The Curio Language Specification and its Machine-Verified Confluence Proof

Malcolm Dowse and Andrew Butterfield*
Trinity College Dublin

July 24, 2005

Contents

*Supported by Enterprise Ireland Basic Research Grant SC-2002-283.

1 Introduction

We present the specification and a simple implementation of the language Curio, a small monadic concurrent functional language. This language was designed to give a semantics to concurrent I/O in pure functional languages by way of modelling the API directly. Central to Curio is the fact that program execution is deterministic if an I/O model obeys a pre-condition. We give the details of the machine-verified proof of this confluence result.

All theorems and lemmas have been fully machine verified with Sparkle [dMvEP01], a semi-automated LCF-style [Pau87] proof assistant specifically designed for reasoning about lazy functional languages. Our metalanguage is Core-Clean, a stripped down version of the functional language Clean [PvE01] with no support for ad-hoc polymorphism (type classes) or I/O. We use Haskell [PHWH03] syntax in this document since it will be more familiar to most readers and, anyway, the differences are slight. The one main difference is the presence of strictness annotation, but this can be emulated in Haskell using `($)` or `seq` and we skim over any references to it in this document.

The proof style is denotational, so we quantify and induct over elements of domains, not syntax. It also means that proofs tend to become cluttered with uninteresting but unavoidable results concerning $\perp$. The main confluence proof requires about 100 separate results, many of which relate solely to how $\perp$ propagates.

The semantics of a less sophisticated precursor to Curio can be found in [DBvE05], published in the proceedings of IFL 2004.

2 API Models and I/O Contexts

We begin by introducing our way of modelling I/O and the notion of an I/O contexts which is required to ensure determinism. These will both be used to give the semantics of a real language in Section 3.

2.1 Definition

One of the goals of our research is to describe the interaction of programs with its environment by way of modelling the behaviour of each individual action on a "world state".

The definition of an I/O model is as follows: in the metalanguage it is the following 4-tuple parameterised by five types:

$$\begin{aligned}\mathfrak{s} :: \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \quad \triangleq \quad & \langle \mathtt{af} :: \alpha \to \omega \to (\omega, \nu) \\ & , \mathtt{wa} :: \alpha \to \omega \to \mathsf{Bool} \\ & , \mathtt{ap} :: \varsigma \to \alpha \to \mathsf{Bool} \\ & , \mathtt{pf} :: \rho \to \varsigma \to (\varsigma, \varsigma) \rangle\end{aligned}$$

The five types take the following roles, all of which will be explained in more detail later: ν: return values from actions; α: actions; ρ: "parameters" to the splitting of contexts; ω: world state; ς: I/O contexts. As we shall see, these four components are enough to describe state-based I/O which permits both a `fork`-like concurrency primitive and communication.

When giving general properties, we always assume the existence of some arbitrary, implicit I/O model called $\mathfrak{s}$ which binds the five types ν, α, ρ, ω and ς, and the four functions `af`, `wa`, `ap` and `pf`. It is important to remember that the five types are domains with a $\bot$ element, and the four functions can be any arbitrary computable function whose results may be undefined $(\bot)$[1].

The four functions are now explained in detail.

The API – `af` and `wa`

The function $\mathtt{af} :: \alpha \to \omega \to (\omega, \nu)$ defines the state-transformer for each action. Elements of type α identify I/O actions which can be performed and for any action $a :: \alpha$, $\mathtt{af}\ a :: \omega \to (\omega, \nu)$ is the state-transformer for that action. This describes how the action changes the global state of type ω, and what return value of type ν it yields. Type ν is typically a sum-type capable of storing Ints, Bools, Chars or any other value that an action might need to return.

[1] This probably isn't ideal, especially for the world model, but it's not serious. Any set-based model can be turned into a domain-based one by (1) turning the sets into "flat" domains by adding a bottom element, (2) making the functions bottom-preserving, and (3) showing that the functions are indeed Turing-computable.

$$
\begin{array}{rcl}
|||_{\mathfrak{s}}, \mathsf{ally}_{\mathfrak{s}} & : & \alpha \to \alpha \to \mathbb{B} \\
a_l \; |||_{\mathfrak{s}} \; a_r & \triangleq & \forall_{w \in \omega}.\forall_{w_2 \in \omega}.\mathtt{wa}\ a_l\ w = \textsc{False} \wedge \mathtt{wa}\ a_r\ w = \textsc{False} \Longrightarrow \forall_{v_l \in \nu}.\forall_{v_r \in \nu}. \\
 & & \quad (\exists_{w_1 \in \omega}.\mathtt{af}\ a_l\ w = (w_1, v_l) \wedge \mathtt{af}\ a_r\ w_1 = (w_2, v_r)) \\
 & & \quad \Longleftrightarrow \\
 & & \quad (\exists_{w_1 \in \omega}.\mathtt{af}\ a_r\ w = (w_1, v_r) \wedge \mathtt{af}\ a_l\ w_1 = (w_2, v_l)) \\
\mathsf{ally}_{\mathfrak{s}}(a_l, a_r) & \triangleq & \forall_{w \in \omega}.\forall_{w_1 \in \omega}.\forall_{v \in \nu}.\mathtt{wa}\ a_l\ w = \textsc{False} \wedge \mathtt{af}\ a_l\ w = (w_1, v) \wedge \\
 & & \quad \neg(\mathtt{wa}\ a_r\ w = \textsc{True}) \Longrightarrow \mathtt{wa}\ a_r\ w\ =\ \mathtt{wa}\ a_r\ w_1
\end{array}
$$

$$
\begin{array}{rcl}
\sqsubseteq_{\mathfrak{s}}, \Diamond_{\mathfrak{s}} & : & \varsigma \to \varsigma \to \mathbb{B} \\
c_1 \sqsubseteq_{\mathfrak{s}} c_2 & \triangleq & \forall_{a \in \alpha}.\mathtt{ap}\ a\ c_1 \Longrightarrow \mathtt{ap}\ a\ c_2 \\
c_l \Diamond_{\mathfrak{s}} c_r & \triangleq & \forall_{a_l \in \alpha}.\forall_{a_r \in \alpha}.\mathtt{ap}\ c_l\ a_l \wedge \mathtt{ap}\ c_r\ a_r \Longrightarrow \\
 & & \quad a_l \; |||_{\mathfrak{s}} \; a_r \wedge \mathsf{ally}_{\mathfrak{s}}(a_l, a_r) \wedge \mathsf{ally}_{\mathfrak{s}}(a_r, a_l)
\end{array}
$$

Figure 1: Relations on actions and contexts

If communication is to be permitted then there must be occasions in which an action is waiting for something to occur and cannot proceed. The function $\mathtt{wa} :: \alpha \to \omega \to \mathsf{Bool}$ indicates exactly this. An action a can only be performed in world w when $\mathtt{wa}\ a\ w = \textsc{False}$. We say an action a is **stalled** (in world state w) if $\mathtt{wa}\ a\ w = \textsc{True}$.

Figure 1 defines two relations on actions and the API.

- $a_1 \; |||_{\mathfrak{s}} \; a_2$: for any two actions a_1 and a_2, if neither are stalled then the order in which they are executed is irrelevant – both with regard to their effect on world state and their return values. $|||_{\mathfrak{s}}$ is symmetric (but not necessarily transitive or reflexive.)
- $\mathsf{ally}_{\mathfrak{s}}(a_1, a_2)$: If action a_1 is not stalled then performing it cannot cause action a_2, if also not stalled, to then become stalled. The word "ally" describes the fact that action a_1 will not obstruct or hinder action a_2 – they are in effect working with each other.

The $a_1 \; |||_{\mathfrak{s}} \; a_2$ condition allows for two possibilities: either both actions succeed for both orderings or for both orderings one action fails. For example, if $\mathtt{af}\ a_1\ w = (w_1, v_1)$ and $\mathtt{af}\ w_2\ w = (w_2, v_2)$ then it is still possible that $\mathtt{af}\ a_2\ w_1 = \bot$ and $\mathtt{af}\ a_1\ w_2 = \bot$.

I/O Contexts – ap

The functions af and wa define the API. What is now needed is some means of reining in the power of a concurrent (sub-)program by giving it only a

limited set of actions which it is allowed to perform.

We call these permission sets **I/O contexts**. I/O contexts are elements of the type ς and the function $\texttt{ap} :: \varsigma \to \alpha \to \mathsf{Bool}$ defines the actions permitted by any context. A context c can be thought of as the set of actions a such that $\texttt{ap}\ c\ a = \textsc{True}$. Each (sub-)program has a context associated with it at run-time. The context determines what actions the (sub-)program is allowed to perform, and if used in a controlled fashion it gives a mechanism for ensuring determinism.

Figure 1 defines two important relations on contexts.

- $c_1 \sqsubseteq_{\mathfrak{s}} c_2$: any action permitted by context c_1 is also permitted by c_2.
- $c_1 \lozenge_{\mathfrak{s}} c_2$: for all actions a_1 and a_2, permitted by contexts c_1 and c_2 respectively, it is true that $a_1 \mid\mid\mid_{\mathfrak{s}} a_2$, $\mathsf{ally}_{\mathfrak{s}}(a_1, a_2)$ and $\mathsf{ally}_{\mathfrak{s}}(a_2, a_1)$.

($\lozenge_{\mathfrak{s}}$ is symmetric and $\sqsubseteq_{\mathfrak{s}}$ is a pre-order – by definition.)

Enforcing Determinism – `pf`

Say a process running in context c forks into two processes running in contexts c_l and c_r. To guarantee deterministic behaviour:

- The order in which actions in the two child sub-programs are performed should be irrelevant. That is to say: if the run-time system can make a choice between doing an action a_l permitted by c_l or an action a_r permitted by c_r, then neither action can impede the other causing it to become stalled, and when both are finally executed their order shouldn't affect the resultant world state or the actions' return values: $c_l \lozenge_{\mathfrak{s}} c_r$.
- No child process should be allowed to perform an action forbidden by the parent context: $c_l \sqsubseteq_{\mathfrak{s}} c$ and $c_r \sqsubseteq_{\mathfrak{s}} c$.

