file-system part of the world:

$$\text{FileOp} : \mathit{Val} \rightarrow \mathit{FS} \rightarrow \mathit{Val} \times \mathit{FS}$$

2.2.1 The fopen Operation

The fopen operation takes a filename, file mode and file-system argument, and returns a file-system and file status block:

$$\text{fopen} \quad : \quad \mathit{FName} \times \mathit{FMode} \rightarrow \mathit{FS} \rightarrow \mathit{FStatus} \times \mathit{FS}$$

The operation is defined if

- the mode is Write and the file does not exist, or
- the mode is Write, the file exists, but is not already open, or
- the mode is Read, the file exists, and is either closed or open for reading.

$$\begin{array}{rcl} \mathit{pre}\text{-fopen} & : & \mathit{FName} \times \mathit{FMode} \rightarrow \mathit{FS} \rightarrow \mathbb{B} \\ \mathit{pre}\text{-fopen}(n, \text{FW}\textsc{rite})\Phi & \mathrel{\widehat=} & n \in \mathtt{dom}\ \Phi \ \rightarrow\ \pi_1\Phi(n) = \textsc{Closed}\ ,\ \textsc{True} \\ \mathit{pre}\text{-fopen}(n, \text{FR}\textsc{ead})\Phi & \mathrel{\widehat=} & n \in \mathtt{dom}\ \Phi \ \rightarrow\ \pi_1\Phi(n) \neq \textsc{Write}\ ,\ \textsc{False} \end{array}$$

The behaviour of the operation is as follows:

- If the mode is FWrite then, a file is created if not already present, it's contents are erased, state set to Write and a file status block is built and returned.
- If the mode is FRead then the file status is set to Read if not already so, and its reader count is adjusted. A file status block is then returned with nothing read, and everything left to read.

$$
\begin{array}{rcl}
\mathrm{fopen}(n, \mathrm{FW}\textsc{rite})\Phi & \widehat{=} & (h, \Phi \dagger \{n \mapsto f\}) \\
& & \textbf{where} \\
& & \quad h = \mathrm{HW}\textsc{rite}\ n\ \Lambda \\
& & \quad f = (\textsc{Write}, \Lambda) \\
\mathrm{fopen}(n, \mathrm{FR}\textsc{ead})\Phi & \widehat{=} & (h, \Phi \dagger \{n \mapsto f\}) \\
& & \textbf{where} \\
& & \quad h = \mathrm{HR}\textsc{ead}\ n\ \Lambda\ \delta \\
& & \quad f = (\textsc{Read}\ r, \delta) \\
& & \quad r = \pi_1(\Phi(n)) = \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(n))) + 1 \\
& & \quad \delta = \pi_2(\Phi(n))
\end{array}
$$

2.2.2 The fclose Operation

The fclose operation takes a file status block, and file-system argument, and returns a file-system:

$$
\mathrm{fclose} \quad : \quad \mathit{FStatus} \rightarrow \mathit{FS} \rightarrow \mathit{FS}
$$

The operation is defined if

- the file is present in the filesystem, and
- the filesystem version is in the same mode

$$
\begin{array}{rcl}
\mathit{pre}\text{-fclose} & : & \mathit{FStatus} \rightarrow \mathit{FS} \rightarrow \mathbb{B} \\
\mathit{pre}\text{-fclose}\ (\mathrm{HW}\textsc{rite}\ n\ \_)\ \Phi & \widehat{=} & n \in \mathtt{dom}\ \Phi \\
& \wedge & \pi_1(\Phi(n)) = \textsc{Write} \\
\mathit{pre}\text{-fclose}\ (\mathrm{HR}\textsc{ead}\ n\ \_\ \_)\ \Phi & \widehat{=} & n \in \mathtt{dom}\ \Phi \\
& \wedge & \pi_1(\Phi(n)) = \textsc{Read}\ \_
\end{array}
$$

Note: no file should exist that does not satisfy this pre-condition, as long as our system has only *one* filesystem and all files are generated by fopen and only modifed by freadi or fwritei. We add the condition to stress this important property.

The behaviour of the operation is as follows:

- If the file was open for writing, then the file-data becomes that of the file-status block, and the file state becomes closed.
- If the file was open for reading, the status block is discarded and the count in the file state is decremented. If the count reaches zero, then the state becomes closed.

$$
\begin{aligned}
\text{fclose}\ (\text{HW}\textsc{rite}\ n\ \delta)\ \Phi &\ \widehat{=}\ \ \Phi \dagger \{n \mapsto (\textsc{Closed}, \delta)\} \\
\text{fclose}\ (\text{HR}\textsc{ead}\ n\ \_)\ \Phi &\ \widehat{=}\ \ \Phi \dagger \{n \mapsto (s, \delta)\} \\
&\qquad \textbf{where} \\
&\qquad\quad ((\textsc{Read}\ r), \delta) = \Phi(n) \\
&\qquad\quad s \mathrel{\widehat{=}} r = 1\ \rightarrow\ \textsc{Closed}\ ,\ \textsc{Read}\ (r-1)
\end{aligned}
$$

2.2.3 The fwritei Operation

The fwritei operation takes a file status block, and integer arguments and returns a modified file-status block:

$$\text{fwritei} \quad : \quad \mathbb{Z} \rightarrow \mathit{FStatus} \rightarrow \mathit{FStatus}$$

The operation is defined if

- the status block mode is HWrite.

$$
\begin{aligned}
\mathit{pre}\text{-fwritei} &\quad : \quad \mathbb{Z} \rightarrow \mathit{FStatus} \rightarrow \mathbb{B} \\
\mathit{pre}\text{-fwritei}[i](\text{HW}\textsc{rite}\ \_) &\quad \widehat{=} \quad \textsc{True} \\
\mathit{pre}\text{-fwritei}[i](\text{HR}\textsc{ead}\ \_) &\quad \widehat{=} \quad \textsc{False}
\end{aligned}
$$

The behaviour of the operation is as follows:

- The integer is appended to the file data sequence

$$\text{fwritei}[i](\text{HW}\textsc{rite}\ n\ \delta) \quad \widehat{=} \quad \text{HW}\textsc{rite}\ n\ (\delta \frown \langle i \rangle)$$

2.2.4 The freadi Operation

The freadi operation takes a file-status block, as input, and returns a modified file-status block and integer as result

$$\text{freadi} \quad : \quad \mathit{FStatus} \rightarrow \mathbb{Z} \times \mathit{FStatus}$$

The operation is defined if

- the status block is in FRead mode and,
- there is at least one more integer to be read.

$$
\begin{aligned}
\mathit{pre}\text{-freadi} &\quad : \quad \mathit{FStatus} \rightarrow \mathbb{B} \\
\mathit{pre}\text{-freadi}(\text{HW}\textsc{rite}\ \_) &\quad \widehat{=} \quad \textsc{False} \\
\mathit{pre}\text{-freadi}(\text{HR}\textsc{ead}\ \_\ \_\ \delta) &\quad \widehat{=} \quad \delta \neq \Lambda
\end{aligned}
$$

The behaviour of the operation is as follows:

- The head of the list of items still to be read is transferred to the tail of the items already read list, and
- it is also returned as the outcome of the read.

$$\text{freadi}(\text{HR}\textsc{ead}\ n\ \delta_r\ (i : \delta_w)) \quad \widehat{=} \quad (i, (\text{HR}\textsc{ead}\ n\ (\delta_r \frown \langle i \rangle)\ \delta_w))$$

2.3 I/O Model Signature Summary

$$
\begin{array}{rcl}
\mathcal{W} \in \mathit{World} & \widehat{=} & \mathit{FS} \times \mathit{Events} \times \mathit{WWW} \times \cdots \\
\mathit{Events} & \widehat{=} & \ldots \\
\mathit{WWW} & \widehat{=} & \ldots \\
\Phi \in \mathit{FS} & \widehat{=} & \mathit{FName} \xrightarrow{m} \mathit{File} \\
n \in \mathit{FName} & \widehat{=} & \mathbb{A}^\star \\
f \in \mathit{File} & \widehat{=} & \mathit{FState} \times \mathit{FData} \\
\delta \in \mathit{FData} & \widehat{=} & \mathbb{Z}^\star \\
\Sigma \in \mathit{FState} & \widehat{=} & \textsc{Closed} \\
 & | & \textsc{Write} \\
 & | & \textsc{Read}\ \mathbb{N} \\
f \in \mathit{FStatus} & \widehat{=} & \text{H}\textsc{Write}\ \mathit{FName}\ \mathit{FData} \\
 & | & \text{H}\textsc{Read}\ \mathit{FName}\ \mathit{FData}\ \mathit{FData} \\
m \in \mathit{FMode} & \widehat{=} & \{\text{F}\textsc{Read}, \text{F}\textsc{Write}\} \\
\text{fopen} & : & \mathit{FName} \times \mathit{FMode} \to \mathit{FS} \to \mathit{FStatus} \times \mathit{FS} \\
\text{fclose} & : & \mathit{FStatus} \to \mathit{FS} \to \mathit{FS} \\
\text{fwritei} & : & \mathbb{Z} \to \mathit{FStatus} \to \mathit{FStatus} \\
\text{freadi} & : & \mathit{FStatus} \to \mathbb{Z} \times \mathit{FStatus}
\end{array}
$$

2.4 Connecting I/O Model to Abstracted Programs

We give the signatures of each I/O function, as they appear in the model, and each programming language

Model	fopen : $\mathit{FName} \times \mathit{FMode} \to \mathit{FS} \to \mathit{FStatus} \times \mathit{FS}$
C	: $\mathit{FName} \times \mathit{FMode} \to \mathit{FStatus}$
Clean	: $\mathit{FName} \to \mathit{FMode} \to \mathit{FS} \to (\mathit{FStatus} \times \mathit{FS})$
Haskell	: $\mathit{FName} \to \mathit{FMode} \to \mathit{IO}\ \mathit{FStatus}$
Model	fclose : $\mathit{FStatus} \to \mathit{FS} \to \mathit{FS}$
C	: $\mathit{FStatus} \to ()$
Clean	: $\mathit{FStatus} \to \mathit{FS} \to \mathit{FS}$
Haskell	: $\mathit{FStatus} \to \mathit{IO}\ ()$
Model	fwritei : $\mathbb{Z} \to \mathit{FStatus} \to \mathit{FStatus}$
C	: $\mathit{FStatus} \times \mathbb{Z} \to ()$
Clean	: $\mathit{FStatus} \to \mathbb{Z} \to \mathit{FStatus}$
Haskell	: $\mathit{FStatus} \to \mathbb{Z} \to \mathit{IO}\ ()$
Model	freadi : $\mathit{FStatus} \to \mathbb{Z} \times \mathit{FStatus}$
C	: $\mathit{FStatus} \to \mathbb{Z}$
Clean	: $\mathit{FStatus} \to (\mathit{FStatus} \times \mathbb{Z})$
Haskell	: $\mathit{FStatus} \to \mathit{IO}\ \mathbb{Z}$

3 Abstract Syntaxes

We present abstract syntax forms for all three programming languages, to facilitate the generation of semantics.

3.1 Common Syntax

Some parts of syntax like constant, variables and certain forms of expression are common to all three languages, and are defined here.

3.1.1 Common Expressions

We start with constants and variables as given lexical entities:

$$\begin{array}{rcl} \mathit{Const} & ::= & \{*, \\ & & \ \ \textsc{fopen}, \textsc{fclose}, \textsc{fwritei}, \textsc{freadi}, \\ & & \ \ \mathrm{F}\textsc{Read}, \mathrm{F}\textsc{Write}, \\ & & \ldots\} \\ \mathit{Var} & ::= & \text{typical identifier lexemes} \end{array}$$

A basic expression (*BExpr*) is a constant, variable, tuple of expressions or the application of one expression to another:

$$\begin{array}{rcl} \mathit{BExpr} & ::= & \textsc{Const}\ \mathit{Const} \\ & | & \textsc{Var}\ \mathit{Var} \\ & | & \textsc{Tuple}\ \mathit{BExpr}^{+} \\ & | & \textsc{App}\ \mathit{BExpr}\ \mathit{BExpr} \end{array}$$

3.1.2 Functional Language Expressions

For functional languages, we introduce patterns, and extend the expression syntax.

Patterns (*Patn*) are basic expressions restricted to constant, variables and tuples:

$$\begin{array}{rcl} \mathit{Patn} & ::= & \textsc{Const}\ \mathit{Const} \\ & | & \textsc{Var}\ \mathit{Var} \\ & | & \textsc{Tuple}\ \mathit{Patn}^{+} \end{array}$$

We obtain functional expressions (*FExpr*) by adding in lambda abstractions and let-expressions to basic expressions:

$$\begin{array}{rcl} \mathit{FExpr} & ::= & \textsc{Const}\ \mathit{Const} \\ & | & \textsc{Var}\ \mathit{Var} \\ & | & \textsc{Tuple}\ \mathit{FExpr}^{+} \\ & | & \textsc{App}\ \mathit{FExpr}\ \mathit{FExpr} \\ & | & \textsc{Abs}\ \mathit{Var}\ \mathit{FExpr} \\ & | & \textsc{Let}\ \mathit{Patn}\ \mathit{FExpr}\ \mathit{FExpr} \end{array}$$

3.2 C Abstract Syntax

3.2.1 C Statements

A C statement (*CStmt*)is either an assignment, or a procedure call:

$$\begin{array}{lcl} \mathit{CStmt} & ::= & \textsc{Asg}\ \mathit{Var}\ \mathit{BExpr} \\ & | & \textsc{Call}\ \mathit{BExpr}\ \mathit{BExpr} \end{array}$$

3.2.2 C Programs

A C program (*CProg*) is a sequence of C statements:

$$\mathit{CProg} \quad ::= \quad \mathit{CStmt}^{\star}$$

3.3 Clean Abstract Syntax

3.3.1 Clean Expressions

Clean has expressions (*ClExpr*) extended with the "hash-let" notation

$$\begin{array}{lcl} \mathit{ClExpr} & ::= & \textsc{Const}\ \mathit{Const} \\ & | & \textsc{Var}\ \mathit{Var} \\ & | & \textsc{Tuple}\ \mathit{ClExpr}^{+} \\ & | & \textsc{App}\ \mathit{ClExpr}\ \mathit{ClExpr}^{+} \\ & | & \textsc{Abs}\ \mathit{Var}\ \mathit{ClExpr} \\ & | & \textsc{Let}\ \mathit{Patn}\ \mathit{ClExpr}\ \mathit{ClExpr} \\ & | & \textsc{Hash}\ \mathit{ClHElem}^{\star}\ \mathit{ClExpr} \end{array}$$

3.3.2 Clean Hash Elements

The Clean "hash-let" construct is a list of hash elements (*ClHElem*), each being a binding of a pattern to an expression:

$$\mathit{ClHElem} \quad ::= \quad \mathit{Patn}\ \mathit{ClExpr}$$

3.3.3 Clean Programs

A Clean program (*ClProg*) is basically an abstraction:

$$\mathit{ClProg} \quad ::= \quad \mathit{Var} \times \mathit{ClExpr}$$

3.4 Haskell Abstract Syntax

3.4.1 Haskell Expressions

Haskell has expressions (*HExpr*) extended with monadic "do" notation

$$
\begin{array}{rcl}
\mathit{HExpr} & ::= & \textsc{Const}\ \mathit{Const} \\
 & | & \textsc{Var}\ \mathit{Var} \\
 & | & \textsc{Tuple}\ \mathit{HExpr}^{+} \\
 & | & \textsc{App}\ \mathit{HExpr}\ \mathit{HExpr} \\
 & | & \textsc{Abs}\ \mathit{Var}\ \mathit{HExpr} \\
 & | & \textsc{Let}\ \mathit{Patn}\ \mathit{HExpr}\ \mathit{HExpr} \\
 & | & \textsc{Do}\ \mathit{MStmt}^{\star}
\end{array}
$$

3.4.2 Haskell Monadic Statements

The Haskell "do" syntax has components which look vaguely like imperative statements. A Monadic Statement (*MStmt*) is either a monadic assignment (binding) or monad function call expression (return?):

$$
\begin{array}{rcl}
\mathit{MStmt} & ::= & \textsc{Bind}\ \mathit{Var}\ \mathit{HExpr} \\
 & | & \textsc{Retn}\ \mathit{HExpr}
\end{array}
$$

3.4.3 Haskell Programs

A Haskell Program (*HProg*) is basically an expression:

$$\mathit{HProg} \quad ::= \quad \mathit{HExpr}$$

Usually it is expected to be a "do" expression.

4 Real Programs

We present the real programs that actually ran here.

4.1 The real C program

```
#include <stdio.h>

int main()
{
  FILE *f;
  int x;

  f = fopen("a","r");
  if(!f){
    perror("prog1");
```

```
    return 1;
  }
  fscanf(f,"%d",&x);
  fclose(f);

  f = fopen("a","w");
  if(!f){
    perror("prog1");
    return 1;
  }
  fprintf(f,"%d",x*x);
  fclose(f);
  return 0;
}
```

4.2 The real Clean program

```
module prog1
import StdEnv

Start w # (_, f, w) = fopen "a" FReadText w
        # (_,x,f)   = freadi f
        # (_,w)     = fclose f w
        # (_,f,w)   = fopen "a" FWriteText w
        # f         = fwritei (x*x) f
        # (_,w)     = fclose f w
        | otherwise = w
```

4.3 The real Haskell program

```
import IO

main = do
         h <- openFile "a" ReadMode
         s <- hGetContents h
         x <- readIO s::IO Int
         hClose h
         h <- openFile "a" WriteMode
         hPutStr h (show (x*x))
         hClose h
```

5 Abstracted Programs

To simplify matters, and to ensure that we focus on differences inherent the basic reasoning models behind each language, rather than specific details of these particular languages, we have re-written the functions to have a uniform appearance, using IO functions with the same names and overall structure.