These properties are usually undecidable. This is a problem, since we want to give describe a language implementation. Therefore we assume the existence of a function which has been proved to obey these exact properties. The function $\texttt{pf} :: \rho \to \varsigma \to (\varsigma, \varsigma)$ splits a context returning two new ones for the two concurrent left and right sub-programs. Elements of the type ρ give the programmer some flexibility with regard to how he or she wishes the current context to be split. $\mathsf{PRE}_{\mathfrak{s}}$, as defined in Figure 2, is the pre-condition that `pf` must obey and we will show later that if it does, any I/O performing program on that I/O model remains deterministic.

Sometimes we don't need the full power of $\mathsf{PRE}_{\mathfrak{s}}$, just something a lot weaker. $\mathsf{pre}_{\mathfrak{s}}$ only guarantees the second of the two properties mentioned above.

$$\mathsf{PRE}_{\mathfrak{s}} \triangleq \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c \wedge c_l \lozenge_{\mathfrak{s}} c_r$$

$$\mathsf{pre}_{\mathfrak{s}} \triangleq \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c$$

Figure 2: The pre-conditions $\mathsf{PRE}_{\mathfrak{s}}$ and $\mathsf{pre}_{\mathfrak{s}}$

2.2 Four Examples

To give an idea of the flexibility of this approach, here are a few examples of complete I/O models all of which obey $\mathsf{PRE}_{\mathfrak{s}}$. (All lemmas have been machine-verified.)

Communication buffers

I/O model $\mathfrak{bfft}$ is a simple 1-1 communication buffer. World state is of type $[\mathsf{Int}]$, and there are two actions: SEND i places i at the end of the list (returning 0, a token value); RCVE removes the first element from the list and returns it. If the current context is SUBC TRUE one can only send, if it is SUBC FALSE one can only receive, and if it is TOPC both are permitted. One can only split the context if it is TOPC. This will give one sub-program the right to send, and the other the right to receive.

Lemma 2.1. $\mathsf{PRE}_{\mathfrak{bfft}}$.

Proof. If $\mathtt{pf}_{\mathtt{Bffr}}$ splits TOPC into SUBC FALSE and SUBC TRUE:

- (SUBC FALSE) $\lozenge_{\mathfrak{bfft}}$ (SUBC TRUE): Only a send and a receive can be performed. RCVE $|||_{\mathfrak{bfft}}$ (SEND i) holds, because if the receive isn't stalled, the buffer must be non-empty, and therefore the actions must affect different parts of the buffer and be order independent. $\mathsf{ally}_{\mathfrak{bfft}}(\text{RCVE}, \text{SEND } i)$ is trivial because a send is always non-stalled. $\mathsf{ally}_{\mathfrak{bfft}}(\text{SEND } i, \text{RCVE})$ is true because adding an element to a buffer cannot cause a receive to become stalled.
- (SUBC FALSE) $\sqsubseteq_{\mathfrak{bfft}}$ TOPC and (SUBC TRUE) $\sqsubseteq_{\mathfrak{bfft}}$ TOPC: trivial (TOPC doesn't forbid any actions.)

□

Many-to-Many Mutexes

I/O model $\mathfrak{lock}$ allows many processes to be synchronised. World state is of type Bool indicating whether or not the lock is set. There are three actions: LOCK sets world state to TRUE; UNLOCK sets world state to FALSE; WAIT will stall until world state is FALSE, then proceed leaving it unchanged.

$$\mathfrak{bfft} :: \mathsf{IOModel}\ \nu_{\mathtt{Bffr}}\ \alpha_{\mathtt{Bffr}}\ \rho_{\mathtt{Bffr}}\ \omega_{\mathtt{Bffr}}\ \varsigma_{\mathtt{Bffr}} \triangleq \langle \mathtt{af}_{\mathtt{Bffr}}, \mathtt{wa}_{\mathtt{Bffr}}, \mathtt{ap}_{\mathtt{Bffr}}, \mathtt{pf}_{\mathtt{Bffr}} \rangle$$

$$\begin{array}{llll} \nu_{\mathtt{Bffr}} \triangleq \mathsf{Int} & \alpha_{\mathtt{Bffr}} \triangleq \textsc{Send}\ \mathsf{Int} \mid \textsc{Rcve} & \omega_{\mathtt{Bffr}} \triangleq [\mathsf{Int}] \\ \rho_{\mathtt{Bffr}} \triangleq \mathsf{Bool} & \varsigma_{\mathtt{Bffr}} \triangleq \textsc{TopC} \mid \textsc{SubC}\ \mathsf{Bool} & \end{array}$$

$$\begin{array}{llll} \mathtt{af}_{\mathtt{Bffr}} & (\textsc{Send}\ i) & is & \triangleq\ (is +\!\!+ [i], 0) \\ \mathtt{af}_{\mathtt{Bffr}} & \textsc{Rcve} & (i : is) & \triangleq\ (is, i) \\ \mathtt{wa}_{\mathtt{Bffr}} & (\textsc{Send}\ i) & is & \triangleq\ \textsc{False} \\ \mathtt{wa}_{\mathtt{Bffr}} & \textsc{Rcve} & is & \triangleq\ \mathtt{null}\ is \end{array}$$

$$\begin{array}{llll} \mathtt{ap}_{\mathtt{Bffr}} & \textsc{TopC} & a & \triangleq\ \textsc{True} \\ \mathtt{ap}_{\mathtt{Bffr}} & (\textsc{SubC}\ b) & (\textsc{Send}\ i) & \triangleq\ b \\ \mathtt{ap}_{\mathtt{Bffr}} & (\textsc{SubC}\ b) & \textsc{Rcve} & \triangleq\ \mathtt{not}\ b \end{array}$$

$$\mathtt{pf}_{\mathtt{Bffr}}\ \ b\ \ \textsc{TopC} \triangleq (\textsc{SubC}\ b, \textsc{SubC}\ (\mathtt{not}\ b))$$

Figure 3: $\mathfrak{bfft}$ – a 1-to-1 communication buffer

Contexts are either "True", meaning all actions are permitted, or "False", meaning just "Unlock" and "Wait" are allowed.

Contexts can be split as many times as one likes, but the child context will always be "False". This means that although a program's context *can* be True at the top-level, allowing "Lock" to be performed, "Lock" can never be performed concurrently with any other action. The world state doesn't keep track of how many processes are waiting for the mutex to be released, so if it was released then locked again, a waiting process might miss this event.

There are no (useful) return values in this I/O model.

Lemma 2.2. $\mathsf{PRE}_{\mathfrak{lock}}$.

Proof. The function $\mathtt{pf}_{\mathtt{Lock}}$ always splits contexts into False and False:

- Proving False $\Diamond_{\mathfrak{lock}}$ False: The combinations of actions are (1) two unlocks, (2) two waits or (3) an unlock and a wait. Unlock $|||_{\mathfrak{lock}}$ Unlock and $\mathsf{ally}_{\mathfrak{lock}}$(Unlock, Unlock) hold because Unlock changes state in the same way, and is never stalled. Wait $|||_{\mathfrak{lock}}$ Wait and $\mathsf{ally}_{\mathfrak{lock}}$(Wait, Wait) is true since Wait doesn't affect world state, is order-independent and also cannot cause any action to become stalled. Unlock $|||_{\mathfrak{lock}}$ Wait and $\mathsf{ally}_{\mathfrak{lock}}$(Unlock, Wait): If both unlocking and waiting are non-stalled, then their order is irrelevant since wait doesn't change the world state – nothing can cause unlock to become stalled, and no amount of unlocking can cause a wait to become stalled.

- False $\sqsubseteq_{\mathfrak{lock}}$ b for all b: True by reflexivity if b = False, and if

$$\mathfrak{lock} :: \mathsf{IOModel}\ \nu_{\mathtt{Lock}}\ \alpha_{\mathtt{Lock}}\ \rho_{\mathtt{Lock}}\ \omega_{\mathtt{Lock}}\ \varsigma_{\mathtt{Lock}} \triangleq \langle \mathtt{af}_{\mathtt{Lock}}, \mathtt{wa}_{\mathtt{Lock}}, \mathtt{ap}_{\mathtt{Lock}}, \mathtt{pf}\, s_{\mathtt{Lock}} \rangle$$

$$\begin{array}{llll} \nu_{\mathtt{Lock}} \triangleq () & \omega_{\mathtt{Lock}} \triangleq \mathsf{Bool} & \multicolumn{2}{l}{\alpha_{\mathtt{Lock}} \triangleq \textsc{Lock} \mid \textsc{Unlock} \mid \textsc{Wait}} \\ \rho_{\mathtt{Lock}} \triangleq () & \varsigma_{\mathtt{Lock}} \triangleq \mathsf{Bool} & \end{array}$$

$$\begin{array}{rrrcl} \mathtt{af}_{\mathtt{Lock}} & \textsc{Lock} & b & \triangleq & (\textsc{True}, ()) \\ \mathtt{af}_{\mathtt{Lock}} & \textsc{Unlock} & b & \triangleq & (\textsc{False}, ()) \\ \mathtt{af}_{\mathtt{Lock}} & \textsc{Wait} & b & \triangleq & (b, ()) \\ \mathtt{pf}_{\mathtt{Lock}} & () & c & \triangleq & (\textsc{False}, \textsc{False}) \end{array}$$

$$\mathtt{wa}_{\mathtt{Lock}}\ a\ b \triangleq \begin{cases} b, & a = \textsc{Wait} \\ \textsc{False}, & \text{otherwise} \end{cases}$$

$$\mathtt{ap}_{\mathtt{Lock}}\ b\ a \triangleq \begin{cases} \textsc{False}, & a = \textsc{Lock}, b = \textsc{False} \\ \textsc{True}, & \text{otherwise} \end{cases}$$

Figure 4: $\mathfrak{lock}$ – a mutex

$b = \textsc{True}$ then it is true because context "True" doesn't forbid any actions.

□

An Integer Variable

The I/O model $\mathfrak{ivar}$ is an integer variable which can be written to and read using the actions ReadI and WriteI. Neither of these can ever be stalled.

The context is either NoneC (no actions are permitted), ReadC (only reading is allowed) or WriteC (reading and writing are both allowed.) When splitting a context, if the context is NoneC or ReadC then the parameter is ignored and the left and right context remain the same as that of the parent. When splitting context "WriteC", however, depending on which of "LeftWr", "RightWr" or "BothRd" is given as parameter, either one side can be allowed to do everything leaving nothing for the other side, or both sides can be allowed to only read the integer.

Lemma 2.3. $\mathsf{PRE}_{\mathfrak{ivar}}$.

Proof. Since no action can be stalled, the $\mathsf{ally}_{\mathfrak{ivar}}$ property holds automatically for any pair of actions.

- Proving $\Diamond_{\mathfrak{ivar}}$: the function $\mathtt{pf}_{\mathtt{IVar}}$ can split contexts in three different ways. NoneC $\Diamond_{\mathfrak{ivar}}$ NoneC and WriteC $\Diamond_{\mathfrak{ivar}}$ NoneC are trivially true, since no actions are permitted by NoneC. When proving ReadC $\Diamond_{\mathfrak{ivar}}$ ReadC, only "ReadI" is allowed, and ReadI $|||_{\mathfrak{ivar}}$ ReadI is true since it doesn't change the world state.

$$\mathfrak{ivar} :: \mathsf{IOModel}\ \nu_{\mathtt{IVar}}\ \alpha_{\mathtt{IVar}}\ \rho_{\mathtt{IVar}}\ \omega_{\mathtt{IVar}}\ \varsigma_{\mathtt{IVar}} \triangleq \langle \mathtt{af}_{\mathtt{IVar}}, \mathtt{wa}_{\mathtt{IVar}}, \mathtt{ap}_{\mathtt{IVar}}, \mathtt{pf}_{\mathtt{IVar}} \rangle$$

$$\begin{array}{rcl@{\qquad}rcl} \nu_{\mathtt{IVar}} & \triangleq & \mathsf{Int} & \rho_{\mathtt{IVar}} & \triangleq & \textsc{LeftWr} \mid \textsc{RightWr} \mid \textsc{BothRd} \\ \omega_{\mathtt{IVar}} & \triangleq & \mathsf{Int} & \varsigma_{\mathtt{IVar}} & \triangleq & \textsc{NoneC} \mid \textsc{ReadC} \mid \textsc{WriteC} \end{array}$$

$$\alpha_{\mathtt{IVar}} \triangleq \textsc{ReadI} \mid \textsc{WriteI}\ \mathsf{Int}$$

$$\begin{array}{lrrcl} \mathtt{af}_{\mathtt{IVar}} & \textsc{ReadI} & i & \triangleq & (i, i) \\ \mathtt{af}_{\mathtt{IVar}} & (\textsc{WriteI}\ i_1) & i & \triangleq & (i_1, 0) \\ \\ \mathtt{wa}_{\mathtt{IVar}} & a & i & \triangleq & \textsc{False} \\ \\ \mathtt{ap}_{\mathtt{IVar}} & \textsc{NoneC} & a & \triangleq & \textsc{False} \\ \mathtt{ap}_{\mathtt{IVar}} & \textsc{ReadC} & \textsc{WriteI} & \triangleq & \textsc{False} \\ \mathtt{ap}_{\mathtt{IVar}} & \textsc{ReadC} & \textsc{ReadI} & \triangleq & \textsc{True} \\ \mathtt{ap}_{\mathtt{IVar}} & \textsc{WriteC} & a & \triangleq & \textsc{True} \end{array}$$

$$\begin{array}{lrrcl} \mathtt{pf}_{\mathtt{IVar}} & p & \textsc{NoneC} & \triangleq & (\textsc{NoneC}, \textsc{NoneC}) \\ \mathtt{pf}_{\mathtt{IVar}} & p & \textsc{ReadC} & \triangleq & (\textsc{ReadC}, \textsc{ReadC}) \\ \mathtt{pf}_{\mathtt{IVar}} & \textsc{LeftWr} & \textsc{WriteC} & \triangleq & (\textsc{WriteC}, \textsc{NoneC}) \\ \mathtt{pf}_{\mathtt{IVar}} & \textsc{RightWr} & \textsc{WriteC} & \triangleq & (\textsc{NoneC}, \textsc{WriteC}) \\ \mathtt{pf}_{\mathtt{IVar}} & \textsc{BothRd} & \textsc{WriteC} & \triangleq & (\textsc{ReadC}, \textsc{ReadC}) \end{array}$$

Figure 5: $\mathfrak{ivar}$ – a shared integer variable

- $c_1 \sqsubseteq_{\mathfrak{ivar}} c_2$ for all c_1, c_2 as split by $\mathtt{pf}_{\mathtt{IVar}}$: It is clear that NoneC $\sqsubseteq_{\mathfrak{ivar}}$ ReadC and ReadC $\sqsubseteq_{\mathfrak{ivar}}$ WriteC, so it can be seen directly from the definition of `pfIVar` that when contexts are split this property is always obeyed.

□

Terminal I/O

As a small real-world I/O example, consider the model of terminal I/O given in Figure 6.

There are two actions: (PutC c) writes character c, and GetC reads a character. The model is not perfect, since it does not capture any interesting temporal properties of `stdin`/`stdout`. There is no notion of absolute time – each outputted `Char` gives rise to 0 or more inputted characters instantaneously. It is adequate, nonetheless, and obeys the simple property that if characters are available for input then outputting characters cannot change that. The model also lets us exploit the fact that `stdin` and `stdout`

$$\mathfrak{term} :: \mathsf{IOModel}\ \nu_{\mathtt{Term}}\ \alpha_{\mathtt{Term}}\ \rho_{\mathtt{Term}}\ \omega_{\mathtt{Term}}\ \varsigma_{\mathtt{Term}} \triangleq \langle \mathtt{af}_{\mathtt{Term}}, \mathtt{wa}_{\mathtt{Term}}, \mathtt{ap}_{\mathtt{Term}}, \mathtt{pf}_{\mathtt{Term}} \rangle$$

$$\begin{array}{rcl@{\qquad}rcl} \nu_{\mathtt{Term}} & \triangleq & \mathsf{Char} & \alpha_{\mathtt{Term}} & \triangleq & \textsc{PutC}\ \mathsf{Char} \mid \textsc{GetC} \\ \omega_{\mathtt{Term}} & \triangleq & (\mathsf{[Char]}, \mathsf{TermIO}) & \rho_{\mathtt{Term}}, \varsigma_{\mathtt{Term}} & \triangleq & \textsc{IOcxt}\ \mathsf{Bool}\ \mathsf{Bool} \end{array}$$

$$\mathsf{TermIO} \triangleq \textsc{TermIO}\ (\mathsf{Char} \to (\mathsf{[Char]}, \mathsf{TermIO}))$$

$$\begin{array}{lllcl} \mathtt{wa}_{\mathtt{Term}} & (\textsc{PutC}\ c) & (cs, t) & \triangleq & \textsc{False} \\ \mathtt{af}_{\mathtt{Term}} & (\textsc{PutC}\ c) & (cs, \textsc{TermIO}\ f) & \triangleq & ((cs +\!\!+ \mathtt{fst}\ (f\ c), \mathtt{snd}\ (f\ c)), \text{' '}) \\ \mathtt{wa}_{\mathtt{Term}} & \textsc{GetC} & (cs, t) & \triangleq & \mathtt{null}\ cs \\ \mathtt{af}_{\mathtt{Term}} & \textsc{GetC} & ((c : cs), t) & \triangleq & ((cs, t), c) \\ \\ \mathtt{ap}_{\mathtt{Term}} & (\textsc{IOcxt}\ b\ \_) & (\textsc{PutC}\ c) & \triangleq & b \\ \mathtt{ap}_{\mathtt{Term}} & (\textsc{IOcxt}\ \_\ b) & \textsc{GetC} & \triangleq & b \end{array}$$

$$\begin{array}{l} \mathtt{pf}_{\mathtt{Term}}\ (\textsc{IOcxt}\ bp_p\ bg_p)\ (\textsc{IOcxt}\ bp\ bg) \triangleq \\ \quad (\textsc{IOcxt}\ (bp\ \mathtt{\&\&}\ bp_p)\ (bg\ \mathtt{\&\&}\ bg_p), \textsc{IOcxt}\ (bp\ \mathtt{\&\&}\ \mathtt{not}\ bp_p)\ (bg\ \mathtt{\&\&}\ \mathtt{not}\ bg_p)) \end{array}$$

Figure 6: $\mathfrak{term}$ – a model for Terminal I/O

are usually two separate handles, and we may want a process to wait for input on one whilst another outputs data on the other. (This is probably only useful for filtering programs such as `grep`. Interactive programs would usually require a lock-step synchronisation of input and output).

The proof of $\mathsf{PRE}_{\mathfrak{term}}$ is not difficult. It permits the same degree of concurrency as the $\mathfrak{bfft}$ model. Reading may take place in parallel with writing in this semantics since (1) if a `Char` is ready for input then writing a character will not change that and (2) if a character is not available for input then the read will stall until one is. The function $\mathtt{pf}_{\mathtt{Term}}$ guarantees that if many processes are running concurrently then at most one can call PutC and at most one can perform GetC.

The $\mathfrak{term}$ model may be seen, in one sense, as just a generalisation of $\mathfrak{bfft}$. Consider the type TermIO once again. One possible instance of it is `loopBack`, defined as follows:

```
loopBack :: TermIO
loopBack = TermIO (\c -> ([c],loopBack))
```

This really just models the "semantics" of a user who re-inputs every character outputted to him/her. Therefore it is no different to a communication buffer between the sending and receiving processes.

2.3 Discussion

The small examples give a flavour of what is possible. One of the most surprising and unusual aspects of this system, we believe, is that we don't need to explicitly mention communication channels at all. By permitting actions to be temporarily stalled, one then has enough machinery to synchronise processes and safely transfer information via the world state.

The key to guaranteeing determinism is making sure that when a context c is split into c_l and c_r to allow concurrency, an action in one context cannot influence the behaviour of an action in the other context. This means that an action permitted by c_l cannot block an action permitted by c_r (or vice versa) and the order in which they are executed must be irrelevant. This rules out competition for a limited resource. Some typically non-deterministic constructs are multiple-writer streams, multiple-reader streams and shared, mutable variables.