5.1 The IO abstraction

We present a table showing the abstracted IO operations and their equivalents in the programming languages:

Abstract	C	Clean	Haskell
fopen	fopen	fopen	openFile/hGetContents
freadi	fscanf	freadi	hGetContents/readIO
fclose	fclose	fclose	hClose
fwritei	fprintf	fwritei	hPutStr/show

Note: the Haskell function hGetContents is a form of lazy read, so it could be associated with either open or reading the integer. We need a decision on this.

Decision: we shall not use hGetContents — instead we define a Haskell version of `freadi`, using `getChar` and similar.

5.2 Concrete Programs using IO Abstraction

5.2.1 The abstracted C program

```
main()
{
  f = fopen("a",FRead);
  x = freadi(f);
  fclose(f);
  f = fopen("a",FWrite);
  fwritei(f,x*x);
  fclose(f);
}
```

We rename functions as appropriate, and discard variable declarations and the error checking for now.

5.2.2 The abstracted Clean program

```
main w # (f,w) = fopen "a" FRead w
       # (x,f) = freadi f
       # w     = fclose f w
       # (f,w) = fopen "a" FWrite w
       # f     = fwritei (x*x) f
       # w     = fclose f w
       = w
```

We remove return condition values, as well as discarding last conditional.

5.2.3 The abstracted Haskell program

We use slightly different names here, mainly because it will make it easier to distinguish the Haskell functions from the underlying I/O model functions.

```
main = do
         h <- openFile "a" ReadMode
         x <- hreadi h
         hclose h
         h <- openFile "a" WriteMode
         hwritei h (x*x)
         hclose h

hreadi :: Handle -> IO Int
hreadi h = do
            s <- hGetWord h
            readIO s::IO Int

hGetWord h = do
               c <- hGetChar h
               if (isSpace c)
                then
                  return ""
                else
                  do
                    cs <- hGetWord h
                    return (c:cs)
```

Note the use of an auxiliary definition, `hreadi`, which gives the semantics required by the IO model. We will use this definition from here on, and will assume that `hreadi` has the obvious semantics.

5.3 Abstract Syntax Forms

We then transform the above examples into fully abstract syntax forms. These will be the basis for denotational style proofs.

5.3.1 Abstract Syntax for C Program

```
ASG f
    APP (CONST FOPEN)
        TUPLE CONST "a"
              CONST FREAD
ASG x
    APP (CONST FREADI)
        VAR f
CALL (CONST FCLOSE) (VAR f)
ASG f
    APP (CONST FOPEN)
        TUPLE CONST "a"
```

```
                  Const FWrite
Call
      App (Const fwritei)
           Tuple Var f
                  App (Const *)
                       Tuple Var x
                              Var x
Call (Const fclose) (Var f)
```

5.3.2 Abstract Syntax for Clean Program

```
w
Hash (Tuple (Var f)
              Var w  )
        App (Const fopen)
             Const "a"
             Const FRead
             Var w
      Tuple (Var x)
             Var f
        App (Const freadi)
             Var f
      Var w
        App (Const fclose)
             Var f
             Var w
      Tuple (Var f)
             Var w
        App (Const fopen)
             Const "a"
             Const FWrite
             Var w
      Var f
        App (Const fwritei)
             Var f
             App (Const *)
                  Tuple Var x
                         Var x
      Var w
        App (Const fclose)
             Var f
             Var w
Var w
```

5.3.3 Abstract Syntax for Haskell Program

```
Do (Bind f
          App (Const fopen)
               Const “a”
               Const FRead )
    Bind x
         App (Const freadi)
              Var f
    Retn App (Const fclose)
              Var f
    Bind f
         App (Const fopen)
              Const “a”
              Const FWrite
    Retn App (Const fwritei)
              Var f
              App (Const *)
                   Tuple Var x
                         Var x
    Retn App (Const fclose)
              Var f
```

6 Denotational Semantics

We start by giving a denotation semantics to each language.

We assume as semantic domains those defined in the IO Model, as well as additional value components.

6.1 Common Semantic Domains

6.1.1 Value Semantic Domain

We first define the I/O semantic domain (IO) include all the components of the I/O domain model, up to and including the world!

$$\begin{array}{rcl} IO & \widehat{=} & World \\ & + & FS \\ & + & FStatus \\ & + & \ldots \end{array}$$

We define the value semantic domain (Val) to be the disjoint union of integer, I/O values, handles over a range of IO types, tuples of values, and (continuous, computable) functions over values:

$$\begin{array}{rcl} Val & \widehat{=} & \mathbb{Z} \\ & + & IO \\ & + & \sum Handle\ T \\ & + & Val^{\star} \\ & + & [Val \rightarrow Val] \end{array}$$

We assume a function (C) that maps all lexical constants to their values:

$$\begin{array}{rcl} \mathsf{C} & : & Const \rightarrow Val \\ \mathsf{C}[\![0]\!] & \widehat{=} & 0 \\ \mathsf{C}[\![*]\!] & \widehat{=} & \lambda(n_1, n_2) \cdot n_1 * n_2 \\ \mathsf{C}[\![\text{FOPEN}]\!] & \widehat{=} & \texttt{fopen} \\ & & \text{etc} \ldots \end{array}$$

Depending on the paradigm, we may override the default values here with modified versions.

6.1.2 Environments

We shall define a local variable environment (*LEnv*) as a (finite) mapping from variables to values:

$$\ell \in LEnv \quad \widehat{=} \quad Var \xrightarrow{m} Val$$

A program variable environment (*PEnv*) is a stack of local variable environments, represented by a non-null sequence.

$$\rho \in PEnv \quad \widehat{=} \quad LEnv^{+}$$

The stack form is used to handle nested scopes.

We extend map lookup to sequences of maps by looking up the maps in sequence until a match is found, or all maps are exhausted. We extend map override to map sequences, by stating that it acts on the first map.

6.1.3 Handles/References

For some of the paradigms, we will need to hand around handles or references to information structures to allow side-effects to occur. We shall view a handle as a natural number, and map this to the appropriate structures. Handles and instances of the relevant structure are then allocated and freed as required. We shall parameterise both handles and the handle mapping by the type (T) of the information structure:

$$\begin{array}{rcl} h \in Handle\ T & \widehat{=} & \mathbb{N} \\ \varrho \in HMap\ T & \widehat{=} & \mathbb{N} \xrightarrow{m} T \end{array}$$

Given a new structure, and a handle map, we can allocate a new entry in the structure and return a handle. The handle must not be one currently in use. We adopt an easy way to guarantee this:

$$\begin{aligned} \text{hAlloc} &: T \to \mathit{HMap}\ T \to \mathit{Handle}\ T \times \mathit{HMap}\ T \\ \text{hAlloc}[t]\varrho &\mathrel{\widehat{=}} (h, \varrho \sqcup \{h \mapsto t\}) \\ &\quad \textbf{where } h = \max(\texttt{dom}\ \varrho) + 1 \end{aligned}$$

We can also free structures, although this is not really necessary for most semantic purposes:

$$\begin{aligned} \text{hFree} &: \mathit{Handle}\ T \to \mathit{HMap}\ T \to \mathit{HMap}\ T \\ \text{hFree}[h]\varrho &\mathrel{\widehat{=}} \lhd[h]\varrho \end{aligned}$$

6.1.4 Overall Environment

The overall environment (Env_X) for a paradigm X is a tuple containing at least the world and a program variable environment, as well as some other components, such as handle-maps, specific to the given paradigm:

$$\varepsilon \in \mathit{Env}_X \mathrel{\widehat{=}} \mathit{World} \times \mathit{PEnv} \times \cdots$$

The paradigms are C (C), K (Clean) and H (Haskell).

6.1.5 Denotation Functions

In all cases, the denotation of a program ($\mathsf{P}[\![\texttt{prog}]\!]$) will be a function from *World* to *World*:

$$\begin{aligned} \mathsf{P}[\![\texttt{prog}]\!] &: \mathit{World} \to \mathit{World} \\ \mathsf{P}[\![\texttt{prog}]\!]\ \mathcal{W} &\mathrel{\widehat{=}} \pi_1(\mathsf{Top}[\![top - stmt]\!]\varepsilon_0) \end{aligned}$$

Such a function will build an initial environment, call the denotation function for the top-level structure, and strip out the final world value from the overall result.

6.1.6 Note on Type-Correctness

In the sequel, it is assumed that all programs are type-correct, so that all functions are applied to the correct argument type. A lot of the functions defined here are total on type-correct programs, but partial on all possible programs.

6.2 C Denotational Semantics

6.2.1 C Program State

The state of a C program will consist of the world, and environment, and a file status handle map:

$$\mathit{Env}_C \mathrel{\widehat{=}} \mathit{World} \times \mathit{PEnv} \times \mathit{HMap}\ \mathit{FStatus}$$

6.2.2 C Program Denotations

$$
\begin{array}{rcl}
\mathsf{P}_C & : & \mathit{CProg} \to \mathit{World} \to \mathit{World} \\
\mathsf{P}_C[\![\sigma]\!]\ \mathcal{W} & \widehat{=} & \pi_1(\mathsf{SS}_C[\![\sigma]\!](\mathcal{W}, \langle\theta\rangle, \theta))
\end{array}
$$

6.2.3 C Statement Denotations

$$
\begin{array}{rcl}
\mathsf{SS}_C & : & \mathit{CStmt}^\star \to \mathit{Env}_C \to \mathit{Env}_C \\
\mathsf{SS}_C[\![\Lambda]\!]\varepsilon & \widehat{=} & \varepsilon \\
\mathsf{SS}_C[\![s : \sigma]\!]\varepsilon & \widehat{=} & (\mathsf{SS}_C[\![\sigma]\!] \circ \mathsf{S}_C[\![s]\!])\varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{S}_C & : & \mathit{CStmt} \to \mathit{Env}_C \to \mathit{Env}_C \\
\mathsf{S}_C[\![\textsc{Asg}\ v\ e]\!](\mathcal{W}, \rho, \varrho) & \widehat{=} & \textbf{let}\ (r, (\mathcal{W}', \rho', \varrho')) = \mathsf{E}_C[\![e]\!](\mathcal{W}, \rho, \varrho) \\
& & \textbf{in}\ (\mathcal{W}', \rho' \dagger \{v \mapsto r\}, \varrho') \\
\mathsf{S}_C[\![\textsc{Call}\ p\ a]\!]\varepsilon & \widehat{=} & \textbf{let}\ (a', \varepsilon') = \mathsf{E}_C[\![a]\!]\varepsilon \\
& & \textbf{in}\ \pi_2(\mathsf{App}_C[\![p]\!](a', \varepsilon')
\end{array}
$$

A procedure call is a function call where the result is discarded.

6.2.4 C Expression Denotations

$$
\begin{array}{rcl}
\mathsf{E}_C & : & \mathit{BExpr} \to \mathit{Env}_C \to \mathit{Val} \times \mathit{Env}_C \\
\mathsf{E}_C[\![\textsc{Const}\ c]\!]\varepsilon & \widehat{=} & (\mathsf{C}[\![c]\!], \varepsilon) \\
\mathsf{E}_C[\![\textsc{Var}\ v]\!](\mathcal{W}, \rho, \varrho) & \widehat{=} & (\rho(v), (\mathcal{W}, \rho, \varrho)) \\
\mathsf{E}_C[\![\textsc{Tuple}\ \sigma]\!]\varepsilon & \widehat{=} & \mathsf{Tuple}_C[\![\sigma]\!](\Lambda, \varepsilon) \\
\mathsf{E}_C[\![\textsc{App}\ f\ a]\!]\varepsilon & \widehat{=} & \textbf{let}\ (a', \varepsilon') = \mathsf{E}_C[\![a]\!]\varepsilon \\
& & \textbf{in}\ \mathsf{App}_C[\![f]\!](a', \varepsilon')
\end{array}
$$

Note that function application is strict — arguments are evaluated before the function call is made.

$$
\begin{array}{rcl}
\mathsf{Tuple}_C & : & \mathit{BExpr}^\star \to \mathit{Val}^\star \times \mathit{Env}_C \to \mathit{Val} \times \mathit{Env}_C \\
\mathsf{Tuple}_C[\![\Lambda]\!](\nu, \varepsilon) & \widehat{=} & (\texttt{rev}\ \nu, \varepsilon) \\
\mathsf{Tuple}_C[\![e : \sigma]\!](\nu, \varepsilon) & \widehat{=} & \textbf{let}\ (v, \varepsilon') = \mathsf{E}_C[\![e]\!]\varepsilon \\
& & \textbf{in}\ \mathsf{Tuple}_C[\![\sigma]\!](v : \nu, \varepsilon')
\end{array}
$$

6.2.5 C Builtin I/O denotations

For applications, we currently assume that the function expression is a (builtin) constant, which we handle on a case-by-case basis. We shall denote the world by (Φ, w), highlighting the file system component, and using w to denote the rest.

$$
\begin{array}{rcl}
\mathsf{App}_C & : & \mathit{BExpr} \to \mathit{Val} \times \mathit{Env}_C \to \mathit{Val} \times \mathit{Env}_C \\
\multicolumn{3}{l}{\quad \mathsf{App}_C [\![\textsc{Const fopen}]\!] (\langle n, m \rangle, ((\Phi, w), \rho, \varrho))} \\
& \widehat{=} & \textbf{let } (f, \Phi') = \texttt{fopen}[n, m]\Phi \\
& & \textbf{in let } (h, \varrho') = \mathrm{hAlloc}[f]\varrho \\
& & \textbf{in } (h, ((\Phi', w), \rho, \varrho')) \\
\multicolumn{3}{l}{\quad \mathsf{App}_C [\![\textsc{Const fclose}]\!] (h, ((\Phi, w), \rho, \varrho))} \\
& \widehat{=} & \textbf{let } \Phi' = \texttt{fclose}[\varrho(h)]\Phi \\
& & \textbf{in let } \varrho' = \mathrm{hFree}[h]\varrho \\
& & \textbf{in } (!, ((\Phi', w), \rho, \varrho')) \\
\multicolumn{3}{l}{\quad \mathsf{App}_C [\![\textsc{Const fwritei}]\!] (\langle h, i \rangle, ((\Phi, w), \rho, \varrho))} \\
& \widehat{=} & \textbf{let } f' = \texttt{fwritei}[i](\varrho(h)) \\
& & \textbf{in let } \varrho' = \varrho \dagger \{h \mapsto f'\} \\
& & \textbf{in } (!, ((\Phi, w), \rho, \varrho')) \\
\multicolumn{3}{l}{\quad \mathsf{App}_C [\![\textsc{Const freadi}]\!] (h, ((\Phi, w), \rho, \varrho))} \\
& \widehat{=} & \textbf{let } (i, f') = \texttt{freadi}(\varrho(h)) \\
& & \textbf{in let } \varrho' = \varrho \dagger \{h \mapsto f'\} \\
& & \textbf{in } (i, ((\Phi, w), \rho, \varrho'))
\end{array}
$$

Note that we pass and return handles rather than file status blocks.

6.3 Clean Denotational Semantics

6.3.1 Clean Program State

The state of a Clean program will consist of a local environment only !

$$
\mathit{Env}_K \ \widehat{=}\ \mathit{LEnv}
$$

The world and it's components will be identified by program variables, and so will appear in the local environment.

6.3.2 Clean Program Denotations

A top level Clean program is the application of an abstraction to an argument, that denotes the world, and which returns the world:

$$
\begin{array}{rcl}
\mathsf{P}_K & : & \mathit{ClProg} \to \mathit{World} \to \mathit{World} \\
\mathsf{P}_K [\![(v, e)]\!]\, \mathcal{W} & \widehat{=} & \mathsf{E}_K \{v \mapsto \mathcal{W}\} [\![e]\!]
\end{array}
$$

6.3.3 Clean Expression Denotations

$$\begin{array}{rcl}
\mathsf{E}_K & : & \mathit{Env}_K \to \mathit{ClExpr} \to \mathit{Val} \\
\mathsf{E}_K \ell[\![\textsc{Const}\ c]\!] & \mathrel{\widehat{=}} & \mathsf{C}_K[\![c]\!] \\
\mathsf{E}_K \ell[\![\textsc{Var}\ v]\!] & \mathrel{\widehat{=}} & \ell(v) \\
\mathsf{E}_K \ell[\![\textsc{Tuple}\ \sigma]\!] & \mathrel{\widehat{=}} & (\mathsf{E}_K \ell)^\star \sigma \\
\mathsf{E}_K \ell[\![\textsc{App}\ f\ \alpha]\!] & \mathrel{\widehat{=}} & (\mathsf{E}_K \ell[\![f]\!])((\mathsf{E}_K \ell)^\star[\![\alpha]\!]) \\
\mathsf{E}_K \ell[\![\textsc{Abs}\ v\ b]\!] & \mathrel{\widehat{=}} & \lambda v' \cdot \mathsf{E}_K(\ell \dagger \{v \mapsto v'\})[\![b]\!] \\
\mathsf{E}_K \ell[\![\textsc{Let}\ p\ e\ b]\!] & \mathrel{\widehat{=}} & \mathsf{E}_K(\ell \dagger \mathrm{M}_K \ell[\![p, e]\!])[\![b]\!] \\
\mathsf{E}_K \ell[\![\textsc{Hash}\ \Lambda\ e']\!] & \mathrel{\widehat{=}} & \mathsf{E}\ell[\![e']\!] \\
\mathsf{E}_K \ell[\![\textsc{Hash}\ (p, e) : \varpi\ e']\!] & \mathrel{\widehat{=}} & \mathsf{E}_K(\ell \dagger \mathrm{M}_K[\![p]\!](\mathsf{E}_K \ell[\![e]\!]))[\![\textsc{Hash}\ \varpi\ e']\!]
\end{array}$$

6.3.4 Clean Pattern Match

A clean pattern match simply binds pattern variables to values, returning the binding as a local environment:

$$\begin{array}{rcl}
\mathsf{M}_K & : & \mathit{Patn} \to \mathit{Val} \to \mathit{LEnv} \\
\mathsf{M}_K[\![\textsc{Const}\ c]\!]\ _ & \mathrel{\widehat{=}} & \theta \\
\mathsf{M}_K[\![\textsc{Var}\ x]\!]\ v & \mathrel{\widehat{=}} & \{x \mapsto v\} \\
\mathsf{M}_K[\![\textsc{Tuple}\ \varpi)]\!]\ \sigma & \mathrel{\widehat{=}} & ({\sqcup}/ \circ (\mathsf{M}_K)^\star)(\mathtt{zip}(\varpi, \sigma))
\end{array}$$

We do not record if a match succeeds or fails at this point.