Actions can become stalled, however, like "RCVE" in the buffer example. Roughly speaking, an action a_1 can only cause a (not necessarily different) action a_2 to become stalled if they can under no circumstances be executed concurrently. Also, if an action a_1 can successfully predict whether another action a_2 is going to stall then a_1 cannot be run concurrently with any action which changes whether or not a_2 is stalled. For example, in the $\mathfrak{buffer}$ model neither of the sub-contexts, which allow just sending or just receiving, could be modified to permit an action which returns whether or not the buffer is empty. However, the receiving context could allow one to wait until the buffer is non-empty and the sending context could allow one to wait until the buffer is empty, and neither would admit non-determinism.

There are a few interesting sub-classes of actions.

- Observer actions, which never change world state (like "WAIT" and "READI".) If a_1 is an observer action, then for any a_0, $\mathsf{ally}_{\mathfrak{s}}(a_1, a_0)$ and for any two observer actions a_1 and a_2, $a_1 \mid\mid\mid_{\mathfrak{s}} a_2$ holds.
- Actions which always return the same value (like "LOCK", "WRITEI 7", or an action which, say, increments a counter without indicating it's value.) If action a_1 is of this form then $a_1 \mid\mid\mid_{\mathfrak{s}} a_1$.
- Actions which can never be stalled, like "SEND i" and "WRITEI i". If action a_1 is never stalled then for any action a_0, $\mathsf{ally}_{\mathfrak{s}}(a_0, a_1)$.
- Commutative contexts: if it is true that $c \Diamond_{\mathfrak{s}} c$ for some c, (which will be true if there is some p such that `pf` p c = `(`c`,`c`)`, as is the case with READC in the $\mathfrak{ivar}$ example) then the actions permitted by c in a monadic setting form a commutative monad. In other words, there are absolutely no constraints on the ordering of actions.

It should also be noted that many "reasonable" properties of I/O models are not always present. These include: a context which permits all actions;

a context which permits no actions; a way of splitting a context such that all permissions are given to one side only; a way splitting contexts in a symmetric way (if a context can be split into (c_l, c_r) can it also be split into (c_r, c_l)?). These weren't necessary for the confluence proof, so we didn't bother to include them, but their existence would probably make things somewhat tidier.

3 Curio – a Language for Reasoning About I/O

In this section we introduce the Curio language. This is a small functional language with concurrency, interprocess communication and monadic constructs which together let the user write expressive programs which perform I/O. This expressivity is due to our ability to enforce determinism.

In reality, Curio is less a full language specification than a rigorous semantics for a collection of powerful I/O primitives which can be wrapped around a pure functional language using solely its denotational semantics. This overall approach is not new. In fact, at the outset it bears many similarities to that taken by the Haskell community:

> "Our semantics is stratified in two levels: an inner denotational semantics which describes the behaviour of pure terms, while an outer monadic semantics describes the behaviour of `IO` computations." [Pey01]

The Concurrent Haskell language [PGF96] is probably the lazy functional language with the most fully-fledged semantics for I/O. What makes Curio different to the semantics of Concurrent Haskell is the nature of this outer operational semantics.

Concurrent Haskell's semantics for I/O is in effect a *co-inductive* one. I/O actions represent labelled transitions in a CCS-style process calculus. Each action is a distinct observable event, and the meaning of a program is solely determined by the (possibly non-terminating) order in which actions occur.

Curio's semantics is *inductive* rather than co-inductive. There exists a "world-state" which the program interacts with and modifies when doing I/O, and the observable effect of a program is its resultant world-state. Whereas with Concurrent Haskell all actions are observable, with Curio it is only the cumulative effect of a finite number of actions over a program's lifetime that is observable.

Naturally there are pros and cons to both approaches. The most immediate advantage to the CCS-style semantics is that by using it one can distinguish two infinite programs (for example, a program which loops continuously and, say, an internet server designed never to terminate). It is also highly elegant.

What makes the semantics of Curio somewhat more expressive (if less elegant) is that it ascribes to each action a precise effect on world state, and therefore allows us to distinguish actions which do and do not interfere with one another. Insofar as a language semantics is a vehicle for compiler optimisations and formal proofs, this would appear to make it better – but we cannot tell at this early stage.

3.1 Language Primitives

A program in CURIO is any element of the type $\mathsf{Prog}_{\nu\alpha\rho}\ \beta$ and the five I/O primitives are as follows: (unless otherwise specified it is assumed that the types ν, α and ρ are all bound by some I/O model).

$$\begin{aligned}
\texttt{(>>=)} &:: \mathsf{Prog}_{\nu\alpha\rho}\ \beta \to (\beta \to \mathsf{Prog}_{\nu\alpha\rho}\ \gamma) \to \mathsf{Prog}_{\nu\alpha\rho}\ \gamma \\
\texttt{return} &:: \beta \to \mathsf{Prog}_{\nu\alpha\rho}\ \beta \\
\texttt{action} &:: \alpha \to \mathsf{Prog}_{\nu\alpha\rho}\ \nu \\
\texttt{test} &:: \alpha \to \mathsf{Prog}_{\nu\alpha\rho}\ \beta \to \mathsf{Prog}_{\nu\alpha\rho}\ \beta \to \mathsf{Prog}_{\nu\alpha\rho}\ \beta \\
\texttt{par} &:: \rho \to \mathsf{Prog}_{\nu\alpha\rho}\ \beta \to \mathsf{Prog}_{\nu\alpha\rho}\ \gamma \to (\beta \to \gamma \to \epsilon) \to \mathsf{Prog}_{\nu\alpha\rho}\ \epsilon
\end{aligned}$$

Central to these language primitives is the notion of an I/O context. Each (sub-)program is executed within a context $c \in \varsigma$, and that context dictates which actions that (sub-)program can perform, and therefore affects the programs outcome.

The full semantics is given further on, but very briefly: the first two are the familiar monadic `>>=` and `return`; `action` performs a primitive action; `test` performs one of two programs depending on whether an action is allowed by the current context; `par` runs two programs concurrently.

A program is said to be a **value** (or **evaluated**) if it is of the form `return` v, for some v. A program is said to be an **action** if it is of the form `action` a, for some a. Programs of the form `par` $p\ m_l\ m_r\ (*)$ are abbreviated using an infix notation as $m_l\ |||^p_*\ m_r$.

Of the five original types, α and ρ end up being part of the language's syntax, ς is best understood as an internal run-time data-structure in the language implementation, and ω is part of the world model (i.e. part of the language's semantics).

CURIO is a non-strict language. None of the five primitives evaluate their arguments. Therefore programs can be partial or infinite, as would be expected in a real lazy language. Consider the following example program: if the current context allows a read then an infinite number of reads are performed concurrently, otherwise -1 is returned.

```
let  reads = par BothRd (action ReadI) reads f
in   test ReadI reads (return (−1))
```

The above example serves to demonstrate why our style of operational semantics will have to be a bit unusual. We don't have lists, or "let" constructs in our language yet we are using them freely when writing programs. Since programs like the above are permitted, our operational semantics can't just manipulate a program as normal syntax. Instead we must use the denotational semantics of the metalanguage and build an operational semantics which acts on and modifies elements of domains – we reason about, induct

over and manipulate the domain of type $\mathsf{Prog}_{\nu\alpha\rho}\ \beta$, not the syntax of programs of type $\mathsf{Prog}_{\nu\alpha\rho}\ \beta$. This has some immediate consequences. Since all our language constructs are lazy, and since there are terms of type $\mathsf{Prog}_{\nu\alpha\rho}\ \beta$ which don't have an outermost constructor, we must also describe the program's behaviour when sub-programs are undefined, or $\bot$.

3.2 Examples

We now give example programs making use of Haskell's do-notation.

These examples are specific to certain I/O models, so the types ν, α and ρ will differ. Given some I/O model, say $\mathfrak{bfft}$, to avoid clutter we use $\mathsf{Prog}_{\mathfrak{bfft}}\ \beta$ as a convenient shorthand for $\mathsf{Prog}_{\nu_{\mathtt{Bffr}}\ \alpha_{\mathtt{Bffr}}\ \rho_{\mathtt{Bffr}}}\ \beta$.

Communication Buffer Examples

The following program attempts to send a list of integers along a communication buffer. The Boolean return value indicates whether the current context permitted the SEND action.

```
sendInts :: [Int] → Prog_bfft Bool
sendInts []     = return True
sendInts (i:is) =
  test (Send i)
    (action (Send i) >>= \_ -> sendInts is)
    (return False)
```

The program `rcveInts` c attempts to retrieve c integers from the communication buffer. If receiving is not permitted by the program's context then it returns `[]`, the empty list.

```
rcveInts :: Int → Prog_bfft [Int]
rcveInts c = test Rcve (rcveInts' c) (return [])
  where rcveInts' | c<=0      = return []
                  | otherwise = do
                                  i  <- action Rcve
                                  is <- rcveInts (c-1)
                                  return (i:is)
```

Mutex Examples

The program `lockPar` runs two $\mathfrak{lock}$ programs in parallel combining the two resultant return values into a tuple. This is completely general since in the $\mathfrak{lock}$ model the ρ type has just one element anyway, `()`.

```
lockPar :: Prog_lock β → Prog_lock γ → Prog_lock (β,γ)
lockPar pl pr = par () pl pr (\vl vr -> (vl,vr))
```

Integer Variable Examples

The following program applies a function f to the integer variable returning TRUE if the context permitted reading and writing.

```
applyFn :: (Int → Int) → Prog_ivar Bool
applyFn f = do
  b <- test (WriteI 0) (return True) (return False)
  if b then do
    i <- action ReadI
    action (WriteI (f i))
  return b
```

Terminal I/O Examples

The function `putStr` writes a string to `stdout` (or fails if that is not allowed).

```
putStr :: String → Prog_term ()
putStr []     = return ()
putStr (c:cs) = action (PutC c) >>= \_ -> putStr cs
```

`stdPermissions` returns a string which indicates which actions the current context permits.

```
stdPermissions :: Prog_term String
stdPermissions = do
  in  <- test GetC (return "stdin") (return "no stdin")
  out <- test (PutC 'x') (return "stdout") (return "no stdout")
  return (in ++ ", " ++ out)
```

The program `getPutC c` outputs character `c` whilst concurrently requesting a character from input. The entire program returns the character that was read.

```
getPutC :: Char → Prog_term Char
getPutC c = par
  (IOcxt True False) (action (PutC c)) (action GetC) (\_ c1 -> c1)
```

3.3 Semantics

The non-deterministic single-step semantics can be found in Figure 7. We use an SOS-style notation [Plo81].