6.3.5 Clean Builtin Function Denotations

At present most Clean constants denote the functions directly. The only exception are FOPEN and FCLOSE, which need a wrapper to select out the filesystem component of the world:

$$\begin{array}{rcl}
\mathsf{C}_K & \mathrel{\widehat{=}} & \mathsf{C}\,\dagger \\
\textsc{fopen} & \mapsto & \lambda(n, m) \cdot \lambda(\Phi, w) \cdot (f, (Phi', w)) \\
 & & \quad \textbf{where } (f, \Phi') = \mathtt{fopen}[n, m]\Phi \\
\textsc{fclose} & \mapsto & \lambda(f) \cdot \lambda(\Phi, w) \cdot (Phi', w) \\
 & & \quad \textbf{where } \Phi' = \mathtt{fclose}[f]\Phi
\end{array}$$

6.4 Haskell Denotational Semantics

No denotational semantics were produced for the Haskell program as it was not considered likely that any additional insight over that provided by the Clean semantics would be gained.

7 Denotational Proofs

7.1 The Property

We want to show that, given the existence of a file called `a` before the program is run, containing at least one integer, that afterwards, the same file exists, containing one integer, being the square of the prior value.

We denote the state of the world before the program is run as:

$$\mathcal{W} = (\Phi, w) \quad \texttt{"a"} \in \texttt{dom}\ \Phi \wedge \Phi(\texttt{"a"}) = (\textsc{Closed}, J : \varsigma)$$

We can capture the initial condition by writing the starting state as

$$\mathcal{W} = (\Phi \sqcup \{\texttt{"a"} \mapsto (\textsc{Closed}, J : \varsigma)\}, w)$$

We denote the state after the program has terminated as:

$$\mathcal{W}' = (\Phi', w') = \mathsf{P}[\![\texttt{prog}]\!]\mathcal{W}$$

We want to show:

$$\Phi'(\texttt{"a"}) = (\textsc{Closed}, \langle J^2 \rangle)$$

We shall label parts of the abstract syntax to make it easier to refer to the components. We shall also use the concrete syntax as convenient abbreviations of the abstract

7.2 Proof for C Program

7.2.1 C Program Labelled Syntax

$$\texttt{Cprog} \quad \widehat{=} \quad \langle s_1, s_2, s_3, s_4, s_5, s_6 \rangle$$

We shall use σ_i as shorthand for $\langle s_i, \ldots, s_6 \rangle$, so $\texttt{cprog} = \sigma_1$, and $\sigma_i = s_i : \sigma_{i+1}$, for $i < 6$.

$s_1 \widehat{=}$
Asg f
    App (Const fopen)
        Tuple Const "a"
            Const FRead
= `f = fopen ("a",Fread)`
$s_2 \widehat{=}$
Asg x
    App (Const freadi)
        Var f
= `x = freadi(f)`
$s_3 \widehat{=}$
Call (Const fclose) (Var f)
= `fclose(f)`
$s_4 \widehat{=}$
Asg f

App (Const fopen)
Tuple Const "a"
Const FWrite
= `f = fopen ("a",FWrite)`
$s_5 \mathrel{\widehat{=}}$
Call
App (Const fwritei)
Tuple Var f
App (Const *)
Tuple Var x
Var x
= `fwrite(f,x*x)`
$s_6 \mathrel{\widehat{=}}$
Call (Const fclose) (Var f)
= `f = fclose(f)`

7.2.2 The Proof

$$
\begin{array}{ll}
 & \mathsf{P}_C[\![\texttt{Cprog}]\!]\mathcal{W} \\
= & \quad \langle \text{defn. of } \texttt{Cprog} \rangle \\
 & \mathsf{P}_C[\![\sigma_1]\!]\mathcal{W} \\
= & \quad \langle \text{defn. of } \mathsf{P}_C \rangle \\
 & \pi_1(\mathsf{SS}_C[\![\sigma_1]\!](\mathcal{W}, \langle\theta\rangle, \theta)) \\
= & \quad \langle \text{defn. of } \sigma_1 \rangle \\
 & \pi_1(\mathsf{SS}_C[\![s_1 : \sigma_2]\!](\mathcal{W}, \langle\theta\rangle, \theta)) \\
= & \quad \langle \text{defn. of } \mathsf{SS}_C, \circ \rangle \\
 & \pi_1(\mathsf{SS}_C[\![\sigma_2]\!]((\mathsf{S}_C[\![s_1]\!])(\mathcal{W}, \langle\theta\rangle, \theta)))
\end{array}
$$

We now introduce a shorthand:

$$\mathcal{S}_i(x) \mathrel{\widehat{=}} \pi_1(\mathsf{SS}_C[\![\sigma_i]\!](x))$$

noting the following property

$$\mathcal{S}_i(x) = \mathcal{S}_{i+1}(\mathsf{S}_C[\![s_i]\!](x))$$

(by defn.of $\mathsf{SS}_C, \circ$).
We continue:

$$
\begin{array}{ll}
 & \pi_1(\mathsf{SS}_C[\![\sigma_2]\!]((\mathsf{S}_C[\![s_1]\!])(\mathcal{W}, \langle\theta\rangle, \theta))) \\
= & \quad \langle \text{shorthand } i = 2 \rangle \\
 & \mathcal{S}_2(\mathsf{S}_C[\![s_1]\!](\mathcal{W}, \langle\theta\rangle, \theta)) \\
= & \quad \langle \text{shorthand } s_1 \rangle \\
 & \mathcal{S}_2(\mathsf{S}_C[\![\texttt{f=fopen("a",FRead)}]\!](\mathcal{W}, \langle\theta\rangle, \theta)) \\
= & \quad \langle \text{defn. } \mathcal{S}_C \text{ on } \textsc{Asg} \rangle \\
 & \mathcal{S}_2(\textbf{let } (r, (\mathcal{W}', \rho', \varrho')) = \mathsf{E}_C[\![\texttt{fopen("a",FRead)}]\!](\mathcal{W}, \langle\theta\rangle, \theta) \\
 & \quad \textbf{in } (\mathcal{W}', \rho' \dagger \{\texttt{f} \mapsto r\}, \varrho') \\
 & \quad)
\end{array}
$$

We introduce the following shorthands (see also Lemma Cd.1)

$$\begin{array}{rcl} \Phi_1 & \widehat{=} & \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \varsigma)\} \\ f_1 & \widehat{=} & \textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \varsigma \\ \rho_1 & \widehat{=} & \{\mathtt{f} \mapsto 1\} \\ \varrho_1 & \widehat{=} & \{1 \mapsto f_1\} \end{array}$$

We continue

$$\begin{array}{cl} & \mathcal{S}_2(\textbf{let}\ (r, (\mathcal{W}', \rho', \varrho')) = \mathsf{E}_C[\![\mathtt{fopen("a",FRead)}]\!](\mathcal{W}, \langle\theta\rangle, \theta) \\ & \quad \textbf{in}\ (\mathcal{W}', \rho' \dagger \{\mathtt{f} \mapsto r\}, \varrho')\ ) \\ = & \quad \langle \text{Lemma Cd.1} \rangle \\ & \mathcal{S}_2(\textbf{let}\ (r, (\mathcal{W}', \rho', \varrho')) = (1, (((\Phi_1), w), \langle\theta\rangle, \varrho_1) \\ & \quad \textbf{in}\ (\mathcal{W}', \rho' \dagger \{\mathtt{f} \mapsto r\}, \varrho')\ ) \\ = & \quad \langle \text{Let clause} \rangle \\ & \mathcal{S}_2((\Phi_1, w), \langle\theta\rangle \dagger \{\mathtt{f} \mapsto 1\}, \varrho_1) \\ = & \quad \langle \text{override on } \mathit{PEnv}, \text{ shorthand} \rangle \\ & \mathcal{S}_2((\Phi_1, w), \langle\rho_1\rangle, \varrho_1) \\ = & \quad \langle \text{Prop. of } \mathcal{S}_i \rangle \\ & \mathcal{S}_3(\mathsf{S}_C[\![s_2]\!]((\Phi_1, w), \langle\rho_1\rangle, \varrho_1)) \\ = & \quad \langle \text{shorthand } s_2 \rangle \\ & \mathcal{S}_3(\mathsf{S}_C[\![\mathtt{x=freadi(f)}]\!]((\Phi_1, w), \langle\rho_1\rangle, \varrho_1)) \\ = & \quad \langle \text{defn. } \mathcal{S}_C \text{ on } \textsc{Asg} \rangle \\ & \mathcal{S}_3(\textbf{let}\ (r, (\mathcal{W}', \rho', \varrho')) = \mathsf{E}_C[\![\mathtt{freadi(f)}]\!]((\Phi_1, w), \langle\rho_1\rangle, \varrho_1) \\ & \quad \textbf{in}\ (\mathcal{W}', \rho' \dagger \{\mathtt{x} \mapsto r\}, \varrho')\ ) \end{array}$$

We introduce the following shorthands (see also Lemma Cd.3)

$$\begin{array}{rcl} f_2 & \widehat{=} & \textsc{HRead}\ \text{"}a\text{"}\ \langle j\rangle\ \varsigma \\ \rho_2 & \widehat{=} & \{\mathtt{f} \mapsto 1, \mathtt{x} \mapsto J\} \\ \varrho_2 & \widehat{=} & \{1 \mapsto f_2\} \end{array}$$

We continue

$$\begin{array}{cl} & \mathcal{S}_3(\textbf{let}\ (r, (\mathcal{W}', \rho', \varrho')) = \mathsf{E}_C[\![\mathtt{freadi(f)}]\!]((\Phi_1, w), \langle\rho_1\rangle, \varrho_1) \\ & \quad \textbf{in}\ (\mathcal{W}', \rho' \dagger \{\mathtt{x} \mapsto r\}, \varrho')\ ) \\ = & \quad \langle \text{Lemma Cd.3} \rangle \\ & \mathcal{S}_3(\textbf{let}\ (r, (\mathcal{W}', \rho', \varrho')) = (J, ((\Phi_1, w), \langle\rho_1\rangle, \varrho_2)) \\ & \quad \textbf{in}\ (\mathcal{W}', \rho' \dagger \{\mathtt{x} \mapsto r\}, \varrho')\ ) \\ = & \quad \langle \text{let clause} \rangle \\ & \mathcal{S}_3((\Phi_1, w), \langle\rho_1\rangle \dagger \{\mathtt{x} \mapsto J\}, \varrho_2\ ) \\ = & \quad \langle \text{override defn., shorthand} \rangle \\ & \mathcal{S}_3((\Phi_1, w), \langle\rho_2\rangle, \varrho_2\ ) \\ = & \quad \langle \text{prop. of } \mathcal{S}_i \rangle \\ & \mathcal{S}_4(\mathsf{S}_C[\![s_3]\!]((\Phi_1, w), \langle\rho_2\rangle, \varrho_2)\ ) \\ = & \quad \langle \text{shorthand } s_3 \rangle \\ & \mathcal{S}_4(\mathsf{S}_C[\![\mathtt{fclose(f)}]\!]((\Phi_1, w), \langle\rho_2\rangle, \varrho_2)\ ) \\ = & \quad \langle \text{Defn. of } \mathsf{S}_C \text{ on } \textsc{Call} \rangle \\ & \mathcal{S}_4(\textbf{let}\ (a', \varepsilon') = \mathsf{E}_C[\![\mathtt{f}]\!]((\Phi_1, w), \langle\rho_2\rangle, \varrho_2) \\ & \quad \textbf{in}\ \pi_2(\mathsf{App}_C[\![\mathtt{fclose}]\!](a', \varepsilon'))\ ) \end{array}$$

$$
\begin{array}{ll}
= & \langle \text{Defn. of } \mathsf{E}_C \text{ on } \textsc{Var}\text{, shorthands}\rangle \\
& \mathcal{S}_4(\textbf{let } (a', \varepsilon') = (1, ((\Phi_1, w), \langle \rho_2 \rangle, \varrho_2)) \\
& \quad \textbf{in } \pi_2(\mathsf{App}_C[\![\texttt{fclose}]\!](a', \varepsilon'))\ ) \\
= & \langle \text{let clause} \rangle \\
& \mathcal{S}_4(\pi_2(\mathsf{App}_C[\![\texttt{fclose}]\!](1, ((\Phi_1, w), \langle \rho_2 \rangle, \varrho_2))\ ))
\end{array}
$$

We introduce the following shorthand (see also Lemma Cd.4)

$$\Phi_3 \quad \widehat{=} \quad \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \varsigma)\}$$

We continue

$$
\begin{array}{ll}
& \mathcal{S}_4(\pi_2(\mathsf{App}_C[\![\texttt{fclose}]\!](1, ((\Phi_1, w), \langle \rho_2 \rangle, \varrho_2))\ )) \\
= & \langle \text{Lemma Cd.4} \rangle \\
& \mathcal{S}_4(\pi_2(!, ((\Phi_3, w), \langle \rho_2 \rangle, \theta))) \\
= & \langle \text{projection} \rangle \\
& \mathcal{S}_4((\Phi_3, w), \langle \rho_2 \rangle, \theta)
\end{array}
$$

7.2.3 Lemma Cd.1

$$
\begin{array}{ll}
& \mathsf{E}_C[\![\texttt{fopen("a",FRead)}]\!](\mathcal{W}, \langle \theta \rangle, \theta) \\
= & \langle \text{defn. of } \mathsf{E}_C \text{ on } \textsc{App} \rangle \\
& \mathsf{App}_C[\![\texttt{fopen}]\!](\mathsf{E}_C[\![\texttt{("a",FRead)}]\!](\mathcal{W}, \langle \theta \rangle, \theta)) \\
= & \langle \text{defn. of } \mathsf{E}_C, \mathsf{C} \text{ on } \textsc{Tuple}, \textsc{Const} \rangle \\
& \mathsf{App}_C[\![\texttt{fopen}]\!](\langle \text{"}a\text{"}, \text{F}\textsc{Read} \rangle, (\mathcal{W}, \langle \theta \rangle, \theta)) \\
= & \langle \text{defn. of } \mathcal{W} \rangle \\
& \mathsf{App}_C[\![\texttt{fopen}]\!](\langle \text{"}a\text{"}, \text{F}\textsc{Read} \rangle, ((\Phi \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \varsigma)\}, w), \langle \theta \rangle, \theta)) \\
= & \langle \text{defn. of } \mathsf{App}_C \text{ on } \textsc{fopen} \rangle \\
& \textbf{let } (f, \Phi') = \texttt{fopen}[\text{"}a\text{"}, \text{F}\textsc{Read}](\Phi \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \varsigma)\}) \\
& \textbf{in let } (h, \varrho') = \text{hAlloc}[f]\theta \\
& \textbf{in } (h, ((\Phi', w), \langle \theta \rangle, \varrho')) \\
= & \langle \text{Lemma Cd.2} \rangle \\
& \textbf{let } (f, \Phi') = ((\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \varsigma), (\Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \varsigma)\})) \\
& \textbf{in let } (h, \varrho') = \text{hAlloc}[f]\theta \\
& \textbf{in } (h, ((\Phi', w), \langle \theta \rangle, \varrho')) \\
= & \langle \text{defn. of hAlloc}, \max \rangle \\
& \textbf{let } (f, \Phi') = ((\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \varsigma), (\Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \varsigma)\})) \\
& \textbf{in let } (h, \varrho') = (1, \{1 \mapsto f\}) \\
& \textbf{in } (h, ((\Phi', w), \langle \theta \rangle, \varrho')) \\
= & \langle \text{2nd let clause} \rangle \\
& \textbf{let } (f, \Phi') = ((\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \varsigma), (\Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \varsigma)\})) \\
& \textbf{in } (1, ((\Phi', w), \langle \theta \rangle, \{1 \mapsto f\})) \\
= & \langle \text{1st let clause} \rangle \\
& (1, ((\Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \varsigma)\}), w), \langle \theta \rangle, \{1 \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \varsigma)\})) \\
= & \langle \text{shorthand} \rangle \\
& (1, (((\Phi_1), w), \langle \theta \rangle, \{1 \mapsto f_1\}))
\end{array}
$$

7.2.4 Lemma Cd.2

$$\texttt{fopen}[\text{"}a\text{"}, \text{F}\textsc{Read}](\Phi \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \varsigma)\})$$

$=$ ⟨defn. of `fopen`⟩
$((\textsc{HRead}\ "a", \Lambda, \delta), (\Phi \sqcup \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\} \dagger \{"a" \mapsto (\textsc{Read}\ r, \delta)\}))$
where $\delta = (\pi_2(\Phi \sqcup \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\})("a"))$
and $r = \pi_1((\Phi \sqcup \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\})("a")) = \textsc{Closed}\ \rightarrow\ 1\ ,\ \ldots$
$=$ ⟨map lookup properties⟩
$((\textsc{HRead}\ "a", \Lambda, \delta), (\Phi \sqcup \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\} \dagger \{"a" \mapsto (\textsc{Read}\ r, \delta)\}))$
where $\delta = \pi_2(\textsc{Closed}, J : \varsigma)$
and $r = \pi_1(\textsc{Closed}, J : \varsigma) = \textsc{Closed}\ \rightarrow\ 1\ ,\ \ldots$
$=$ ⟨projection, conditional⟩
$((\textsc{HRead}\ "a", \Lambda, \delta), (\Phi \sqcup \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\} \dagger \{"a" \mapsto (\textsc{Read}\ r, \delta)\}))$
where $\delta = J : \varsigma$ **and** $r = 1$
$=$ ⟨map property — override after extend⟩
$((\textsc{HRead}\ "a", \Lambda, \delta), (\Phi \dagger \{"a" \mapsto (\textsc{Read}\ r, \delta)\}))$
where $\delta = J : \varsigma$ **and** $r = 1$
$=$ ⟨where clause⟩
$((\textsc{HRead}\ "a"\ \Lambda\ J : \varsigma), (\Phi \dagger \{"a" \mapsto (\textsc{Read}\ 1, J : \varsigma)\}))$

7.2.5 Lemma Cd.3

$\mathsf{E}_C[\![\texttt{freadi(f)}]\!]((\Phi_1, w), \langle\rho_1\rangle, \varrho_1)$
$=$ ⟨defn. of E_C on $\textsc{App}$⟩
$\mathsf{App}_C[\![\texttt{freadi}]\!](\mathsf{E}_C[\![\texttt{f}]\!]((\Phi_1, w), \langle\rho_1\rangle, \varrho_1)$
$=$ ⟨defn. of E_C on $\textsc{Var}$, shorthand ⟩
$\mathsf{App}_C[\![\texttt{freadi}]\!](1, ((\Phi_1, w), \langle\rho_1\rangle, \varrho_1)$
$=$ ⟨defn. of App_C on $\textsc{freadi}$⟩
let $(i, f') = \texttt{freadi}(\varrho_1(1))$
in let $\varrho' = \varrho_1 \dagger \{1 \mapsto f'\}$
in $(i, ((\Phi_1, w), \langle\rho_1\rangle, \varrho')))$
$=$ ⟨shorthands, map application⟩
let $(i, f') = \texttt{freadi}(\textsc{HRead}\ "a"\ \Lambda\ J : \varsigma)$
in let $\varrho' = \varrho_1 \dagger \{1 \mapsto f'\}$
in $(i, ((\Phi_1, w), \langle\rho_1\rangle, \varrho')))$
$=$ ⟨defn. `freadi`⟩
let $(i, f') = (J, \textsc{HRead}\ "a"\ \langle j\rangle\ \varsigma)$
in let $\varrho' = \varrho_1 \dagger \{1 \mapsto f'\}$
in $(i, ((\Phi_1, w), \langle\rho_1\rangle, \varrho')))$
$=$ ⟨both let clauses⟩
$(J, ((\Phi_1, w), \langle\rho_1\rangle, \varrho_1 \dagger \{1 \mapsto \textsc{HRead}\ "a"\ \langle j\rangle\ \varsigma\})))$
$=$ ⟨defn. of override⟩
$(J, ((\Phi_1, w), \langle\rho_1\rangle, \{1 \mapsto \textsc{HRead}\ "a"\ \langle j\rangle\ \varsigma\})))$
$=$ ⟨shorthands⟩
$(J, ((\Phi_1, w), \langle\rho_1\rangle, \varrho_2))$

7.2.6 Lemma Cd.4

$\mathsf{App}_C[\![\texttt{fclose}]\!](1, ((\Phi_1, w), \langle\rho_2\rangle, \varrho_2))$
$=$ ⟨defn. of App_C on $\textsc{fclose}$⟩
let $\Phi' = \texttt{fclose}[\varrho_2(1)]\Phi_1$

$$
\begin{array}{ll}
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{map lookup} \rangle \\
 & \textbf{let } \Phi' = \texttt{fclose}[\textsc{HRead}\ "a"\ \langle j \rangle\ \varsigma]\Phi_1 \\
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{defn. of } \texttt{fclose} \rangle \\
 & \textbf{let } \Phi' = \Phi_1 \dagger \{"a" \mapsto (s, \delta)\} \\
 & \qquad\qquad \textbf{where } ((\textsc{Read}\ r), \delta) = \Phi_1("a") \\
 & \qquad\qquad \textbf{and } s = r = 1 \ \rightarrow\ \textsc{Closed}\ ,\ \ldots \\
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{map lookup on } \Phi_1 \rangle \\
 & \textbf{let } \Phi' = \Phi_1 \dagger \{"a" \mapsto (s, \delta)\} \\
 & \qquad\qquad \textbf{where } (\textsc{Read}\ r, \delta) = (\textsc{Read}\ 1, J : \varsigma) \\
 & \qquad\qquad \textbf{and } s = r = 1 \ \rightarrow\ \textsc{Closed}\ ,\ \ldots \\
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{1st where clause} \rangle \\
 & \textbf{let } \Phi' = \Phi_1 \dagger \{"a" \mapsto (s, J : \varsigma)\} \\
 & \qquad\qquad \textbf{where } s = 1 = 1 \ \rightarrow\ \textsc{Closed}\ ,\ \ldots \\
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{conditional} \rangle \\
 & \textbf{let } \Phi' = \Phi_1 \dagger \{"a" \mapsto (s, J : \varsigma)\} \\
 & \qquad\qquad \textbf{where } s = \textsc{Closed} \\
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{where-clause} \rangle \\
 & \textbf{let } \Phi' = \Phi_1 \dagger \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\} \\
 & \textbf{in let } \varrho' = \mathrm{hFree}[1]\varrho_2 \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{shorthands, defn of hFree} \rangle \\
 & \textbf{let } \Phi' = \Phi_1 \dagger \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\} \\
 & \textbf{in let } \varrho' = \theta \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{override, shorthands} \rangle \\
 & \textbf{let } \Phi' = \Phi \dagger \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\} \\
 & \textbf{in let } \varrho' = \theta \\
 & \textbf{in } (!, ((\Phi', w), \langle \rho_2 \rangle, \varrho')) \\
= & \quad \langle \text{let clauses} \rangle \\
 & (!, ((\Phi \dagger \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\}, w), \langle \rho_2 \rangle, \theta)) \\
= & \quad \langle \text{shorthands} \rangle \\
 & (!, ((\Phi_3, w), \langle \rho_2 \rangle, \theta))
\end{array}
$$

7.3 Proof for Clean Program

7.3.1 Clean Program Labelled Syntax

The program is a 6-pronged hash-let:

$$\texttt{Kprog} \quad \widehat{=} \quad (\texttt{w}, \textsc{Hash}\ \langle h_1, h_2, h_3, h_4, h_5, h_6 \rangle\ \texttt{w})$$

We adopt ζ_i as shorthand for $\langle h_i, \ldots, h_6 \rangle$
The annotated,labelled syntax is:

```
w
Hash
h1 =
       (Tuple (Var f)
                 Var w  )
          App (Const fopen)
                Const "a"
                Const FRead
                Var w
= # (f,w) = fopen "a" Fread w
h2 =
       Tuple (Var x)
                Var f
          App (Const freadi)
                Var f
= # (x,f) = freadi f
h3 =
       Var w
          App (Const fclose)
                Var f
                Var w
= # w = fclose f w
h4 =
       Tuple (Var f)
                Var w
          App (Const fopen)
                Const "a"
                Const FWrite
                Var w
= # (f,w) = fopen "a" Fwrite w
h5 =
       Var f
          App (Const fwritei)
                Var f
                App (Const *)
                      Tuple Var x
                              Var x
= # f = fwritei f (x*x)
h6 =
       Var w
```

$$
\begin{array}{l}
\quad\quad \textsc{App}\ (\textsc{Const}\ \textsc{fclose}) \\
\quad\quad\quad\quad \textsc{Var}\ \mathrm{f} \\
\quad\quad\quad\quad \textsc{Var}\ \mathrm{w}
\end{array}
$$

= `# f = fclose f w`

Var w

7.3.2 The Proof

$$
\begin{array}{cl}
 & \mathsf{P}_K[\![\texttt{Kprog}]\!]\mathcal{W} \\
= & \quad \langle \text{defn. of } \texttt{Kprog} \rangle \\
 & \mathsf{P}_K[\![(\texttt{w}, \textsc{Hash}\ \zeta_1\ \texttt{w})]\!]\mathcal{W} \\
= & \quad \langle \text{defn. of } \mathsf{P}_K \rangle \\
 & \mathsf{E}_K\{\texttt{w} \mapsto \mathcal{W}\}[\![\textsc{Hash}\ \zeta_1\ \texttt{w}]\!]
\end{array}
$$

We introduce a shorthand $\ell_0 = \{\texttt{w} \mapsto \mathcal{W}\}$

$$
\begin{array}{cl}
 & \mathsf{E}_K\{\texttt{w} \mapsto \mathcal{W}\}[\![\textsc{Hash}\ \zeta_1\ \texttt{w}]\!] \\
= & \quad \langle \text{expand } \zeta_1 \rangle \\
 & \mathsf{E}_K\ell_0[\![\textsc{Hash}\ (\texttt{(f,w)=fopen "a" FRead w}) : \zeta_2\ \texttt{w}]\!] \\
= & \quad \langle \text{defn, } \mathsf{E}_K \rangle \\
 & \mathsf{E}_K(\ell_0 \dagger \mathsf{M}_K[\![\texttt{f,w}]\!](\mathsf{E}_K\ell_0[\![\texttt{fopen "a" FRead w}]\!]))[\![\textsc{Hash}\ \zeta_2\ \texttt{w}]\!]
\end{array}
$$

We introduce more shorthands (see also Lemma Kd.1):

$$
\begin{array}{rcl}
f_1 & \widehat{=} & \textsc{HRead}\ "a"\ \Lambda\ J : \varsigma \\
\Phi_1 & \widehat{=} & \Phi \sqcup \{"a" \mapsto (\textsc{Read}\ 1, J : \varsigma)\} \\
\ell_1 & \widehat{=} & \{\texttt{f} \mapsto f_1, \texttt{w} \mapsto (\Phi_1, w)\}
\end{array}
$$

we continue

$$
\begin{array}{cl}
 & \mathsf{E}_K(\ell_0 \dagger \mathsf{M}_K[\![\texttt{f,w}]\!](\mathsf{E}_K\ell_0[\![\texttt{fopen "a" FRead w}]\!]))[\![\textsc{Hash}\ \zeta_2\ \texttt{w}]\!] \\
= & \quad \langle \text{Lemma Kd.1} \rangle \\
 & \mathsf{E}_K(\ell_0 \dagger \mathsf{M}_K[\![\texttt{f,w}]\!](f_1, (\Phi_1, w)))[\![\textsc{Hash}\ \zeta_2\ \texttt{w}]\!] \\
= & \quad \langle \text{defn. of } \mathsf{M}_K \text{ on } \textsc{Tuple}\text{, map, reduce} \rangle \\
 & \mathsf{E}_K(\ell_0 \dagger \{\texttt{f} \mapsto f_1, \texttt{w} \mapsto (\Phi_1, w)\})[\![\textsc{Hash}\ \zeta_2\ \texttt{w}]\!] \\
= & \quad \langle \text{override} \rangle \\
 & \mathsf{E}_K\{\texttt{f} \mapsto f_1, \texttt{w} \mapsto (\Phi_1, w)\}[\![\textsc{Hash}\ \zeta_2\ \texttt{w}]\!] \\
= & \quad \langle \text{shorthand} \rangle \\
 & \mathsf{E}_K\ell_1[\![\textsc{Hash}\ \zeta_2\ \texttt{w}]\!]
\end{array}
$$

7.3.3 Lemma Kd.1

$$
\begin{array}{cl}
 & \mathsf{E}_K\ell_0[\![\texttt{fopen "a" FRead w}]\!] \\
= & \quad \langle \text{defn. } \mathsf{E}_K \text{ on } \textsc{App} \rangle \\
 & (\mathsf{E}_K\ell_0[\![\texttt{fopen}]\!])((\mathsf{E}_K\ell_0)^\star[\![\texttt{"a",FRead,w}]\!]) \\
= & \quad \langle \text{map, defn. } \mathsf{E}_K \text{ on } \textsc{Const}, \textsc{Var}\text{, currying} \rangle \\
 & \textsc{fopen}\langle "a", \textsc{FRead} \rangle \mathcal{W} \\
= & \quad \langle \text{defn. of } \textsc{fopen} \text{ in } \mathsf{C}_K\text{, currying, application} \rangle \\
 & (f, (\Phi', w)) \\
 & \textbf{where } (f, \Phi') = \texttt{fopen}["a", \textsc{FRead}](\Phi \sqcup \{"a" \mapsto (\textsc{Closed}, J : \varsigma)\})
\end{array}
$$

$$
\begin{array}{ll}
= & \langle \text{Lemma C.2} \rangle \\
 & (f, (\Phi', w)) \\
 & \textbf{where } (f, \Phi') = ((\textsc{HRead}\ ''a''\ \Lambda\ J : \varsigma), (\Phi \dagger \{''a'' \mapsto (\textsc{Read}\ 1, J : \varsigma)\})) \\
= & \langle \text{where-clause} \rangle \\
 & ((\textsc{HRead}\ ''a''\ \Lambda\ J : \varsigma), (\Phi \dagger \{''a'' \mapsto (\textsc{Read}\ 1, J : \varsigma)\}, w)) \\
= & \langle \text{shorthand} \rangle \\
 & (f_1, (\Phi_1, w))
\end{array}
$$

8 Language-Based Semantics

These semantics are operational in character, being, in the main, transformation laws or inference rules that preserve a programs meaning.

8.1 C Language Semantics

8.1.1 Hoare Triple Rules

From [HJ98, pp64–5] with change of notation.

$$
\begin{array}{rcl}
\{p\}\texttt{Q}\{r\}, \{p\}\texttt{Q}\{s\} & \vdash & \{p\}\texttt{Q}\{r \wedge s\} \\
\{p\}\texttt{Q}\{r\}, \{q\}\texttt{Q}\{r\} & \vdash & \{p \vee q\}\texttt{Q}\{r\} \\
\{p\}\texttt{Q}\{r\} & \vdash & \{p \wedge q\}\texttt{Q}\{r \vee s\} \\
 & \vdash & \{r(e)\}\texttt{x=e}\{r(x)\} \\
\{p \wedge b\}\texttt{Q1}\{r\}, \{p \wedge \neg b\}\texttt{Q2}\{r\} & \vdash & \{p\}\texttt{if b then Q1 else Q2}\{r\} \\
\{p\}\texttt{Q1}\{s\}, \{s\}\texttt{Q2}\{r\} & \vdash & \{p\}\texttt{Q1;Q2}\{r\}
\end{array}
$$

We can deduce the following:

$$\{R\}\texttt{x=e}\{R \wedge x = e\}$$

from the assignment rule by taking $r(z) \mathrel{\widehat{=}} R \wedge z = e$, as long as R does *not* mention x [HJ98, p30]

8.1.2 wp-rules

From [HJ98, p66] with change of notation

$$
\begin{array}{rcl}
\wp\,[\texttt{x=e}]\{r(x)\} & \widehat{=} & r(e) \\
\wp\,[\texttt{P;Q}]\{r\} & \widehat{=} & \wp\,[\texttt{P}]\{\wp\,[\texttt{Q}]\{r\}\} \\
\wp\,[\texttt{if b then P else Q}]\{r\} & \widehat{=} & b \ \rightarrow\ \wp\,[\texttt{P}]\{r\}\ ,\ \wp\,[\texttt{Q}]\{r\} \\
[r \Rightarrow s] & \vdash & [\wp\,[\texttt{Q}]\{r\} \Rightarrow \wp\,[\texttt{Q}]\{s\}] \\
[Q \Rightarrow S] & \vdash & [\wp\,[\texttt{S}]\{r\} \Rightarrow \wp\,[\texttt{Q}]\{r\}]
\end{array}
$$

8.1.3 C Program Language Semantics

We assume three global program variables `WORLD`, `FS`, `FSH`, denoting the world, it's file-system component and a file system handle environment, with corresponding semantic variables $\mathcal{W}$: *World*, Φ : *FS* and ϱ : *HMap FStatus*. We assume that FS is a component of WORLD, which is a C-struct. We also assume the existence of maps and map manipulators in the C-language. We also introduce an program environment ($\rho : PEnv$) into the semantic domain.

```
WORLD = (FS,...)
```

The C-program mainline initialises FSH

We define the meaning of

$$\{P\}\texttt{main()}\{\texttt{cstmts}\}\{Q\}$$

as being

$$\{P\}\texttt{FSH=nullmap;cstmts}\{Q\}$$

which simplifies to

$$\{P \wedge \varrho = \theta\}\texttt{cstmts}\{Q\}$$

8.1.4 I/O Model in Hoare Triple Form

Hoare-Triple form of `fopen`

$\{\ n \in \texttt{dom}\ \Phi \wedge \pi_1\Phi(n) \neq \textsc{Write}\ \}$

```
h = fopen(n,Fread)
```

$$\left\{\begin{array}{l} h' = \max(\texttt{dom}\ \varrho) + 1 \\ \varrho' = \varrho \sqcup \{h' \mapsto (\textsc{Hread}\ n\ \Lambda\ \pi_2(\Phi(n)))\} \\ \Phi' = \Phi \dagger \{n \mapsto (\textsc{Read}\ r, \pi_2(\Phi(n)))\} \\ \textbf{where}\ r = \pi_1(\Phi(n)) \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(n))) + 1 \end{array}\right\}$$

$\{\ n \notin \texttt{dom}\ \Phi \vee \pi_1\Phi(n) = \textsc{Closed}\ \}$

```
h = fopen(n,FWrite)
```

$$\left\{\begin{array}{l} h' = \max(\texttt{dom}\ \varrho) + 1 \\ \varrho' = \varrho \sqcup \{h' \mapsto (\textsc{Hwrite}\ n\ \Lambda)\} \\ \Phi' = \Phi \dagger \{n \mapsto (\textsc{Write}, \Lambda)\} \end{array}\right\}$$

Hoare-Triple form of `fclose`

$\{\ h \in \texttt{dom}\ \varrho \wedge \varrho(h) = (\text{HR}\textsc{ead}\ n\ \_) \wedge n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_)\ \}$

```
fclose(h)
```

$$\left\{\begin{array}{l} \varrho' = \lhd[h]\varrho \\ \Phi' = \Phi \dagger \{n \mapsto (s, \pi_2(\Phi(n)))\} \\ \textbf{where}\ s = r = 1\ \rightarrow\ \textsc{Closed}\ ,\ \textsc{Read}\ (r-1) \end{array}\right\}$$

$\{\ h \in \texttt{dom}\ \varrho \wedge \varrho(h) = (\text{HW}\textsc{rite}\ n\ \delta) \wedge n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Write}, \_)\ \}$

```
fclose(h)
```

$$\left\{\begin{array}{l} \varrho' = \lhd[h]\varrho \\ \Phi' = \Phi \dagger \{n \mapsto (\textsc{Closed}, \delta)\} \end{array}\right\}$$

Hoare-Triple form of `fwritei`

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HW}\textsc{rite}\ n\ \delta)\ \}$

```
fwritei(h,i)
```

$\{\ \varrho' = \varrho \dagger \{h \mapsto (\text{HW}\textsc{rite}\ n\ \delta \frown \langle i \rangle)\}\ \}$

Hoare-Triple form of `freadi`

$\{\ h \in \varrho \wedge \varrho(h) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w)\ \}$

```
i = freadi(h)
```

$\{\ i' = J\ \wedge\ \varrho' = \varrho \dagger \{h \mapsto (\text{HR}\textsc{ead}\ n\ \delta_r \frown \langle J \rangle\ \delta_w)\}\ \}$

8.1.5 IO Model in C Language form

This model exists solely to be able to give a Hoare-Triple or WP semantics to the IO call. We define the behaviour using C like programming constructs as well as (ASCII forms of) modelling concepts such as maps, etc. We then use these to derive the relevant Hoare triples.