All the reduction rules describe the behaviour of what we call a world/program pair, written $w \Vdash m$, where w is the world and m is the program. This allows one to describe how a program and world-state interact over time. The context $c \in \varsigma$ in which a program is run also affects how the program behaves, so reduction rules are also annotated with the current context.

$$\frac{w \Vdash m \uparrow^c}{w \Vdash m \texttt{ >>= } f \uparrow^c} \qquad \frac{w \Vdash m \downarrow^c}{w \Vdash m \texttt{ >>= } f \downarrow^c} \quad (m \text{ not a value})$$

$$w \Vdash \texttt{return } v \texttt{ >>= } f \longrightarrow^c w \Vdash f\ v \qquad \frac{w \Vdash m \longrightarrow^c w_1 \Vdash m_1}{w \Vdash m \texttt{ >>= } f \longrightarrow^c w_1 \Vdash m_1 \texttt{ >>= } f}$$

`ap` c a	`wa` a w	`af` a w	Behaviour of `action` a
$\bot$			$w \Vdash \texttt{action}\ a \uparrow^c$
False			$w \Vdash \texttt{action}\ a \uparrow^c$
True	$\bot$		$w \Vdash \texttt{action}\ a \uparrow^c$
True	False	$\bot$	$w \Vdash \texttt{action}\ a \uparrow^c$
True	False	(w_1, v)	$w \Vdash \texttt{action}\ a \longrightarrow^c w_1 \Vdash \texttt{return}\ v$
True	True		$w \Vdash \texttt{action}\ a \downarrow^c$

`ap` c a	Behaviour of `test` $a\ m_1\ m_2$
$\bot$	$w \Vdash \texttt{test}\ a\ m_1\ m_2 \longrightarrow^c w \Vdash \bot$
False	$w \Vdash \texttt{test}\ a\ m_1\ m_2 \longrightarrow^c w \Vdash m_2$
True	$w \Vdash \texttt{test}\ a\ m_1\ m_2 \longrightarrow^c w \Vdash m_1$

$$w \Vdash \texttt{return } v \downarrow^c$$

$$w \Vdash \bot \uparrow^c$$

$$\frac{\texttt{pf}\ p\ c\ =\ \bot}{w \Vdash m_l \texttt{ ||| }^p_* m_r \uparrow^c}$$

$$\texttt{pf}\ p\ c = (c_l, c_r) \begin{cases} \dfrac{w \Vdash m_l \uparrow^{c_l}}{w \Vdash m_l \texttt{ ||| }^p_* m_r \uparrow^c} \quad \dfrac{w \Vdash m_r \uparrow^{c_r}}{w \Vdash m_l \texttt{ ||| }^p_* m_r \uparrow^c} \\[2ex] w \Vdash \texttt{return } v_l \texttt{ ||| }^p_* \texttt{return } v_r \longrightarrow^c w \Vdash \texttt{return } v_l * v_r \\[2ex] \dfrac{w \Vdash m_l \longrightarrow^{c_l} w' \Vdash m_l'}{w \Vdash m_l \texttt{ ||| }^p_* m_r \longrightarrow^c w' \Vdash m_l' \texttt{ ||| }^p_* m_r} \ (m_r \neq \bot) \\[2ex] \dfrac{w \Vdash m_r \longrightarrow^{c_r} w' \Vdash m_r'}{w \Vdash m_l \texttt{ ||| }^p_* m_r \longrightarrow^c w' \Vdash m_l \texttt{ ||| }^p_* m_r'} \ (m_l \neq \bot) \\[2ex] \dfrac{w \Vdash m_l \downarrow^{c_l} \quad w \Vdash m_r \downarrow^{c_r}}{w \Vdash m_l \texttt{ ||| }^p_* m_r \downarrow^c} \ (m_l, m_r \text{ not both values}) \end{cases}$$

Figure 7: Non-deterministic Single-Step Semantics for Curio

There are three reduction relations, $\longrightarrow^c$, $\uparrow^c$ and $\downarrow^c$.

$$
\begin{array}{rcl}
w \Vdash m \longrightarrow^c w' \Vdash m' & \triangleq & \text{“}w \Vdash m \text{ can reduce to } w' \Vdash m' \text{ in context } c\text{”} \\
w \Vdash m \uparrow^c & \triangleq & \text{“}w \Vdash m \text{ can fail in context } c\text{”} \\
w \Vdash m \downarrow^c & \triangleq & \text{“}w \Vdash m \text{ is in normal form in context } c\text{”}
\end{array}
$$

We say a world/program pair *is* in normal form (rather than can be in normal form) because, as Lemma 4.13 in Section 4 shows, despite nondeterminism, if a program has converged to normal form then it cannot either fail or reduce. Similarly, if a world/program pair can fail or reduce, then it cannot be in normal form. Failure may seem a slightly curious thing to include, but, as the implementation in the next subsection will show, it is necessary. A single reduction step in Curio can correspond to many individual steps in the operational semantics of the metalanguage. Equivalently, a failed single-step reduction may denote a never-ending sequence of reduction steps in the metalanguage.

Values (programs of the form `return` v) on their own are in normal form. If a world/program pair is in normal form but not a value, then we say it is **stalled**. If this occurs, the convergence to normal form is caused by stalled actions within the program.

`action` a attempts to perform action a. If it isn't permitted by the context, `action` a always fails. If it is permitted, it may fail, be stalled or reduce to the action's return value all depending on the API. If it reduces, it will modify the world state. `test` $a\ m_t\ m_f$ allows a program to query the context in which it is being run. If action a is permitted in the current context then `test` $a\ m_t\ m_f$ executes m_t, otherwise it executes m_f. Usually m_f would be a sort of exception handler, returning a value indicating that certain actions were locked out by the current context. Programs of the form m `>>=` f behave in the normal monadic style: m is reduced continually until it is a value `return` v for some v, then $f\ v$ is reduced. If m at any stage fails or becomes stalled, the same will happen to m `>>=` f. A program $\bot$ always fails.

A **base program** refers to any program of the form `return` v, $\bot$, `return` v `>>=` f, `action` a, `test` $a\ m_t\ m_f$ or `return` v_l $|||^p_*$ `return` v_r. The behaviour of these programs is always entirely deterministic for a given context and world. It is concurrency on its own which introduces nondeterminism. m_l $|||^p_*$ m_r, executed in context c, runs m_l and m_r in parallel in contexts c_l and c_r respectively, where `pf` $p\ c$ = `(`c_l`,`c_r`)`. If two concurrent sub-programs can either reduce or fail, then either side may be chosen arbitrarily. This continues until

- one side fails, causing both programs in parallel to fail.
- m_l and m_r become values `return` v_l and `return` v_r respectively,

in which case the parallel execution terminates, becoming the value `return` $v_l * v_r$.

- one side becomes stalled and the other side converges to normal form (that is: it is either a value or also stalled), causing the concurrent execution of both programs to be stalled.

This non-deterministic single-step form of semantics is the most "obviously correct" way of describing concurrency. The side-condition on the parallel reduction rules that $m_l \neq \bot$ and $m_r \neq \bot$ is undesirable but hard to avoid. If we check at the outset whether both the left and right sub-programs are values it makes the implementation *much* simpler. Unfortunately this means forcing both m_l and m_r to an outermost constructor.

3.4 Convergence/Divergence and the Implementation

Figure 8 contains information about how the language is implemented in the metalanguage.

Programs in Curio are elements of a higher-order algebraic type. The use of an algebraic type is necessary since we need to be precise about how there are exactly five ways of constructing a program. This means that for our machine-verified proofs the types of `>>=` and `par` become monomorphic. This isn't a serious problem. The property of having only five constructors can be informally guaranteed in a real language using class and module interfaces, and it is not necessary for a program to be able to query another program's outermost constructor, which is something that algebraic types specifically allow.

The function $\texttt{next}_\mathfrak{s}$ implements single-step reduction, and non-determinism is implemented by supplying it with an additional parameter of type Guess. This is, roughly, a stream of Boolean values and it guides the reduction algorithm's search for a redex in the presence of concurrency. We existentially quantify over its values to show that reducing to a particular reduct is possible. If a world/program pair is in normal form in some context then $\texttt{next}_\mathfrak{s}$ returns Converged. Otherwise it returns Reduct (w_1, m_1), for some w_1, m_1, or fails ($\bot$). These three possibilities define the three single-step reduction relations, $\downarrow$, $\longrightarrow$ and $\uparrow$.

To make the move from a single-step (or reduction) semantics to a big-step (or evaluation) semantics we must investigate the repeated single-step reduction of a world/program pair. The single-step semantics is non-deterministic, so non-determinism must also have some presence in a big-step semantics. First off, one must define a non-deterministic evaluation relation. $w \Vdash m \Downarrow^c w' \Vdash m'$ means "$w \Vdash m$ can, after zero or more single-step reductions in context c yield $w' \Vdash m'$, which is in normal form."