Derivation of Hoare Triple for `fopen` (Read). The call

```
h = fopen(n,FRead)
```

is equivalent to

$\{\ n \in \mathtt{dom}\ \Phi \wedge \pi_1\Phi(n) \neq \textsc{Write}\ \}$

```
f0 = lookup(PHI,n);
ds = snd(f);
r = fst(f0)==Closed ? 1 : fst(fst(f0))+1 ;
f = (Read r,ds);
PHI = override(PHI,n,f);
fs = Hread n [] ds;
(h,FSH) = hAlloc FSH fs;
```

We proceed to compute the post-condition:

$\{\ n \in \mathtt{dom}\ \Phi \wedge \pi_1\Phi(n) \neq \textsc{Write}\ \}$

```
f0 = lookup(PHI,n);
```

$\{\ f_0' = \Phi(n)\ \}$

```
ds = snd(f);
```

$\{\ \delta' = \pi_2 f_0'\ \}$

```
r = fst(f0)==Closed ? 1 : fst(fst(f0))+1 ;
```

$\{\ r' = \pi_1 f_0' \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(f_0')) + 1\ \}$

```
f = (Read r,ds);
```

$\{\ f' = (\textsc{Read}\ r', \delta')\ \}$

```
  PHI = override(PHI,n,f);
```
$\{\ \Phi' = \Phi \dagger \{n \mapsto f'\}\ \}$
```
  fs = Hread n [] ds;
```
$\{\ f'_s = \textsc{Hread}\ n\ \Lambda\ \delta'\ \}$
```
  (h,FSH) = hAlloc FSH fs;
```
$\{\ (h', \varrho') = \text{hAlloc}[f'_s]\varrho\ \}$

The variables visible outside `fopen` are h, Φ and ϱ, so we can summarise the overall effect of `fopen`(read) as:

$\{\ n \in \mathtt{dom}\ \Phi \wedge \pi_1\Phi(n) \neq \textsc{Write}\ \}$
```
 h = fopen(n,Fread)
```
$$\left\{\begin{array}{l} h' = \max(\mathtt{dom}\ \varrho) + 1 \\ \varrho' = \varrho \sqcup \{h' \mapsto (\textsc{Hread}\ n\ \Lambda\ \pi_2(\Phi(n)))\} \\ \Phi' = \Phi \dagger \{n \mapsto (\textsc{Read}\ r, \pi_2(\Phi(n)))\} \\ \textbf{where}\ r = \pi_1(\Phi(n)) \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(n))) + 1 \end{array}\right\}$$

Derivation of Hoare Triple for `fopen`(Write). The call

```
h = fopen(n,FWrite)
```

is equivalent to

$\{\ n \notin \mathtt{dom}\ \Phi \vee \pi_1\Phi(n) = \textsc{Closed}\ \}$
```
  f = (Write,[]);
  PHI = override(PHI,n,f);
  fs = Hwrite n [];
  (h,FSH) = hAlloc FSH fs;
```

We proceed to compute the post-condition:

$\{\ n \notin \mathtt{dom}\ \Phi \vee \pi_1\Phi(n) = \textsc{Closed}\ \}$
```
  f = (Write,[]);
```
$\{\ f' = (\textsc{Write}, \Lambda)\ \}$
```
  PHI = override(PHI,n,f);
```
$\{\ \Phi' = \Phi \dagger \{n \mapsto f'\}\ \}$
```
  fs = Hwrite n [];
```
$\{\ f'_s = \textsc{Hwrite}\ n\ \Lambda\ \}$
```
  (h,FSH) = hAlloc FSH fs;
```
$\{\ (h, \varrho') = \text{hAlloc}[f'_s]\varrho\ \}$

The variables visible outside are h, Φ and ϱ, so we can summarise the overall effect as:

$\{\ n \notin \mathtt{dom}\ \Phi \vee \pi_1\Phi(n) = \textsc{Closed}\ \}$
```
  h = fopen(n,FWrite)
```
$$\left\{\begin{array}{l} h' = \max(\mathtt{dom}\ \varrho) + 1 \\ \varrho' = \varrho \sqcup \{h' \mapsto (\textsc{Hwrite}\ n\ \Lambda)\} \\ \Phi' = \Phi \dagger \{n \mapsto (\textsc{Write}, \Lambda)\} \end{array}\right\}$$

Derivation of Hoare Triple for `fclose` (Read). The call

```
fclose(h)
```

where `h` is opened for reading, is equivalent to

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HR}\textsc{ead}\ n\ \_) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_)\ \}$

```
fs = lookup(FSH,h);
n = fst(fs);
(Read r,ds) = lookup(PHI,n);
s = r == 1 ? Closed : Read (r-1)

PHI' = override(PHI,n,(s,ds))
FSH' = hFree(h,FSH)
```

Computing the postcondtion:

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HR}\textsc{ead}\ n\ \_) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_)\ \}$

```
fs = lookup(FSH,h);
```

$\{\ f_s' = \varrho(n)\ \}$

```
n = fst(fs);
```

$\{\ n = \pi_1 f_s'\ \}$

```
(Read r,ds) = lookup(PHI,n);
```

$\{\ (\textsc{Read}\ r, \delta') = \Phi(n)\ \}$

```
s = r == 1 ? Closed : Read (r-1)
```

$\{\ s' = r = 1\ \rightarrow\ \textsc{Closed}\ ,\ \textsc{Read}\ (r-1)\ \}$

```
PHI' = override(PHI,n,(s,ds))
```

$\{\ \Phi' = \Phi \dagger \{n \mapsto (s', \delta')\}\ \}$

```
FSH' = hFree(h,FSH)
```

$\{\ \varrho' = \lhd[h]\varrho\ \}$

The variables visible are h, Φ and ϱ, so we can summarise the overall effect as:

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HR}\textsc{ead}\ n\ \_) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_)\ \}$

```
fclose(h)
```

$$\left\{\begin{array}{l} \varrho' = \lhd[h]\varrho \\ \Phi' = \Phi \dagger \{n \mapsto (s, \pi_2(\Phi(n)))\} \\ \textbf{where}\ s = r = 1\ \rightarrow\ \textsc{Closed}\ ,\ \textsc{Read}\ (r-1) \end{array}\right\}$$

Derivation of Hoare Triple for `fclose` (Write). The call

```
fclose(h)
```

where `h` is opened for writing, is equivalent to

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HW}\textsc{rite}\ n\ \delta) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Write},\_)\ \}$

```
  (Write n,ds) = lookup(FSH,h);
  PHI = override(PHI,n,(Closed,ds));
  FSH = hFree(h,FSH);
```

Computing the postcondition:

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HW}\textsc{rite}\ n\ \delta) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Write},\_)\ \}$

```
  (Write n,ds) = lookup(FSH,h);
```

$\{\ (\text{HW}\textsc{rite}\ n\,\delta) = \varrho(h)\ \}$

```
  PHI = override(PHI,n,(Closed,ds));
```

$\{\ \Phi' = \Phi \dagger n'(\textsc{Closed}, \delta)\ \}$

```
  FSH = hFree(h,FSH);
```

$\{\ \varrho' = \text{hFree}[h]\varrho\ \}$

The variables visible outside `fopen` are h, Φ and ϱ, so we can summarise the overall effect of `fopen`(Write) as:

$\{\ h \in \mathtt{dom}\ \varrho \wedge \varrho(h) = (\text{HW}\textsc{rite}\ n\ \delta) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Write},\_)\ \}$

```
  fclose(h)
```

$$\left\{\begin{array}{l} \varrho' = \lhd[h]\varrho \\ \Phi' = \Phi \dagger \{n \mapsto (\textsc{Closed}, \delta)\} \end{array}\right\}$$

Derivation of Hoare Triple for `fwritei` The call

```
fwrite(h,i)
```

is equivalent to

$\{\ h \in \varrho \wedge \varrho(h) = (\text{HW}\textsc{rite}\ n\ \delta\ \}$

```
 (HWrite n ds) = lookup(FSH,h);
 FSH=override(FSH,h,(HWrite n ds++[i]));
```

$\{\ \varrho' = \varrho \dagger \{h \mapsto (\text{HW}\textsc{rite}\ n\ \delta \frown \langle i \rangle)\}\ \}$

We obtain the post-condition immediately.

Derivation of Hoare Triple for `freadi` The call

```
i = freadi(f)
```

is equivalent to

$\{\ h \in \varrho \wedge \varrho(h) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w)\ \}$

```
 (HRead n dsr (J:dsw)) = lookup(FSH,h);
 i = J;
 FSH=override(FSH,h,(HRead n dsr dsw));
```

$\{\ i' = J\ \wedge\ \varrho' = \varrho \dagger \{h \mapsto (\text{HR}\textsc{ead}\ n\ \delta_r\ \delta_w)\}\ \}$

Again, we obtain the post-condition immediately.

8.2 Clean Language Semantics

We use the symbol `letb` in this rewrite to indicate that the scoping of this form is different to the scoping of the usual `let` expression in Clean and Haskell, as indicated by the let-evaluation rule.

```
 # p = expr1
 expr 2
```
= $\langle$ Hash Syntactic Sugar $\rangle$
```
 letb p = expr1 in expr 2
```

```
 (\x->b)e
```
= $\langle$ β-reduction $\rangle$
```
 b[x->e]
```

```
 letb v = e1 in e2
```
= $\langle$ Let Evaluation $\rangle$
```
 e2[v -> e1]
```

```
 letb (v1,v2) = (e1,e2) in e3
```
= $\langle$ Partial Let Evaluation $\rangle$
```
 letb v2 = e2 in e3[v1->e1]
```

```
 letb x1 = e1 in
 letb x2 = e2 in
 e3
```
= $\langle$ Let Swap — provided `x1,x2` not free in `e1,e2` $\rangle$
```
 letb x2 = e2 in
 letb x1 = e1 in
 e3
```

```
 e1 where x = e2
```
= $\langle$ Where Evaluation $\rangle$
```
 e1[x -> e2]
```

8.3 IO Model in Clean Language Form

```
pre_fopen (n,FWrite)(phi,_)
 = if (member(n,dom phi)) (fst(lookup phi n)==Closed) True

fopen (n,FWrite) (phi,rest)
 = (h,(override phi n f),rest))
   where
     h = HWrite n []
     f = (Write,[])
```

```
pre_fopen(n,FRead) (phi,_)
 = if (member(n,dom phi)) (fst(lookup phi n) != Write) False

fopen (n,FRead) (phi,rest)
 = (h,(override phi n f),rest)
   where
     h = HRead n [] ds
     f = (Read r,ds)
     f0 = lookup phi n
     r = if fst f0 == Closed then 1 else fst(fst f0)+1
     ds = snd f0

pre_fclose (HWrite n ds) (phi,_)
 = member(n,dom phi) && fst(lookup phi n)==Write

fclose (HWrite n ds) (phi,rest)
 = (override phi n (Closed,ds),rest)

pre_fclose (HRead n _) (phi,_)
 = member(n,dom phi) && fst(lookup phi n)==(Read _)

fclose (HRead n _) (phi,rest)
 = (override phi n (s,ds),rest)
   where
     (Read r,ds) = lookup phi n
     s = if r == 1 then Closed else (Read (s-1))

pre_fwritei  _ (HWrite _) = True
pre_fwritei  _ (HRead _)  = False

fwritei i (HWrite n ds) = Hwrite n (ds++[i])

pre_freadi (HWrite _)       = False
pre_freadi (HRead n rd rem) = rem != []

freadi (HRead n rd i:rest) = (i,Hread n rd++[i] rest)
```

8.4 Haskell Language Semantics

The “do” notation can be rewritten to use explicit bind, seq and lambda forms (this is defined in the Haskell report)

```
  x <- a
  b
```
= ⟨ `do` desugaring ⟩
```
  a >>= \x ->
  b
```
= ⟨ Bind elimination ⟩
```
  \w -> letb (val,w') = a w in b w'
```

```
  a
  b
```

= ⟨ `do` desugaring ⟩

```
  a >> b
```

= ⟨ Seq elimination ⟩

```
  \w -> letb w' = a w in b w'
```

Let evaluation, partial let evaluation, let swap, where evaluation all as Clean semantics.

8.5 IO Model in Haskell Language Form

The fopen, fclose, freadi and fwritei functions as for the Clean semantics.

"Handle" versions of the file operations also needed to encode the Haskell IO system.

```
:: IO a = (W,Hmap) -> (a, (W,Hmap))
:: Hmap = Int -> FStatus

openFile n m = \(w,l) -> (h, (w',override (h,fs) l))
               where   (fs,w') = fopen n m w
                       h       = hAlloc l

hreadi h = \(w,l) -> (the_int, (w, override (h,fs') l))
           where   (the_int,fs') = freadi fs
                   fs            = lookup h l

hwritei h i = \(w,l) -> (w, override (h,fs') l)
                        where fs' = fwritei i fs
                              fs  = lookup h l

hclose h = \(w,l) -> (w', remove h l)
                     where w' = fclose fs w
                           fs  = lookup h l

ReadMode = Fread
WriteMode = Fwrite

hAlloc [] = 1
hAlloc l  = (max dom l)+1
```

9 Language-Based Proofs

Language-based proofs are ones that work with the program text directly, possibly with some extra notation. The language based semantics from the previous section will be used to transform each program to a condition where the property to be proved can be seen immediately.