This is a rather weak property. It would be nice if we knew that $w \Vdash m$ would *always* reduce to a normal form $w' \Vdash m'$ in some context c. This

$$
\begin{array}{rcl}
\mathsf{Prog}\ \nu\ \alpha\ \rho & \triangleq & \textsc{Bind}\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\nu \to \mathsf{Prog}\ \nu\ \alpha\ \rho) \\
& | & \textsc{Ret}\ \nu \\
& | & \textsc{Action}\ \alpha \\
& | & \textsc{Test}\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\mathsf{Prog}\ \nu\ \alpha\ \rho) \\
& | & \textsc{Par}\ \rho\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\nu \to \nu \to \nu)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Reduction}\ \beta & \triangleq & \textsc{Reduct}\ \beta\ |\ \textsc{Converged} \\
\mathtt{next_s} & :: & \mathsf{Guess} \to \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \to \mathsf{Reduction}\ (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \\
\mathtt{rdce_s} & :: & \mathsf{Nat} \to [\mathsf{Guess}] \to \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho)
\end{array}
$$

$$
\begin{array}{c}
\mathtt{rdce_s}\ (i+1)\ [g{:}gs]\ c\ (w,m) = (w'',m'') \\
\Longleftrightarrow \\
\left(\exists_{w' \in \omega}.\exists_{m' \in \mathsf{Prog}\ \nu\ \alpha\ \rho}.\ \begin{array}{c} \mathtt{next_s}\ g\ c\ (w,m) = \textsc{Reduct}\ (w',m') \wedge \\ \mathtt{rdce_s}\ i\ gs\ c\ (w',m') = (w'',m'') \end{array} \right)
\end{array}
$$

$$
\mathtt{rdce_s}\ 0\ gs\ c\ (w,m) = (w,m)
$$

$$
\begin{array}{rcl}
w \Vdash m \longrightarrow^c w' \Vdash m' & \triangleq & \exists_{g \in \mathsf{Guess}}.\mathtt{next_s}\ g\ c\ (w,m) = \textsc{Reduct}\ (w',m') \\
w \Vdash m \downarrow^c & \triangleq & \exists_{g \in \mathsf{Guess}}.\mathtt{next_s}\ g\ c\ (w,m) = \textsc{Converged} \\
w \Vdash m \uparrow^c & \triangleq & \exists_{g \in \mathsf{Guess}}.\mathtt{next_s}\ g\ c\ (w,m) = \bot \\
w \Vdash m\ ^i\!\!\longrightarrow\!\!\!\!\!\rightarrow^c w' \Vdash m' & \triangleq & \exists_{gs \in [\mathsf{Guess}]}.\mathtt{rdce_s}\ i\ gs\ c\ (w,m) = (w',m') \\
w \Vdash m \longrightarrow\!\!\!\!\!\rightarrow^c w' \Vdash m' & \triangleq & \exists_{i \in \mathbb{N}}.w \Vdash m\ ^i\!\!\longrightarrow\!\!\!\!\!\rightarrow^c w' \Vdash m' \\
w \Vdash m\ ^i\!\!\downarrow\!\!\!\downarrow^c w' \Vdash m' & \triangleq & w \Vdash m\ ^i\!\!\longrightarrow\!\!\!\!\!\rightarrow^c w' \Vdash m' \quad \wedge \quad w' \Vdash m' \downarrow^c \\
w \Vdash m \downarrow\!\!\!\downarrow^c w' \Vdash m' & \triangleq & \exists_{i \in \mathbb{N}}.w \Vdash m\ ^i\!\!\downarrow\!\!\!\downarrow^c w' \Vdash m' \\
w \Vdash m\ ^i\!\Downarrow^c\ w' \Vdash m' & \triangleq & \forall_{gs \in [\mathsf{Guess}]}.\mathtt{rdce_s}\ i\ gs\ c\ (w,m) = (w',m') \\
& & \qquad \wedge\ w' \Vdash m' \downarrow^c \\
w \Vdash m \Downarrow^c\ w' \Vdash m' & \triangleq & \exists_{i \in \mathbb{N}}.w \Vdash m\ ^i\!\Downarrow^c\ w' \Vdash m' \\
w \Vdash m \Uparrow^c & \triangleq & \neg\exists_{w' \in \omega}.\exists_{m' \in \mathsf{Prog}\ \nu\ \alpha\ \rho}.w \Vdash m \downarrow\!\!\!\downarrow^c w' \Vdash m'
\end{array}
$$

Figure 8: Implementation Details

is expressed as $w \Vdash m \Downarrow^c w' \Vdash m'$. Divergence in Curio, expressed as $w \Vdash m \Uparrow^c$, means that a program can under no circumstances reduce to some normal form in context c. The relationship between $\uparrow$ and $\Uparrow$ is subtle. The former is failure in the denotational semantics of the metalanguage, the latter is failure in the operational semantics of our language. Theorem 4.2 in Section4 states that if $\mathsf{PRE}_{\mathtt{s}}$, then $w \Vdash m \uparrow^c$ implies $w \Vdash m \Uparrow^c$.

The relation $\twoheaddownarrow$ is implemented with $\mathtt{rdce}_{\mathtt{s}}$ and expresses the repeated single-step reduction of a world/program pair until it is in normal form. It requires a list, or stream of Guesses, one for each single-step reduction (we don't mention any boundary or definedness conditions, but they do exist.) Convergence, $\Downarrow$, differs to $\twoheaddownarrow$ in that it universally quantifies over the guesses. It is trivially true that $w \Vdash m \Downarrow^c w' \Vdash m'$ implies $w \Vdash m \twoheaddownarrow^c w' \Vdash m'$ and the confluence proof shows that if $\mathsf{PRE}_{\mathtt{s}}$ holds, $w \Vdash m \twoheaddownarrow^c w' \Vdash m'$ implies $w \Vdash m \Downarrow^c w' \Vdash m'$. Sometimes $\twoheaddownarrow$ and $\Downarrow$ are annotated with a number which denotes how many reduction steps took place.

Proposition 3.1. *If $w \Vdash m \twoheaddownarrow^c w_1 \Vdash m_1$ then it is not the case that $w \Vdash m \Uparrow^c$.*

Proof. Direct from the definition of $\Uparrow$. □

Proposition 3.2. *Each of the following statements imply those below it:*

1. $w \Vdash m \Downarrow^c w_1 \Vdash m_1$
2. $w \Vdash m \twoheaddownarrow^c w_1 \Vdash m_1$
3. $w \Vdash m \twoheadrightarrow^c w_1 \Vdash m_1$

Proof. (1) implies (2) since if a property holds for all Guesses it must hold for some Guess. (2) implies (3) because the only extra condition on the former is that $w_1 \Vdash m_1 \downarrow^c$ is in normal form. □

It should also be noted that although it's an implementation in a real language, we have no interest at the moment in its efficiency.

3.5 Convergence is Recursively Enumerable

It is reassuring to note that although reduction etc. are defined in the form of an existential quantification over reduction steps, this is just for convenience.

We now prove separately that $\twoheaddownarrow$ is recursively enumerable. This is a (machine-verified) proof that there exists a function

$$\mathtt{run}_{\mathtt{s}} :: [\mathsf{Guess}] \to \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho)$$

such that

$$\texttt{run}_\mathfrak{s}\ gs\ c\ (w,m) = (w_1,m_1)$$
$$\Longleftrightarrow$$
$$\exists_{i\in\mathbb{N}}.\texttt{rdce}_\mathfrak{s}\ i\ gs\ c\ (w,m) = (w_1,m_1)\ \wedge\ w_1 \Vdash m_1 \downarrow^c$$

Once this is proved, convergence and divergence can be expressed in a more intuitive manner as follows:

$$\begin{aligned} w \Vdash m \not\Downarrow^c\ w_1 \Vdash m_1 &\iff \exists_{gs\in[\mathsf{Guess}]}.\texttt{run}_\mathfrak{s}\ gs\ c\ (w,m) = (w_1,m_1) \\ w \Vdash m \Downarrow^c\ w_1 \Vdash m_1 &\iff \forall_{gs\in[\mathsf{Guess}]}.\texttt{run}_\mathfrak{s}\ gs\ c\ (w,m) = (w_1,m_1) \\ w \Vdash m \Uparrow^c &\iff \forall_{gs\in[\mathsf{Guess}]}.\texttt{run}_\mathfrak{s}\ gs\ c\ (w,m) = \bot \end{aligned}$$

Showing that this holds is a non-trivial task. If one writes an implementation which just repeatedly applies $\texttt{next}_\mathfrak{s}$, perhaps an infinite number of times, then there is no structure to induct over. We must build an implementation which internally constructs an intermediate list.

The proof relies on a more general (and quite powerful) lemma. Consider the following two functions:

```
      iterate  ::  (β → β) → β → [β]
iterate f x  ≜  x : iterate f (f x)
      dountil  ::  (β → β) → (β → Bool) → β → [β]
dountil f p x  ≜  x : if (p x) then [] else (dountil f p (f x))
```

`iterate` is a standard Haskell library function. The term `iterate` $f\ x$ creates an infinite list, each successive element of which contains the next iteration of function f to an initial value x. `dountil` is somewhat similar except there is an extra computable predicate $p :: \beta \to \mathsf{Bool}$ which indicates whether the iteration of f should stop. Therefore `dountil` may or may not return an infinite list.

The following lemma gives a useful relationship between the two. It shows that given side-conditions relating p and f, some i applications of f to x_0 yields an x_1 such that $p\ x_1$ if and only if `last` (`dountil` $f\ p\ x_0$) $= x_1$ and x_1 is defined.

Lemma 3.1.

$$\begin{aligned} &\forall_{f\in\beta\to\beta}.\forall_{p\in\beta\to\mathit{Bool}}.p\ \bot = \bot\ \wedge\ f\ \bot = \bot\ \wedge\ (\forall_{x'\in\beta}.p\ x' \neq \textsc{False} \Longrightarrow f\ x' = \bot) \\ &\quad\Longrightarrow \forall_{x_0\in\beta}.\forall_{x_1\in\beta}. \\ &\qquad(\exists_{i\in\mathbb{N}}.x_1 = \mathit{iterate}\ f\ x_0\ !!\ i\ \wedge\ p\ x_1) \\ &\qquad\quad\Longleftrightarrow \\ &\qquad(x_1 = \mathit{last}\ (\mathit{dountil}\ f\ p\ x_0)\ \wedge\ x_1 \neq \bot) \end{aligned}$$

Proof. The proof needs quite a few extra lemmas, and on the whole requires an extremely careful treatment of non-termination. In particular, with the given side-conditions

- If $f\ x = \bot$ then $\texttt{iterate}\ f\ x = [x, \bot, \bot, \bot, \cdots]$.
- If $p\ x = \bot$ then $\texttt{dountil}\ f\ p\ x = x : \bot$ and $\texttt{iterate}\ f\ x = [x, \bot, \bot, \bot, \cdots]$.
- If $p\ x = \textsc{True}$ then $\texttt{dountil}\ f\ p\ x = [x]$ and $\texttt{iterate}\ f\ x = [x, \bot, \bot, \bot, \cdots]$.
- If $p\ x = \textsc{False}$ and $f\ x = \bot$ then $\texttt{dountil}\ f\ p\ x = x : (\bot : \bot)$.

To prove the $\Longrightarrow$ direction we induct over i, the number of iterations required. We must show that if an iteration results in $\bot$ then it cannot revert back to a non-$\bot$ term and also prove that if $p\ x = \textsc{False}$ then $\texttt{last}\ (\texttt{dountil}\ f\ p\ x) = \texttt{last}\ (\texttt{dountil}\ f\ p\ (f\ x))$.

To prove the $\Longleftarrow$ direction we must prove first that

- if $l = \texttt{length}\ (\texttt{dountil}\ f\ p\ x)$ and $l \neq \bot$ then $p\ (\texttt{iterate}\ f\ x\ !!\ (l - 1))$.
- if for all i, $0 \leq i < k$, $p\ (\texttt{iterate}\ f\ x\ !!\ i) = \textsc{False}$ then $\texttt{dountil}\ f\ p\ x\ !!\ k = \texttt{iterate}\ f\ x\ !!\ k$.
- various other results to do with `last`, `iterate` and infinite lists.

We can show that if $(\texttt{dountil}\ f\ p\ x)$ is infinite then $\texttt{last}\ (\texttt{dountil}\ f\ p\ x)$ will be $\bot$ (thus proving a contradiction) and if $(\texttt{dountil}\ f\ p\ x)$ is a specific finite length then i will be that length minus one. □

To prove the final result we construct the function $\texttt{nextWrap}_\mathfrak{s}$ which treats the state of the programs evolution as a 4-tuple containing (1) the world state, (2) the program, (3) the current fresh list of Guess and (4) a Boolean value indicating whether the previous iteration resulted in a world/program pair in normal form.

$$\texttt{nextWrap}_\mathfrak{s}\ ::\ \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho, [\mathsf{Guess}], \mathsf{Bool}) \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho, [\mathsf{Guess}], \mathsf{Bool})$$

Now, with a suitable wrapper, the function

$$\texttt{last}\ (\texttt{dountil}\ (\texttt{nextWrap}_\mathfrak{s}\ c)\ \texttt{fth4}\ (w, m, gs, \textsc{False}))$$

forms the implementation of $\texttt{run}_\mathfrak{s}\ gs\ c\ (w, m)$, where `fth4` returns the fourth element from a 4-tuple.

Proposition 3.3. *If* $w \Vdash m \Downarrow^c w_1 \Vdash m_1$ *and* $w \Vdash m \Downarrow^c w_2 \Vdash m_2$ *then it is true that* $m_1 = m_2$ *and* $w_1 = w_2$

Proof. Obvious, since we have shown that reduction to some normal form defines a function, not just a relation. □

4 The Machine-Verified Confluence Proof

4.1 Introduction

A non-deterministic reduction system is said to be confluent if for a given term all possible reduction sequences eventually yield the same normal form, or no normal form at all. In this section we prove that reduction in Curio is confluent when PRE_s holds – that $\downarrow$ is equivalent to $\Downarrow$. This powerful property means that although our definition of concurrency is a natural, non-deterministic one involving arbitrary choices, this arbitrariness is contained and has no effect on the overall outcome. For us, this means that, for all reduction orders, a program will either always terminate with the same resultant world/program pair or always diverge.

That the confluence proof has been machine-verified is also, on its own, a relatively notable result. Confluence proofs for the λ-calculus have been machine-verified before (in Coq [Hue94], and Isabelle/HOL [Nip96]) but we have yet to see one in an LCF style. Perhaps there is a good reason for this – confluence proofs usually wouldn't require one to prove properties about a *program*. This, however, is the approach we took. We prove that a simple *implementation* of Curio is confluent.

Instructions on the actual implementation and how the machine-readable form of the proofs can be obtained is available in Appendix A.

Terminology

Confluence is also known as the "Church-Rosser" property after the authors of the original proof in 1935 for the λ-calculus [CR36]. Formal definitions of confluence tend to differ slightly depending on which texts are read.

Barendregt, in the standard reference text on the λ-calculus [Bar84], defines a reduction system to be confluent if it obeys the "diamond property". This means that if $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$ then there is some D such that $B \twoheadrightarrow D$ and $C \twoheadrightarrow D$, where $\twoheadrightarrow$ is the reflexive, transitive closure of the reduction relation '$\longrightarrow$'. In Term Rewriting Systems [Ter03], however, the diamond property (and confluence) is defined to be the property "$A \longrightarrow B$ and $A \longrightarrow C$ implies there is some D such that $B \longrightarrow D$ and $C \longrightarrow D$". It is the latter definition of confluence and the diamond property which is closest to what we use, but because our operational semantics also has a notion of failure this muddies the water somewhat.

There is also a certain amount of confusion concerning the differences between (finite) reduction *sequences* and reduction *strategies*. The pure λ-calculus is confluent, yet certain reduction strategies may, for a given term, not result in a normal form when others do. In particular, given the term $(\lambda x.\lambda y.y)\Omega$, call-by-name reduction will find the normal form $(\lambda y.y)$ but call-by-value reduction will not terminate.

Our confluence proof is stronger. All reduction strategies will have the same effect. The intuitive reason why this is true is that unlike function application in the λ-calculus, our `par` construct is highly symmetric.

Overview

The full machine-verified proof of confluence is long and full of complicated details. Many of these relate to the propagation of $\bot$ and the fact that induction over lazy structures must be admissible [Iga74]. We adopt a hybrid approach to describing the proof. We try to explain all the nitty-gritty technical problems encountered, while still never losing sight of the overall picture.

To try to give some structure to the proof, three important subsections of the proof each culminate with the proof of a key theorem:

- Theorem 4.1: If $c_l \diamondsuit_{\mathfrak{s}} c_r$ and $\mathsf{pre}_{\mathfrak{s}}$ then reduction of programs in context c_l and c_r, if possible in both contexts, is order-independent.
- Theorem 4.2: If $\mathsf{PRE}_{\mathfrak{s}}$, then $w \Vdash m \uparrow^c$ implies $w \Vdash m \Uparrow^c$.
- Theorem 4.3 (Confluence): If $\mathsf{PRE}_{\mathfrak{s}}$, then $w \Vdash m \downarrow\!\!\!\downarrow^c w' \Vdash m'$ implies $w \Vdash m \Downarrow^c w' \Vdash m'$.

4.2 Preliminaries

Non-deterministic single-step reduction "$\longrightarrow$" is a fine high-level notation for expressing how a world/program pair can change within a given context. Its downside, however, is that it hides some internal details of the reduction and its implementation, and these details are central to the machine-verified proof.

They include:

- The initial Guess which guides the search for a redex.
- The actions that were performed, if any.
- The whereabouts of the redex if one is eventually found.

The initial Guess is usually "hidden" by an existential quantification, but it is still explicit in the implementation. The other two are truly internal, however, and for this reason the implementation $\mathtt{next}_{\mathfrak{s}}$ had to be rewritten as the interaction of three different functions. (Admissibility states that to obtain information about a lazy structure one must do so constructively and write a function which computes it. One is not able to just prove that it exists.)

$$
\begin{aligned}
\texttt{nextR}_{\mathfrak{s}} &:: \mathsf{Guess} \to \varsigma \to \omega \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Redex}\ (\mathsf{Route}, \mathsf{RxType}\ \alpha) \\
\texttt{advS}_{\mathfrak{s}} &:: \varsigma \to \mathsf{Route} \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho \\
\texttt{advA} &:: \nu \to \mathsf{Route} \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{next}'_{\mathfrak{s}}\ g\ c\ (w, m) \triangleq \texttt{case}\ (\texttt{nextR}_{\mathfrak{s}}(g, c, w, m))\ \texttt{of} \\
&\quad \textsc{NoRedex} \to \textsc{Converged} \\
&\quad \textsc{Redex}\ (r, \textsc{Silent}) \to \textsc{Reduct}\ (w, \texttt{advS}_{\mathfrak{s}}(c, r, m)) \\
&\quad \textsc{Redex}\ (r, \textsc{Action}\ a) \to \texttt{case}\ (\texttt{af}\ a\ w)\ \texttt{of} \\
&\qquad\qquad (w_1, v) \to \textsc{Reduct}\ (w_1, \texttt{advA}(v, r, m))
\end{aligned}
$$

Figure 9: Definition of $\texttt{next}'_{\mathfrak{s}}$

Deconstructing $\texttt{next}_{\mathfrak{s}}$

The new definition of $\texttt{next}_{\mathfrak{s}}$ and the types of the three new functions are in Figure 9.

The function $\texttt{nextR}_{\mathfrak{s}}$ searches for a redex. If it finds one it indicates the Route to that redex and the action it will perform, if any. A Route is just a finite list of L/R values. When searching for a redex, each time a `par` is encountered the next element of the Route indicates whether to look to the left- or right-hand side.

If a redex doesn't perform an action we call it a **silent** redex. The function $\texttt{advA}(v, r, m)$ modifies the action at route r in program m by replacing it with the program `return` v. The program $\texttt{advS}_{\mathfrak{s}}(c, r, m)$ modifies the silent redex at route r in program m. The context information c is required so that a program of the form `test` $a\ m_1\ m_2$ can determine which of the two programs must be executed.

We now prove that the two definitions of $\texttt{next}_{\mathfrak{s}}$ are equivalent, thus allowing us to pick whichever is the more appropriate when proving a lemma.