9.1 C Language Proof

We shall try using Hoare Triples to prove:

$\{ \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}), \_) \}$

```
Cprog
```

$\left\{ \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}), \_) \right\}$

This expands to

$\{ \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}), \_) \}$

```
main() { Cstmts }
```

$\left\{ \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}), \_) \right\}$

9.1.1 Condition Annotated Program.

$\{ P_0 \equiv \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}), \_) \wedge \varrho = \theta \}$

```
f = fopen("a",FRead)
```

$\{ P_1 \}$

```
x = freadi(f)
```

$\{ P_2 \}$

```
fclose(f)
```

$\{ P_3 \}$

```
f = fopen("a",FWrite)
```

$\{ P_4 \}$

```
fwritei(f,x*x)
```

$\{ P_5 \}$

```
fclose(f)
```

$\left\{ P_6 \Rightarrow \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}), \_) \right\}$

The proof for statement i (s_i) will proceed by showing that $P_i \Rightarrow \textit{pre-}s_i$, having identified the substitution that makes this so, then using this to generate P_{i+1}

9.1.2 C Statement 1

$$s_1 : \texttt{f = fopen("a",FRead)}$$

The pre-condition, with $n = \text{"}a\text{"}$ is

$$\text{"}a\text{"} \in \texttt{dom}\ \Phi \wedge \pi_1 \Phi(\text{"}a\text{"}) \neq \textsc{Write}$$

We have to show that P_0 implies this, so, assuming

$$\Phi = \Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}$$

we try to show the pre-condition is satisfied.

$$
\begin{array}{ll}
 & \text{"}a\text{"} \in \mathtt{dom}\,\Phi \wedge \pi_1\Phi(\text{"}a\text{"}) \neq \textsc{Write} \\
= & \quad \langle \text{Lemma C.1} \rangle \\
 & \textsc{True} \wedge \pi_1\Phi(\text{"}a\text{"}) \neq \textsc{Write} \\
= & \quad \langle \text{prop. calc., Lemma C.2} \rangle \\
 & \pi_1(\textsc{Closed}, J : \_) \neq \textsc{Write} \\
= & \quad \langle \text{defn.of proj.} \rangle \\
 & \textsc{Closed} \neq \textsc{Write} \\
= & \quad \langle \text{ineq.} \rangle \\
 & \textsc{True}
\end{array}
$$

The post-condition, with $n = \text{"}a\text{"}$ and $h = f$ is:

$$
\begin{array}{l}
f' = \max(\mathtt{dom}\ \varrho) + 1 \\
\varrho' = \varrho \sqcup \{f' \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ \pi_2(\Phi(\text{"}a\text{"})))\} \\
\Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ r, \pi_2(\Phi(\text{"}a\text{"})))\} \\
\textbf{where}\ r = \pi_1(\Phi(\text{"}a\text{"})) \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(\text{"}a\text{"}))) + 1
\end{array}
$$

We evaluate each term given the P_0 n as assumption.

$$
\begin{array}{ll}
 & f' = \max(\mathtt{dom}\ \varrho) + 1 \\
= & \quad \langle \text{val. of } \varrho \rangle \\
 & f' = \max\ \emptyset + 1 \\
= & \quad \langle \text{defn. of max} \rangle \\
 & f' = 0 + 1 \\
= & \quad \langle \text{arith.} \rangle \\
 & f' = 1
\end{array}
$$

$$
\begin{array}{ll}
 & \varrho' = \varrho \sqcup \{f' \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ \pi_2(\Phi(\text{"}a\text{"})))\} \\
= & \quad \langle \text{val. of } \varrho, f', \text{defn. } \sqcup \rangle \\
 & \varrho' = \{1 \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ \pi_2(\Phi(\text{"}a\text{"})))\} \\
= & \quad \langle \text{Lemma C.2} \rangle \\
 & \varrho' = \{1 \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ \pi_2(\textsc{Closed}, J : \_))\} \\
= & \quad \langle \text{defn. proj.} \rangle \\
 & \varrho' = \{1 \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \_)\}
\end{array}
$$

$$
\begin{array}{ll}
 & \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ r, \pi_2(\Phi(\text{"}a\text{"})))\} \\
 & \textbf{where}\ r = \pi_1(\Phi(\text{"}a\text{"})) \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(\text{"}a\text{"}))) + 1 \\
= & \quad \langle \text{Lemma C.2} \rangle \\
 & \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ r, \pi_2(\Phi(\text{"}a\text{"})))\} \\
 & \textbf{where}\ r = \pi_1(\textsc{Closed}, J : \_) \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(\text{"}a\text{"}))) + 1 \\
= & \quad \langle \text{defn. of proj} \rangle \\
 & \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ r, \pi_2(\Phi(\text{"}a\text{"})))\} \\
 & \textbf{where}\ r = \textsc{Closed} \equiv \textsc{Closed}\ \rightarrow\ 1\ ,\ \pi_1(\pi_1(\Phi(\text{"}a\text{"}))) + 1 \\
= & \quad \langle \text{cond.} \rangle \\
 & \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ r, \pi_2(\Phi(\text{"}a\text{"})))\} \\
 & \textbf{where}\ r = 1 \\
= & \quad \langle \text{where-clause.} \rangle \\
 & \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, \pi_2(\Phi(\text{"}a\text{"})))\} \\
 & \textbf{where}\ r = 1 \\
= & \quad \langle \text{val. of } \Phi \rangle \\
 & \Phi' = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}) \dagger \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, \pi_2(\Phi(\text{"}a\text{"})))\}
\end{array}
$$

$$
\begin{array}{ll}
= & \langle \text{prop. of override and extend} \rangle \\
& \Phi' = \Phi_0 \dagger \{"a" \mapsto (\textsc{Read}\ 1, \pi_2(\Phi("a")))\} \\
= & \langle \text{Lemma C.2} \rangle \\
& \Phi' = \Phi_0 \dagger \{"a" \mapsto (\textsc{Read}\ 1, \pi_2(\textsc{Closed}, J : \_))\} \\
= & \langle \text{defn. of proj.} \rangle \\
& \Phi' = \Phi_0 \sqcup \{"a" \mapsto (\textsc{Read}\ 1, J : \_)\}
\end{array}
$$

The postcondition becomes:

$$
\begin{array}{l}
f' = 1 \\
\varrho' = \{1 \mapsto (\textsc{HRead}\ "a"\ \Lambda\ J : \_)\} \\
\Phi' = \Phi_0 \sqcup \{"a" \mapsto (\textsc{Read}\ 1, J : \_\}
\end{array}
$$

We merge this with P_0 to obtain P_1, dropping primes:

$$
P_1 \equiv \left\{ \begin{array}{l}
\mathcal{W} = (\Phi_0 \sqcup \{"a" \mapsto (\textsc{Read}\ 1, J : \_)\}, \_) \\
\varrho = \{1 \mapsto (\textsc{HRead}\ "a"\ \Lambda\ J : \_)\} \\
f = 1
\end{array} \right.
$$

9.1.3 C Statement 2

$$s_2 : \texttt{x = freadi(f)}$$

We show that P_1 implies the pre-condition for this instance of `freadi`, which is

$$f \in \varrho \wedge \varrho(f) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w)$$

Assuming P_1, we show the pre-condition is satisfied:

$$
\begin{array}{ll}
& f \in \texttt{dom}\ \varrho \wedge \varrho(f) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w) \\
= & \langle \text{val. of } f, \varrho \rangle \\
& 1 \in \texttt{dom}\ \{1 \mapsto (\textsc{HRead}\ "a"\ \Lambda\ J : \_)\} \\
& \wedge \{1 \mapsto (\textsc{HRead}\ "a"\ \Lambda\ J : \_)\}(1) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w) \\
= & \langle \text{defn. of } \texttt{dom} \rangle \\
& \textsc{True}\ \wedge\ \{1 \mapsto (\textsc{HRead}\ "a"\ \Lambda\ J : \_)\}(1) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w) \\
= & \langle \text{prop. calc., map appl.} \rangle \\
& (\textsc{HRead}\ "a"\ \Lambda\ J : \_) = (\text{HR}\textsc{ead}\ n\ \delta_r\ J : \delta_w) \\
= & \langle \text{eq.} \rangle \\
& n = "a"\ \wedge\ \delta_r = \Lambda\ \wedge\ \delta_w = \_
\end{array}
$$

The precondition holds true under the given binding.
The post-condition of `freadi` with substitutions is:

$$x' = J\ \wedge\ \varrho' = \varrho \dagger \{1 \mapsto (\text{HR}\textsc{ead}\ "a"\ \Lambda \frown \langle J \rangle\ \_)\}$$

We evaluate each term given P_1 as assumption:

$$i' = J$$

$$
\begin{array}{ll}
& \varrho' = \varrho \dagger \{1 \mapsto (\text{HR}\textsc{ead}\ "a"\ \Lambda \frown \langle J \rangle\ \_)\} \\
= & \langle \text{defn. of conc.} \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \varrho' = \varrho \dagger \{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\} \\
= & \quad \langle \text{val. of } \varrho \rangle \\
 & \varrho' = \{1 \mapsto (\textsc{Hread}\ ''a''\ \Lambda\ J : \_)\} \dagger \{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\} \\
= & \quad \langle \text{defn. of override} \rangle \\
 & \varrho' = \{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\}
\end{array}
$$

The postcondition becomes:

$$
x' = J \ \wedge \ \varrho' = \{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\}
$$

We merge this with P_1, dropping primes, to get P_2:

$$
P_2 \equiv \begin{cases}
\mathcal{W} = (\Phi_0 \sqcup \{''a'' \mapsto (\textsc{Read}\ 1, J : \_)\}, \_) \\
\varrho = \{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\} \\
f = 1 \\
x = J
\end{cases}
$$

9.1.4 C Statement 3

$$
s_3 : \texttt{fclose(f)}
$$

We show that P_2 implies the pre-condition for this instance of `fclose`, which is

$$
f \in \texttt{dom}\ \varrho \wedge \varrho(f) = (\text{HR}\textsc{ead}\ n\ \_) \wedge n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_)
$$

Assuming P_2, we show the pre-condition is satisfied:

$$
\begin{array}{ll}
 & f \in \texttt{dom}\ \varrho \wedge \varrho(f) = (\text{HR}\textsc{ead}\ n\ \_) \wedge n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{val. of } f, \varrho \rangle \\
 & 1 \in \texttt{dom}\{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\} \wedge \{1 \mapsto (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_)\}(1) = (\text{HR}\textsc{ead}\ n\ \_) \\
 & \wedge\ n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{prop.of } \texttt{dom}, \text{ map appl.} \rangle \\
 & \textsc{True} \wedge (\text{HR}\textsc{ead}\ ''a''\ \langle J \rangle\ \_) = (\text{HR}\textsc{ead}\ n\ \_) \\
 & \wedge\ n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{prop. calc., eq.} \rangle \\
 & n = ''a'' \wedge n \in \texttt{dom}\ \Phi \wedge \Phi(n) = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{val. of } n \rangle \\
 & n = ''a'' \wedge ''a'' \in \texttt{dom}\ \Phi \wedge \Phi(''a'') = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{Lemma 1.} \rangle \\
 & n = ''a'' \wedge \textsc{True} \wedge \Phi(''a'') = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{prop. calc., Lemma 2. } (f_s = (\textsc{Read}\ 1, J : \_)) \rangle \\
 & n = ''a'' \wedge (\textsc{Read}\ 1, J : \_) = (\textsc{Read}\ r\ \_) \\
= & \quad \langle \text{eq.} \rangle \\
 & n = ''a'' \wedge r = 1
\end{array}
$$

The precondition holds true under the given binding.

The post-condition of `fclose` with these substitutions is:

$$
\begin{array}{l}
\varrho' = \lhd [f] \varrho \\
\Phi' = \Phi \dagger \{''a'' \mapsto (s, \pi_2(\Phi(''a'')))\} \\
\textbf{where } s = 1 = 1 \ \rightarrow \ \textsc{Closed}\ ,\ \textsc{Read}\ (1 - 1)
\end{array}
$$

We evaluate each term given P_2 as assumption:

$$
\begin{array}{ll}
& \varrho' = \lessdot[f]\varrho \\
= & \quad \langle \text{val. of } f, \varrho \rangle \\
& \varrho' = \lessdot[1]\{1 \mapsto \_\} \\
= & \quad \langle \text{defn. of } \mathit{mremove} \rangle \\
& \varrho' = \theta
\end{array}
$$

$$
\begin{array}{ll}
& \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (s, \pi_2(\Phi(\text{"}a\text{"})))\} \\
& \quad \textbf{where } s = 1 = 1 \;\rightarrow\; \textsc{Closed}\;,\; \textsc{Read}\;(1-1) \\
= & \quad \langle \text{defn. of cond.} \rangle \\
& \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (s, \pi_2(\Phi(\text{"}a\text{"})))\} \\
& \quad \textbf{where } s = \textsc{Closed} \\
= & \quad \langle \text{where clause} \rangle \\
& \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \pi_2(\Phi(\text{"}a\text{"})))\} \\
= & \quad \langle \text{Lemma C.2} \rangle \\
& \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \pi_2(\textsc{Read}\;1, J : \_))\} \\
= & \quad \langle \text{defn. of proj.} \rangle \\
& \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\} \\
= & \quad \langle \text{val. of } \Phi. \rangle \\
& \Phi' = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Read}\;1, J : \_)\}) \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\} \\
= & \quad \langle \text{prop. of } \dagger. \rangle \\
& \Phi' = \Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}
\end{array}
$$

The postcondition becomes:

$$
\begin{array}{l}
\varrho' = \theta \\
\Phi' = \Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}
\end{array}
$$

We merge this with P_2, dropping primes, to get P_3:

$$
P_3 \equiv \left\{
\begin{array}{l}
\mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}, \_) \\
\varrho = \theta \\
f = 1 \\
x = J
\end{array}
\right.
$$

9.1.5 C Statement 4

$$s_4 : \texttt{f = fopen("a",FWrite)}$$

We show that P_3 implies the pre-condition for this instance of `fopen`, which is

$$\text{"}a\text{"} \notin \texttt{dom}\;\Phi \vee \pi_1\Phi(\text{"}a\text{"}) = \textsc{Closed}$$

Assuming P_3, we show the pre-condition is satisfied:

$$
\begin{array}{ll}
& \text{"}a\text{"} \notin \texttt{dom}\;\Phi \vee \pi_1\Phi(\text{"}a\text{"}) = \textsc{Closed} \\
= & \quad \langle \text{Lemma C.1} \rangle \\
& \textsc{False} \vee \pi_1\Phi(\text{"}a\text{"}) = \textsc{Closed} \\
= & \quad \langle \text{prop. calc., Lemma C.2} \rangle \\
& \pi_1(\textsc{Closed}, J : \_) = \textsc{Closed} \\
= & \quad \langle \text{defn. proj.} \rangle \\
& \textsc{Closed} = \textsc{Closed} \\
= & \quad \langle \text{eq.} \rangle \\
& \textsc{True}
\end{array}
$$

The post-condition of `fopen` with substitutions is:

$$\begin{array}{l} f' = \max(\mathtt{dom}\ \varrho) + 1 \\ \varrho' = \varrho \sqcup \{f' \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \\ \Phi' = \Phi \dagger \{"a" \mapsto (\textsc{Write}, \Lambda)\} \end{array}$$

We evaluate each term given P_3 as assumption:

$$\begin{array}{ll} & f' = \max(\mathtt{dom}\ \varrho) + 1 \\ = & \quad \langle \text{val. of } \varrho \rangle \\ & f' = \max(\mathtt{dom}\ \theta) + 1 \\ = & \quad \langle \text{prop. } \mathtt{dom}, \max, \text{ arith.} \rangle \\ & f' = 1 \end{array}$$

$$\begin{array}{ll} & \varrho' = \varrho \sqcup \{f' \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \\ = & \quad \langle \text{val. of } \varrho \rangle \\ & \varrho' = \theta \sqcup \{f' \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \\ = & \quad \langle \text{extend, val. of } f' \rangle \\ & \varrho' = \{1 \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \end{array}$$

$$\begin{array}{ll} & \Phi' = \Phi \dagger \{"a" \mapsto (\textsc{Write}, \Lambda)\} \\ = & \quad \langle \text{val. of } \Phi \rangle \\ & \Phi' = (\Phi_0 \sqcup \{"a" \mapsto (\textsc{Closed}, J : \_)\}) \dagger \{"a" \mapsto (\textsc{Write}, \Lambda)\} \\ = & \quad \langle \text{prop. of } \sqcup, \dagger \rangle \\ & \Phi' = \Phi_0 \sqcup \{"a" \mapsto (\textsc{Write}, \Lambda)\} \end{array}$$

The postcondition becomes:

$$\begin{array}{l} f' = 1 \\ \varrho' = \{1 \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \\ \Phi' = \Phi_0 \sqcup \{"a" \mapsto (\textsc{Write}, \Lambda)\} \end{array}$$

We merge this with P_3, dropping primes, to get P_4:

$$P_4 \equiv \begin{cases} \mathcal{W} = (\Phi_0 \sqcup \{"a" \mapsto (\textsc{Write}, \Lambda)\}, \_) \\ \varrho = \{1 \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \\ f = 1 \\ x = J \end{cases}$$

9.1.6 C Statement 5

$$s_5 : \mathtt{fwritei(f,x*x)}$$

We show that P_4 implies the pre-condition for this instance of `fwritei`, which is

$$f \in \varrho \wedge \varrho(f) = (\text{HW}\textsc{rite}\ n\ \delta)$$

Assuming P_4, we show the pre-condition is satisfied:

$$\begin{array}{ll} & f \in \mathtt{dom}\ \varrho \wedge \varrho(f) = (\text{HW}\textsc{rite}\ n\ \delta) \\ = & \quad \langle \text{val. of } f, \varrho \rangle \\ & 1 \in \mathtt{dom}\{1 \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\} \wedge \{1 \mapsto (\textsc{HWrite}\ "a"\ \Lambda)\}(1) = (\text{HW}\textsc{rite}\ n\ \delta) \end{array}$$

$$
\begin{aligned}
= &\quad \langle \text{def. of } \mathtt{dom} \rangle \\
& \textsc{True} \wedge \{1 \mapsto (\textsc{Hwrite}\ ''a''\ \Lambda)\}(1) = (\text{HW}\textsc{rite}\ n\ \delta) \\
= &\quad \langle \text{prop. calc., map app.} \rangle \\
& (\textsc{Hwrite}\ ''a''\ \Lambda) = (\text{HW}\textsc{rite}\ n\ \delta) \\
= &\quad \langle \text{eq.} \rangle \\
& n = ''a'' \wedge \delta = \Lambda
\end{aligned}
$$

The pre-condition holds true under the resulting binding.

The post-condition of `fwritei` with substitutions is:

$$\varrho' = \varrho \dagger \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \Lambda \frown \langle x^2 \rangle)\}$$

We evaluate this given P_4 as assumption:

$$
\begin{aligned}
& \varrho' = \varrho \dagger \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \Lambda \frown \langle x^2 \rangle)\} \\
= &\quad \langle \text{def. of } \frown \rangle \\
& \varrho' = \{1 \mapsto (\textsc{Hwrite}\ ''a''\ \Lambda)\} \dagger \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle x^2 \rangle)\} \\
= &\quad \langle \text{val. of } \varrho \rangle \\
& \varrho' = \{1 \mapsto (\textsc{Hwrite}\ ''a''\ \Lambda)\} \dagger \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle x^2 \rangle)\} \\
= &\quad \langle \text{prop. of } \dagger \rangle \\
& \varrho' = \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle x^2 \rangle)\} \\
= &\quad \langle \text{val. of } x \rangle \\
& \varrho' = \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle J^2 \rangle)\}
\end{aligned}
$$

The postcondition becomes:

$$\varrho' = \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle J^2 \rangle)\}$$

We merge this with P_4, dropping primes, to get P_5:

$$
P_5 \equiv \begin{cases}
\mathcal{W} = (\Phi_0 \sqcup \{''a'' \mapsto (\text{W}\textsc{rite}, \Lambda)\}, \_) \\
\varrho = \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle J^2 \rangle)\} \\
f = 1 \\
x = J
\end{cases}
$$

9.1.7 C Statement 6

$$s_6 : \mathtt{fclose(f)}$$

We show that P_5 implies the pre-condition for this instance of `fclose`, which is

$$f \in \mathtt{dom}\ \varrho \wedge \varrho(f) = (\text{HW}\textsc{rite}\ n\ \delta) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\text{W}\textsc{rite}, \_)$$

Assuming P_5, we show it is satisfied:

$$
\begin{aligned}
& f \in \mathtt{dom}\ \varrho \wedge \varrho(f) = (\text{HW}\textsc{rite}\ n\ \delta) \wedge n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\text{W}\textsc{rite}, \_) \\
= &\quad \langle \text{val. of } f, \varrho \rangle \\
& 1 \in \mathtt{dom}\{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle J^2 \rangle)\} \wedge \{1 \mapsto (\text{HW}\textsc{rite}\ ''a''\ \langle J^2 \rangle)\}(1) = (\text{HW}\textsc{rite}\ n\ \delta) \\
& \wedge\ n \in \mathtt{dom}\ \Phi \wedge \Phi(n) = (\text{W}\textsc{rite}, \_) \\
= &\quad \langle \text{prop. of } \mathtt{dom}\text{, map app.} \rangle \\
& \textsc{True} \wedge (\text{HW}\textsc{rite}\ ''a''\ \langle J^2 \rangle) = (\text{HW}\textsc{rite}\ n\ \delta)
\end{aligned}
$$

$$
\begin{array}{rl}
& \wedge\, n \in \mathtt{dom}\,\Phi \wedge \Phi(n) = (\textsc{Write},\_) \\
= & \quad \langle \text{prop. calc., eq.} \rangle \\
& n = \text{"}a\text{"} \wedge \delta = \langle J^2 \rangle \\
& \wedge\, n \in \mathtt{dom}\,\Phi \wedge \Phi(n) = (\textsc{Write},\_) \\
= & \quad \langle \text{subs.} \rangle \\
& n = \text{"}a\text{"} \wedge \delta = \langle J^2 \rangle \wedge \text{"}a\text{"} \in \mathtt{dom}\,\Phi \wedge \Phi(\text{"}a\text{"}) = (\textsc{Write},\_) \\
= & \quad \langle \text{Lemma C.1} \rangle \\
& n = \text{"}a\text{"} \wedge \delta = \langle J^2 \rangle \wedge \textsc{True} \wedge \Phi(\text{"}a\text{"}) = (\textsc{Write},\_) \\
= & \quad \langle \text{prop. calc., Lemma C.2} \rangle \\
& n = \text{"}a\text{"} \wedge \delta = \langle J^2 \rangle \wedge (\textsc{Write}, \Lambda) = (\textsc{Write},\_) \\
= & \quad \langle \text{eq.} \rangle \\
& n = \text{"}a\text{"} \wedge \delta = \langle J^2 \rangle \wedge \textsc{True} \\
= & \quad \langle \text{prop. calc.} \rangle \\
& n = \text{"}a\text{"} \wedge \delta = \langle J^2 \rangle
\end{array}
$$

Precondtion holds subject to these substitutions.

The post-condition of `fclose` with substitutions is:

$$
\begin{array}{l}
\varrho' = \lhd[1]\varrho \\
\Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}
\end{array}
$$

We evaluate each term given P_5 as assumption:

$$
\begin{array}{rl}
& \varrho' = \lhd[1]\varrho \\
= & \quad \langle \text{val. of } \varrho \rangle \\
& \varrho' = \lhd[1]\{1 \mapsto \_\} \\
= & \quad \langle \text{defn. of } \lhd \rangle \\
& \varrho' = \theta
\end{array}
$$

$$
\begin{array}{rl}
& \Phi' = \Phi \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\} \\
= & \quad \langle \text{val. of } \Phi \rangle \\
& \Phi' = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Write}, \Lambda)\}) \dagger \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\} \\
= & \quad \langle \text{map props.} \rangle \\
& \Phi' = \Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}
\end{array}
$$

The postcondition becomes:

$$
\begin{array}{l}
\varrho' = \theta \\
\Phi' = \Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}
\end{array}
$$

We merge this with P_5, dropping primes, to get P_6:

$$
P_6 \equiv \left\{
\begin{array}{l}
\mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}, \_) \\
\varrho = \theta \\
f = 1 \\
x = J
\end{array}
\right.
$$

9.1.8 Finishing the Proof

The annotated program is:

$$
\left\{
\begin{array}{l}
\mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}, \_) \\
\varrho = \theta
\end{array}
\right\}
$$

```
f = fopen("a",FRead)
```

$$\left\{\begin{array}{l} \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \_)\}, \_) \\ \varrho = \{1 \mapsto (\textsc{HRead}\ \text{"}a\text{"}\ \Lambda\ J : \_)\} \\ f = 1 \end{array}\right\}$$

```
x = freadi(f)
```

$$\left\{\begin{array}{l} \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Read}\ 1, J : \_)\}, \_) \\ \varrho = \{1 \mapsto (\text{HR}\textsc{ead}\ \text{"}a\text{"}\ \langle J \rangle\ \_)\} \\ f = 1 \\ x = J \end{array}\right\}$$

```
fclose(f)
```

$$\left\{\begin{array}{l} \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, J : \_)\}, \_) \\ \varrho = \theta \\ f = 1 \\ x = J \end{array}\right\}$$

```
f = fopen("a",FWrite)
```

$$\left\{\begin{array}{l} \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Write}, \Lambda)\}, \_) \\ \varrho = \{1 \mapsto (\textsc{HWrite}\ \text{"}a\text{"}\ \Lambda)\} \\ f = 1 \\ x = J \end{array}\right\}$$

```
fwritei(f,x*x)
```

$$\left\{\begin{array}{l} \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Write}, \Lambda)\}, \_) \\ \varrho = \{1 \mapsto (\text{HW}\textsc{rite}\ \text{"}a\text{"}\ \langle J^2 \rangle)\} \\ f = 1 \\ x = J \end{array}\right\}$$

```
fclose(f)
```

$$\left\{\begin{array}{l} \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}, \_) \\ \varrho = \theta \\ f = 1 \\ x = J \end{array}\right\}$$

which must imply

$$\{\ \mathcal{W} = (\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto (\textsc{Closed}, \langle J^2 \rangle)\}), \_)\ \}$$

This is vacuosly the case ♣

9.1.9 Lemma C.1

Given

$$\Phi = \Phi_0 \sqcup \{\text{"}a\text{"} \mapsto \_\}$$

show

$$\text{"}a\text{"} \in \mathtt{dom}\ \Phi = \textsc{True}$$

$$\begin{array}{cl} & \text{"}a\text{"} \in \mathtt{dom}\ \Phi \\ = & \quad \langle \text{val. of } \Phi \rangle \\ & \text{"}a\text{"} \in \mathtt{dom}(\Phi_0 \sqcup \{\text{"}a\text{"} \mapsto \_\}) \\ = & \quad \langle \text{defn. of } \mathtt{dom} \rangle \\ & \text{"}a\text{"} \in (\mathtt{dom}\ \Phi_0 \sqcup \{\text{"}a\text{"}\}) \\ = & \quad \langle \text{set theory} \rangle \\ & True \end{array}$$

9.1.10 Lemma C.2

Given

$$\Phi = \Phi_0 \sqcup \{"a" \mapsto f_s\}$$

show

$$\Phi("a") = f_s$$

$$
\begin{array}{ll}
 & \Phi("a") \\
= & \quad \langle \text{val. of } \Phi \rangle \\
 & (\Phi_0 \sqcup \{"a" \mapsto f_s\})("a") \\
= & \quad \langle \text{defn. of application} \rangle \\
 & f_s
\end{array}
$$

9.2 Clean Language Proof

We wish to show that

```
lookup phi’ "a" = (Closed,[J*J])
where
  (phi’,_) = main (extend phi "a" (Closed,J:_),_)
```

The program can be re-written, using the Hash Syntactic Sugar rule as follows

```
main =\ w -> h1
h1 = letb (f,w)=fopen "a" FRead w in h2
h2 = letb (i,f) = freadi f in h3
h3 = letb w = fclose f w in h4
h4 = letb (f,w)=fopen "a" Fwrite w in h5
h5 = letb f =fwritei (x*x) f in h6
h6 = letb w = fclose f w in w
```

`main (extend phi "a" (Closed,J:_),w)`

= ⟨ defn. of `main` ⟩

```
(\w->h1) (extend phi "a" (Closed,J:_),_)
```

= ⟨ shorthand `h1` ⟩

```
(\w->letb (f,w)=fopen "a" FRead w in h2)
(extend phi "a" (Closed,J:_),_)
```

= ⟨ β-reduction ⟩

```
letb (f,w)
     =fopen "a" FRead (extend phi "a" (Closed,J:_),_)
in h2
```

= ⟨ Lemma K.1 ⟩

```
letb (f,w)=(Hread "a" [] J:_ ,(override phi "a" (Read 1,J:_),_))
in h2
```

= ⟨ expand `h2` ⟩

```
letb (f,w)=(Hread "a" [] J:_ ,(override phi "a" (Read 1,J:_),_))
in letb (x,f) = freadi f in h3
```

= ⟨ partial let evalution on `f` ⟩

```
letb w = (override phi "a" (Read 1,J:_),_)
in letb (x,f) = freadi (Hread "a" [] J:_) in h3
```
= ⟨ Lemma K.4, defn of `freadi` ⟩
```
letb w = (override phi "a" (Read 1,J:_),_)
in letb (x,f) = (J,Hread "a" [J] _) in h3
```
= ⟨ expand `h3` ⟩
```
letb w = (override phi "a" (Read 1,J:_),_)
in letb (x,f) = (J,Hread "a" [J] _)
in letb w = fclose f w in h4
```
= ⟨ Let Evaluation ⟩
```
letb (x,f) = (J,Hread "a" [J] _)
in letb w = fclose f (override phi "a" (Read 1,J:_),_) in h4
```
= ⟨ Partial Let Evaluation ⟩
```
letb x = J in
letb w = fclose (Hread "a" [J] _) (override phi "a" (Read 1,J:_),_)
in h4
```
= ⟨ Lemma K.2 ⟩
```
letb x = J in
letb w = (override phi "a" (Closed,J:_),_)
in h4
```
= ⟨ expand `h4` ⟩
```
letb x = J in
letb w = (override phi "a" (Closed,J:_),_)
in letb (f,w)=fopen "a" Fwrite w in h5
```
= ⟨ Let Evaluation ⟩
```
letb x = J in
in letb (f,w)=fopen "a" Fwrite (override phi "a" (Closed,J:_),_) in h5
```
= ⟨ Lemma K.3 ⟩
```
letb x = J in
in letb (f,w)=(Hwrite "a" [], ((override phi "a" (Write,[])),_)) in h5
```
= ⟨ expand `h5` ⟩
```
letb x = J in
in letb (f,w)=(Hwrite "a" [], ((override phi "a" (Write,[])),_))
in letb f =fwritei (x*x) f in h6
```
= ⟨ Let Evaluation ⟩
```
letb (f,w)=(Hwrite "a" [], ((override phi "a" (Write,[])),_))
in letb f =fwritei (J*J) f in h6
```
= ⟨ Partial Let Evaluation ⟩
```
letb w=((override phi "a" (Write,[])),_)
in letb f =fwritei (J*J) (Hwrite "a" []) in h6
```
= ⟨ Lemma K.5, defn. of `fwritei` ⟩
```
letb w=((override phi "a" (Write,[])),_)
in letb f = Hwrite "a" [J*J] in h6
```
= ⟨ expand `h6` ⟩
```
letb w=((override phi "a" (Write,[])),_)
in letb f = Hwrite "a" [J*J]
in letb w = fclose f w in w
```

= ⟨ Let Evaluation (f) ⟩
```
letb w=((override phi "a" (Write,[])),_)
in letb w = fclose (Hwrite "a" [J*J]) w in w
```
= ⟨ Let Evaluation (w) ⟩
```
letb w = fclose (Hwrite "a" [J*J]) ((override phi "a" (Write,[])),_)
in w
```
= ⟨ Lemma K.6, defn. `fclose` ⟩
```
letb w = (override (override phi "a" (Write,[])) "a" (Closed,[J*J]),_)
in w
```
= ⟨ prop. of override ⟩
```
letb w = (override phi "a" (Closed,[J*J]),_)
in w
```
= ⟨ Let Evaluation ⟩
```
(override phi "a" (Closed,[J*J]),_)
```

We have shown that

```
main (extend phi "a" (Closed,J:_),w)
=
(override phi "a" (Closed,[J*J]),_)
```

Now we evaluate our property:

```
lookup phi' "a"
where
  (phi',_) = main (extend phi "a" (Closed,J:_),_)
```
= ⟨ just demonstrated ⟩
```
lookup phi' "a"
where
  (phi',_) = (override phi "a" (Closed,[J*J]),_)
```
= ⟨ where clause ⟩
```
lookup (override phi "a" (Closed,[J*J])) "a"
```
= ⟨ lookup ⟩
```
(Closed,[J*J])
```

Proof is complete ♣

9.2.1 Lemma K.1

```
fopen "a" FRead (extend phi "a" (Closed,J:_),_)
```
= ⟨ Lemma K.1.1, defn. of `fopen` ⟩
```
(h,(override (extend phi "a" (Closed,J:_),_) "a" f,_))
where
  h = Hread "a" [] ds
  f = (Read r,ds)
  f0 = lookup phi n
  r = if fst f0 == Closed then 1 else _
  ds = snd f0
```

= ⟨ prop. of override and extend ⟩
```
(h,(override phi "a" f,_))
where
  h = Hread "a" [] ds
  f = (Read r,ds)
  f0 = lookup phi n
  r = if fst f0 == Closed then 1 else _
  ds = snd f0
```
= ⟨ eval `f0` and subs. ⟩
```
(h,(override phi "a" f,_))
where
  h = Hread "a" [] ds
  f = (Read r,ds)
  r = if fst (Closed,J:_) == Closed then 1 else _
  ds = snd (Closed,J:_)
```
= ⟨ eval `fst`,`snd` and subs. ⟩
```
(h,(override phi "a" f,_))
where
  h = Hread "a" [] ds
  f = (Read r,ds)
  r = if Closed == Closed then 1 else _)
  ds = J:_
```
= ⟨ eval `ds`,`snd`, cond. and subs. ⟩
```
(h,(override phi "a" f,_))
where
  h = Hread "a" [] J:_
  f = (Read 1,J:_)
```
= ⟨ subs for `h`,`f` ⟩
```
(Hread "a" [] J:_ ,(override phi "a" (Read 1,J:_),_))
```

9.2.2 Lemma K.1.1

```
pre_fopen "a" FRead (extend phi "a" (Closed,J:_),_)
```
= ⟨ defn of `pre_fopen` ⟩
```
if (member("a",dom (extend phi "a" (Closed,J:_)))
   (fst(lookup (extend phi "a" (Closed,J:_)) "a")==Closed)
   True
```
= ⟨ defn of `dom` ⟩
```
if (member("a",dom phi union {"a"}))
   (fst(lookup (extend phi "a" (Closed,J:_)) "a")==Closed)
   True
```
= ⟨ prop. of member ⟩
```
if True
   (fst(lookup (extend phi "a" (Closed,J:_)) "a")==Closed)
   True
```
= ⟨ cond. ⟩
```
fst(lookup (extend phi "a" (Closed,J:_)) "a")==Closed
```
= ⟨ defn. lookup. ⟩
```
fst(Closed,J:_)==Closed
```
= ⟨ defn. fst, eq. ⟩
```
True
```

9.2.3 Lemma K.2

```
 fclose (Hread "a" [J] _) (override phi "a" (Read 1,J:_),_)
=  ⟨ Lemma K.2.1, defn of fclose ⟩
(override (override phi "a" (Read 1,J:_)) "a" (s,ds),_)
where
  (Read r,ds) = lookup (override phi "a" (Read 1,J:_)) "a"
  s = if r == 1 then Closed else (Read (s-1))
=  ⟨ prop. of override ⟩
(override phi "a" (s,ds),_)
  where
  (Read r,ds) = lookup (override phi "a" (Read 1,J:_)) "a"
  s = if r == 1 then Closed else (Read (s-1))
=  ⟨ lookup and override ⟩
(override phi "a" (s,ds),_)
  where
  (Read r,ds) = (Read 1,J:_)
  s = if r == 1 then Closed else (Read (s-1))
=  ⟨ where clause ⟩
(override phi "a" (s,J:_),_)
  where
  s = if 1 == 1 then Closed else (Read (s-1))
=  ⟨ cond., where clause ⟩
(override phi "a" (Closed,J:_),_)
```

9.2.4 Lemma K.2.1

```
 pre_ fclose (Hread "a" [J] _) (override phi "a" (Read 1,J:_),_)
=  ⟨ def. pre_fclose ⟩
 member("a",dom (override phi "a" (Read 1,J:_)))
 && fst(lookup (override phi "a" (Read 1,J:_)) "a")=(Read _)
=  ⟨ prop. dom, member, lookup ⟩
 True && fst(Read 1,J:_)=(Read _)
=  ⟨ prop. calc., defn. fst, eq. ⟩
 True
```

9.2.5 Lemma K.3

```
 fopen "a" Fwrite (override phi "a" (Closed,J:_),_)
=  ⟨ Lemma K.3.1, defn. of fopen ⟩
 (h,(override (override phi "a" (Closed,J:_)) "a" f),_))
    where
      h = Hwrite "a" []
      f = (Write,[])
=  ⟨ prop. of override ⟩
 (h,((override phi "a" f),_))
    where
      h = Hwrite "a" []
      f = (Write,[])
=  ⟨ where clause ⟩
 (Hwrite "a" [], ((override phi "a" (Write,[])),_))
```

9.2.6 Lemma K.3.1

```
pre_fopen "a" Fwrite (override phi "a" (Closed,J:_),_)
```
= ⟨ defn. ⟩
```
if (member("a",dom(override phi "a" (Closed,J:_))))
   (fst(lookup (override phi "a" (Closed,J:_)) "a")==Closed)
   True
```
= ⟨ prop. member and dom ⟩
```
if True
   (fst(lookup (override phi "a" (Closed,J:_)) "a")==Closed)
   True
```
= ⟨ cond. ⟩
```
fst(lookup (override phi "a" (Closed,J:_)) "a")==Closed
```
= ⟨ lookup ⟩
```
fst(Closed,J:_)==Closed
```
= ⟨ fst, eq. ⟩
```
True
```

9.2.7 Lemma K.4

```
pre_freadi (Hread "a" [] J:_)
```
= ⟨ defn. ⟩
```
(J:_) != []
```
= ⟨ lst eq. ⟩
```
True
```

9.2.8 Lemma K.5

```
pre_fwritei (J*J) (Hwrite "a" [])
```
= ⟨ defn. ⟩
```
True
```

9.2.9 Lemma K.6

```
pre_fclose (Hwrite "a" [J*J]) ((override phi "a" (Write,[])),_)
```
= ⟨ defn. `pre_fclose` ⟩
```
member("a",dom (override phi "a" (Write,[])))
&& fst(lookup (override phi "a" (Write,[])) "a")==Write
```
= ⟨ defn. `dom`, defn. `lookup` ⟩
```
member("a",(dom phi `union` {"a"} )) && fst(Write,[])==Write
```
= ⟨ prop. `member`, defn. `fst` ⟩
```
True && Write==Write
```
= ⟨ eq., prop. calc ⟩
```
True
```

9.3 Haskell Language Proof

We start with the program text, and transform it by effectively replacing the do-notation and monads by let expressions and lambda abtractions, in order to make the world explicit.

Converted to “let” form:

```
main = do
         h <- openFile "a" ReadMode
         x <- hreadi h
         hclose h
         h <- openFile "a" WriteMode
         hwritei h (x*x)
         hclose h
```
= ⟨ `do` desugaring ⟩
```
main = openFile "a" ReadMode >>= \h ->
       hreadi h >>= \x ->
       hclose h >>
       h <- openFile "a" WriteMode >>= \h ->
       hwritei h (x*x) >>
       hclose h
```
= ⟨ `bind` and `seq` elimination ⟩
```
main = h1
   h1 = \w -> letb (h,w') = openFile "a" ReadMode w in h2 w'
   h2 = \w -> letb (x,w') = hreadi h w in h3 w'
   h3 = \w -> letb w'     = hclose h w in h4 w'
   h4 = \w -> letb (h,w') = openFile "a" WriteMode w in h5 w'
   h5 = \w -> letb w'     = hwritei h (x*x) w in h6
   h6 = \w -> hclose h w
```

Given this definition of `main` we wish to show that

```
  lookup phi' "a" = (Closed,[J*J])
  where ((phi',_),_) = main ((extend phi "a" (Closed,J:_),_),_)
```

Beginning with the evaluation of `main`

```
main ((extend phi "a" (Closed,J:_),W),[])
```
= ⟨ Definition of `main` ⟩
```
h1 ((extend phi "a" (Closed,J:_),W),[])
```
= ⟨ expansion of `h1` ⟩
```
 \w -> letb (h,w') = openFile "a" ReadMode w
       in h2 w' ((extend phi "a" (Closed,J:_),W),[])
```
= ⟨ β-reduction ⟩
```
  letb (h,w') = openFile "a" ReadMode
                  ((extend phi "a" (Closed,J:_),W),[]) in h2 w'
```
= ⟨ Lemma H.1 ⟩
```
  letb (h,w') = (1, ((override phi "a" (Read 1,J:_),W),
                    override [] 1 (Hread "a" [] (J:_)))) in h2 w'
```

= ⟨ Partial let evaluation ⟩

```
  letb h = 1 in h2
    ((override phi "a" (Read 1,J:_),W), override [] 1 (Hread "a" [] (J:_)))
```

= ⟨ expansion of `h2` ⟩

```
letb h = 1 in
  \w -> let (x,w') = hreadi h w in h3 w'
    ((override phi "a" (Read 1,J:_),W), override [] 1 (Hread "a" [] (J:_)))
```

= ⟨ β-reduction ⟩

```
let h = 1 in
  letb (x,w') = hreadi h ((override phi "a" (Read 1,J:_),W),
                          override 1 [] (Hread "a" [] (J:_)))
  in h3 w'
```

= ⟨ Lemma H.2 ⟩

```
  letb h = 1 in letb (x,w') = (J, (override phi "a" (Read 1,J:_),W),
                                  override [] 1 (Hread "a" [J] _))
                in h3 w'
```

= ⟨ partial Let evaluation ⟩

```
  letb h = 1 in letb x = J
  in h3 ((override phi "a" (Read 1,J:_),W),
          override [] 1 (Hread "a" [J] _))
```

= ⟨ expansion of `h3` ⟩

```
  letb h = 1 in letb x = J in \w -> let w' = hclose h w in h4 w'
              ((override phi "a" (Read 1,J:_),W),
                override [] 1 (Hread "a" [J] _))
```

= ⟨ β-reduction ⟩

```
  letb h = 1 in letb x = J in
  letb w' = hclose h ((override phi "a" (Read 1,J:_),W),
                      override [] 1 (Hread "a" [J] _)) in h4 w'
```

= ⟨ Lemma H.3 ⟩

```
  letb h = 1 in let x = J in
  letb w' = ((override phi "a" (Closed,J:_),W),[]) in h4 w'
```

= ⟨ let evaluation on `w'` ⟩

```
  letb h = 1 in letb x = J in h4 ((override phi "a" (Closed,J:_),W),[])
```

= ⟨ expansion of `h4` ⟩

```
  letb h = 1 in letb x = J in
      \w -> letb (h,w') = openFile "a" WriteMode w
            in h5 w' ((override phi "a" (Closed,J:_),W),[])
```

= ⟨ β-reduction ⟩

```
  letb h = 1 in letb x = J in
  letb (h,w') = openFile "a" WriteMode ((override phi "a" (Closed,J:_),W),[])
    in h5 w'
```

= ⟨ Lemma H.4 ⟩

```
  letb h = 1 in letb x = J in letb (h,w') =
    (1, ( (override phi "a" (Write,[]),W), override [] 1 (Hwrite "a" [])))
    in h5 w'
```

= ⟨ Partial Let evaluation ⟩

```
  letb h = 1 in letb x = J in letb h = 1 in
    h5 ((override phi "a" (Write,[]),W), override [] 1 (Hwrite "a" []))
```
$=$ ⟨ Let evaluation ⟩
```
  letb x = J in letb h = 1 in
    h5 ((override phi "a" (Write,[]),W), override [] 1 (Hwrite "a" []))
```
$=$ ⟨ expansion of `h5` ⟩
```
  letb x = J in letb h = 1 in
    \w -> letb w' = hwritei h (x*x) w in h6 ((override phi "a"
       (Write,[]),W), override [] 1 (Hwrite "a" []))
```
$=$ ⟨ β-reduction ⟩
```
  letb x = J in letb h = 1 in
    letb w' = hwritei h (x*x) ((override phi "a"
       (Write,[]),W), override [] 1 (Hwrite "a" [])) in h6 w'
```
$=$ ⟨ Let evaluation on `x` ⟩
```
  letb h = 1 in letb w' =
    hwritei h (J*J) ((override phi "a"
       (Write,[]),W), override [] 1 (Hwrite "a" [])) in h6 w'
```
$=$ ⟨ Substitution for `h` ⟩
```
  letb h = 1 in letb w' =
    hwritei 1 (J*J) ((override phi "a"
       (Write,[]),W), override [] 1 (Hwrite "a" [])) in h6 w'
```
$=$ ⟨ Lemma H.5 ⟩
```
  letb h = 1 in let w' =
    ((override phi "a" (Write,[]),W),
      override [] 1 (Hwrite "a" [J*J])) in h6 w'
```
$=$ ⟨ Let evaluation on `w'` ⟩
```
  letb h = 1 in
    h6 ((override phi "a" (Write,[]),W),
         override [] 1 (Hwrite "a" [J*J]))
```
$=$ ⟨ expansion of h6 ⟩
```
  letb h = 1 in
    \w -> hclose h w ((override phi "a" (Write,[]),W),
                       override [] 1 (Hwrite "a" [J*J]))
```
$=$ ⟨ Let reduction on `h` and definition of `hclose` ⟩
```
  (override phi "a" (Closed, [J*J]), [])
```

So we have shown that

```
main ((extend phi "a" (Closed,J:_),W),[])
```

$=$

```
  (override phi "a" (Closed, [J*J]) W, [])
```

As for the clean proof, we can now use lookup to establish the property.

10 Lemmas for Haskell proof

10.1 Lemma H.1

```
  openFile "a" ReadMode ((extend phi "a" (Closed,J:_),W),[])
```

= ⟨ definition of `openFile` ⟩

```
  \(w,l) -> (h,(w',override l h fs)) ((extend phi "a" (Closed,J:_),W),[])
            where (fs,w') = fopen "a" ReadMode w
                  h       = nextint l
```

= ⟨ `nextint` of l; `where` substitution; ReadMode conversion ⟩

```
  \(w,l) -> (1,(w', override l 1 fs)) ((extend phi "a" (Closed,J:_),W),[])
            where (fs, w') = fopen "a" Fread w
```

= ⟨ β-reduction ⟩

```
  (1,(w', override [] 1 fs))
    where (fs, w') = fopen "a" Fread (extend phi "a" (Closed,J:_),W)
```

= ⟨ Lemma K.1 ⟩

```
  (1,(w', override [] 1 fs))
    where (fs, w') = ( hRead "a" [] (J:_), override phi "a" (Read 1,J:_),W)
```

= ⟨ `where` substitution ⟩

```
  (1,((override phi "a" (Read 1,J:_),W),
       override [] 1 (hRead "a" [] (J:_))))
```

10.1.1 Lemma H.2

```
hreadi 1 ((override phi "a" (Read 1,J:_),W),
           override [] 1 (hRead "a" [] (J:_)))
```

= ⟨ definition of `hreadi` and β-reduction ⟩

```
  (the_int, ((override phi "a" (Read 1,J:_),W), override (1,fs') []))
          where (the_int,fs') = freadi fs
          fs = lookup 1 (override [] 1 (Hread "a" [] (J:_)))
```

= ⟨ definition of `lookup` ⟩

```
 (the_int, ((override phi "a" (Read 1,J:_),W), override [] 1 fs'))
          where (the_int,fs') = freadi fs
          fs = (Hread "a" [] (J:_))
```

= ⟨ `where` substitution on `fs` ⟩

```
 (the_int, ((override phi "a" (Read 1,J:_),W), override [] 1 fs'))
          where (the_int,fs') = freadi (Hread "a" [] (J:_))
```

= ⟨ definition of `freadi` ⟩

```
 (the_int, ((override phi "a" (Read 1,J:_),W), override [] 1 fs'))
          where (the_int,fs') = (J, HRead "a" [J] _)
```

= ⟨ `where` substitution on `freadi` ⟩

```
 (J, ((override phi "a" (Read 1,J:_),W), override [] 1 (HRead "a" [J] _)))
```

10.1.2 Lemma H.3

```
hclose 1 ((override phi "a" (Read 1,J:_),W),
           override [] 1 (Hread "a" [J] _))
```

= ⟨ Definition of `hclose` ⟩

```
\(w,l) -> (w', remove 1 l)
             ((override phi "a" (Read 1,J:_),W),
               override [] 1 (Hread "a" [J] _))
                     where w' = fclose fs w
                          fs  = lookup 1 l
```

= ⟨ β-reduction ⟩

```
  (w', remove 1 (override [] 1 (Hread "a" [J] _)))
  where w' = fclose fs (override phi "a" (Read 1,J:_),W)
        fs  = lookup 1 (override [] 1 (Hread "a" [J] _))
```

= ⟨ `lookup` and `where` substitution of `fs` ⟩

```
  (w', remove 1 (override [] 1 (Hread "a" [J] _)))
  where w' = fclose (Hread "a" [J] _) (override phi "a" (Read 1,J:_),W)
```

= ⟨ `remove` ⟩

```
  (w', [])
  where w' = fclose (Hread "a" [J] _) (override phi "a" (Read 1,J:_),W)
```

= ⟨ definition of `fclose` ⟩

```
  (w', [])
  where w' = (override phi "a" (Closed,J:_),W)
```

= ⟨ `where` substitution ⟩

```
  ((override phi "a" (Closed,J:_),W), [])
```

10.1.3 Lemma H.4

```
openFile "a" WriteMode ((override phi "a" (Closed,J:_),W),[])
```

= ⟨ definition of openFile; WriteMode substitution ⟩

```
\(w,l) -> (h, (w',override l h fs)) ((override phi "a" (Closed,J:_),W),[])
         where   (fs,w') = fopen "a" Fwrite w
                 h       = nextint l
```

= ⟨ β-reduction ⟩

```
  (h, (w',override [] h fs))
        where   (fs,w') = fopen "a" Fwrite (override phi "a" (Closed,J:_),W)
                h       = nextint []
```

= ⟨ `nextint` and `where` substitution of `h` ⟩

```
  (1, (w',override [] 1 fs))
      where (fs,w') = fopen "a" Fwrite (override phi "a" (Closed,J:_),W)
```

= ⟨ Lemma K.3 ⟩

```
  (1, (w',override [] 1 fs))
               where   (fs,w') = (Hwrite "a" [], ((override phi "a" (Write,[])),W))
```

= ⟨ `where` substitution ⟩

```
  (1, (((override phi "a" (Write,[])),W),override [] 1 (Hwrite "a" [])))
```

10.1.4 Lemma H.5

```
hwritei 1 (J*J) ((override phi "a" (Write,[]),W), override [] 1 (Hwrite "a" []))
```

= ⟨ definition of `hwritei` and argument substitution ⟩

```
\(w,l) -> (w, override l 1 fs') ((override phi "a" (Write,[]),W),
                                    override [] 1 (Hwrite "a" []))
   where fs' = fwritei (J*J) fs
         fs  = lookup 1 l
```

= ⟨ β-reduction ⟩

```
((override phi "a" (Write,[]),W), override (override [] 1 (Hwrite "a" [])) 1 fs')
   where fs' = fwritei (J*J) fs
         fs  = lookup 1 (override [] 1 (Hwrite "a" []))
```

= ⟨ lookup and where substitution of fs ⟩

```
((override phi "a" (Write,[]),W), override (override [] 1 (Hwrite "a" [])) 1 fs')
   where fs' = fwritei (J*J) (Hwrite "a" [])
```

= ⟨ definition of `fwritei` ⟩

```
((override phi "a" (Write,[]),W), override (override [] 1 (Hwrite "a" [])) 1 fs')
   where fs' = (Hwrite "a" [J*J])
```

= ⟨ `where` substitution of `fs'` ⟩

```
((override phi "a" (Write,[]),W),
  override (override [] 1 (Hwrite "a" [])) 1 (Hwrite "a" [J*J]))
```

= ⟨ override ⟩

```
((override phi "a" (Write,[]),W),
  override [] 1 (Hwrite "a" [J*J]))
```

References

[Bd87] Richard Bird and Oege de Moor. *Algebra of Programming*. Series in Computer Science. Prentice Hall International, London, 1987.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, second edition edition, 1998.

[BJLM91] J.-P. Banâtre, S.B. Jones, and D. Le Métayer. *Prospects for Functional Programming in Software Engineering*, volume 1 of *ESPRIT Research Reports, Project 302*. Springer-Verlag, Berlin, 1991.

[BS00] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 00:pp1–36, 2000. URL: `ftp: //ftp.cs.kun.nl /pub/Clean/papers/1996/bare96-uniclosed.ps.gz`.

[BS01] Andrew Butterfield and Glenn Strong. Comparing I/O Proofs in Three Programming Paradigms: a Baseline. Technical Report TCD-CS-2001-31, Dublin University, Computer Science Department, Trinity College, Dublin, August 2001.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall International,, London, 1988.

[Dav92] A.J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge Computer Science Texts. Cambridge University Press, 1992.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/ Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Hen87] M.C. Henson. *Elements of Functional Languages*. Computer Science Texts. Blackwell Scientific Publications, 1987.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall International, London, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, 2nd edition, 1988.

[Mor94] Carroll Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, London, 2nd. edition, 1994.

[PH+99] Simon Peyton Jones, John Hughes, et al. Haskell 98. Technical report, `www.haskell.org`, 1999. URL: `http://www.haskell.org/onlinereport/`.

[PJ87] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice Hall International, London, 1987.

[PvE98] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean (version 1.3). Technical report, High Level Software Tools B.V. and University of Nijmegen, 1998. URL: `ftp://ftp.cs.kun.nl/pub/Clean/Clean13/doc/refman13.pdf.gz`.

[Sch88] David A. Schmidt. *Denotational Semantics*. William C. Brown, Dubuque, Iowa, 1988.