Lemma 4.1. $\mathit{next}_{\mathfrak{s}} = \mathit{next}'_{\mathfrak{s}}$

Proof. A long but straightforward induction on program structure. We omit the details, but it is a proof that the three individual functions add up to the single original one. $\texttt{nextR}_{\mathfrak{s}}$ does all the searching for a redex but never modifies either the world state or the program. The only reason it needs w at all is to check if an action is stalled. The four stages of the reduction of an action redex `action` a are (1) checking if the action is permitted ($\texttt{ap}\ c\ a = \textsc{True}$), (2) checking if the action is stalled, ($\texttt{wa}\ a\ w = \textsc{False}$) (3) performing the action (evaluating $\texttt{af}\ a\ w$ to some (w_1, v)) and (4) updating the program with the value v. Of these, the first two are performed by $\texttt{nextR}_{\mathfrak{s}}$, the third takes place in the "wiring" and the fourth is performed

$$
\begin{aligned}
w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{a} w_1 \Vdash m_1 &\triangleq \mathtt{nextR_s}(g,c,w,m) = \textsc{Redex}\ (r\,,\textsc{Action}\ a)\ \wedge \\
&\qquad \mathtt{af}\ a\ w = (w_1\,,v)\ \wedge\ m_1 = \mathtt{advA}(v,r,m) \\
w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{\bullet} w_1 \Vdash m_1 &\triangleq \mathtt{nextR_s}(g,c,w,m) = \textsc{Redex}\ (r\,,\textsc{Silent})\ \wedge \\
&\qquad m_1 = \mathtt{advS_s}(c,r,m)\ \wedge\ w = w_1 \\
w \Vdash m\ {}^{g \mapsto r}\!\uparrow^{c}_{a} &\triangleq \mathtt{nextR_s}(g,c,w,m) = \textsc{Redex}\ (r\,,\textsc{Action}\ a)\ \wedge \\
&\qquad \mathtt{af}\ a\ w = \bot \\
w \Vdash m\ {}^{g \mapsto \bot}\!\uparrow^{c} &\triangleq \mathtt{nextR_s}(g,c,w,m) = \bot \\
w \Vdash m \downarrow^{c} &\triangleq \mathtt{nextR_s}(g,c,w,m) = \textsc{NoRedex}
\end{aligned}
$$

Figure 10: Annotating single-step reduction

by `advA`. With any silent redex, $\mathtt{nextR_s}$ just finds the redex returning the route, and $\mathtt{advS_s}$ modifies the program itself. □

Annotating single-step reduction

Redexes are inherently slippery things to reason about since there is no obvious type or set which is isomorphic to them in any useful way. We therefore use Guesses to *quantify* over redexes when we're looking for one and Routes to *identify* a redex when we have found one. The function from Guesses to redexes is onto; the function from redexes to Routes is one-to-one.

Guesses and Routes can be confusingly similar, at times. Although they are both implemented as lists of Bool, they are conceptually somewhat different. A Guess is best understood as always being infinite, and Routes as always being finite. So we can always successfully retrieve the head and tail of a Guess, whereas there is the possibility that a Route is empty. Also, we sometimes implicitly "cast" a Route to a Guess by padding the finite list to make it infinite.

The new annotated single-step reduction can be found in Figure 10. It should be clear that the new notation covers all possible cases and is a consistent extension to the old notation. If the guess, route or redex type is omitted this means there is an implicit existential quantification. $w \Vdash m \longrightarrow^{c}_{a} w' \Vdash m'$ means there exists some Guess g and a route Route r such that $w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{a} w' \Vdash m'$. If an action a or $\bullet$ is omitted from a reduction then we just don't specify whether it was silent or performed an action.

Corollary 4.2. *If* $w \Vdash m \longrightarrow^{c}_{\bullet} w_1 \Vdash m_1$ *then* $w = w_1$.

Proof. Immediate. □

Corollary 4.3. *If* $w \Vdash m \longrightarrow_a^c w_1 \Vdash m_1$ *then* $\mathtt{af}\ a\ w = (w_1, v)$*, for some* v*.*

Proof. Immediate. □

Corollary 4.4. *If* $w \Vdash m\ ^{g \mapsto r}\!\uparrow_a^c$ *then* $\mathtt{af}\ a\ w = \bot$*.*

Proof. Immediate. □

A template for inductive proofs

A great many lemmas are proved by inducting over the recursive structure of Prog $\nu\ \alpha\ \rho$. They are all, of course, different, but we can give a general shape to many of the proofs, and this will serve as a basic template.

Base programs (those of the form `return` v, $\bot$, `return` v `>>=` f, `action` a, `test` $a\ m_t\ m_f$ or `return` v_l `|||`$^p_*$ `return` v_r) are always entirely deterministic for a given context and world, and never requires any "deeper" knowledge, such as an inductive hypothesis. When inducting over Prog $\nu\ \alpha\ \rho$, it is usually relatively easy to prove properties for these programs.

The behaviour of a program m `>>=` f, where m isn't a value `return` v, is solely determined by the behaviour of m. This is importance because when inducting over Prog $\nu\ \alpha\ \rho$ no inductive hypothesis can be given for f. If m fails, diverges or is in normal form then the same will be true of m `>>=` f.

Programs of the form m_l `|||`$^p_*$ m_r are usually the most troublesome to prove properties about. If m_l and m_r are both values then it is a base term. It always fails if `pf` $p\ c = \bot$, $m_l = \bot$ or $m_r = \bot$, so we often just omit this simple case altogether. If neither of the above are true an inductive hypothesis will be needed for m_l or m_r (and occasionally both) and how they behave in their respective contexts. Proving lemmas inductively for programs like this is made easier if we can perform a simple case analysis on whether the left- or right-hand side was reduced. The three following lemmas show how by examining the resultant Route we can learn whether a redex lied on the left or right hand side. Doing case analysis on the Guess would not give this sort of information. This is our standard procedure for determining the side in which a reduction took place.

Lemma 4.5. *If* $w \Vdash m_l$ `|||`$^p_*$ $m_r \xrightarrow{g \mapsto []}_c w_1 \Vdash m_1$ *then* $\mathtt{pf}\ p\ c = (c_l, c_r)$ *for some* c_l*,*c_r *and for some* v_l*,*v_r*,* $m_l = \mathtt{return}\ v_l$*,* $m_r = \mathtt{return}\ v_r$*,* $w_1 = w$ *and* $m_1 = \mathtt{return}\ v_l * v_r$*.*

Proof. Since the route is [] the redex cannot be within either m_l or m_r. Therefore m_l and m_r must be values. □

Lemma 4.6. *If* m *is not a value,* $\mathtt{nextR}_5(g, c, w, m\ \texttt{>>=}\ f) = \mathtt{nextR}_5(g, c, w, m)$*.*

Proof. Immediate from implementation. □

Lemma 4.7. *If m is not a value, then $w \Vdash m \texttt{ >>= } f \xrightarrow{g \mapsto r}_c w' \Vdash m' \texttt{ >>= } f$ if and only if $w \Vdash m \xrightarrow{g \mapsto r}_c w' \Vdash m'$.*

Proof. Immediate from Lemma 4.6. □

Lemma 4.8. *If $\mathtt{nextR_s}([b{:}g], c, w, m_l \texttt{ /|/}^p_* m_r) = \textsc{Redex}\ (L{:}r, x)$ then $\mathtt{pf}\ p\ c = (c_l, c_r)$ for some c_l,c_r and $\mathtt{nextR_s}(g, c_l, w, m_l) = \textsc{Redex}\ (r, x)$.*

Proof. Regardless of the initial direction b of the Guess, if the resultant Route was L then the left-hand side will have been reduced. □

Lemma 4.9. *If $\mathtt{nextR_s}([b{:}g], c, w, m_l \texttt{ /|/}^p_* m_r) = \textsc{Redex}\ (R{:}r, x)$ then $\mathtt{pf}\ p\ c = (c_l, c_r)$ for some c_l,c_r and $\mathtt{nextR_s}(g, c_r, w, m_r) = \textsc{Redex}\ (r, x)$.*

Proof. Symmetric to proof of Lemma 4.9. □

Lemma 4.10. *If $w \Vdash m_l \texttt{ /|/}^p_* m_r \xrightarrow{[b{:}g] \mapsto [L{:}r]}_c w_1 \Vdash m_1$ then $\mathtt{pf}\ p\ c = (c_l, c_r)$ for some c_l,c_r and there is a m'_l such that $w \Vdash m_l \xrightarrow{g \mapsto r}_{c_l} w_1 \Vdash m'_l$ and $m_1 = m'_l \texttt{ /|/}^p_* m_r$.*

Proof. Immediate from Lemma 4.9. □

Lemma 4.11. *If $w \Vdash m_l \texttt{ /|/}^p_* m_r \xrightarrow{[b{:}g] \mapsto [R{:}r]}_c w_1 \Vdash m_1$ then $\mathtt{pf}\ p\ c = (c_l, c_r)$ for some c_l,c_r and there is a m'_r such that $w \Vdash m_r \xrightarrow{g \mapsto r}_{c_r} w_1 \Vdash m'_r$ and $m_1 = m_l \texttt{ /|/}^p_* m'_r$.*

Proof. Immediate from Lemma 4.9. □

4.3 Initial results

Having established some new notation, in this subsection we are now ready to begin the confluence proof. We prove some important lemmas relating to how single-step reduction affects world state and what actions single-step reduction can perform.

Lemma 4.12. *If $\mathtt{nextR_s}(g, c, w, m) = \textsc{NoRedex}$ then for all g_1, it is true that $\mathtt{nextR_s}(g_1, c, w, m) = \textsc{NoRedex}$.*

Proof. Induction on m. It is trivial that `return` v is always in normal form, and if $w \Vdash$ `action` a is in normal form it always will be. $m_l \texttt{ |||}^p_* m_r$ is only in normal form if m_l and m_r are both in normal form and it's not the case that both are values. g_1 can be either $[L{:}g'_1]$ or $[R{:}g'_1]$, for some g'_1, but in each case m_l and m_r will both be in normal form, regardless of g'_1's value (IH), so the same will be true of $m_l \texttt{ |||}^p_* m_r$. □

Lemma 4.13. *If $w \Vdash m \downarrow^c$ then it is not the case that $w \Vdash m \longrightarrow^c w' \Vdash m'$ for some w',m' or that $w \Vdash m \uparrow^c$. (This is a proof of confluence for programs which require no reduction steps.)*

Proof. Immediate from Lemma 4.12. □

Lemma 4.14. *If $\mathit{nextR}_{\mathfrak{s}}(g, c, w, m) = \text{REDEX } (r, \text{ACTION } a)$ then $\mathtt{wa}\ a\ w =$ FALSE.*

Proof. Induction on m. The only base redex which isn't silent is `action` a, and if this reduces then `wa` $a\ w =$ FALSE. This is true, by induction, for any reduction which performs an action. □

Lemma 4.15. *If $w \Vdash m \longrightarrow_a^c w' \Vdash m'$ then $\mathtt{wa}\ a\ w =$ FALSE.*

Proof. Immediate from Lemma 4.14. □

Lemma 4.16. *If $\mathsf{pre}_{\mathfrak{s}}$ and $\mathit{nextR}_{\mathfrak{s}}(g, c, w, m) = \text{REDEX } (r, \text{ACTION } a)$ then $\mathtt{ap}\ c\ a =$ TRUE.*

Proof. Induction on m. Trivial for `action` a, and true by contradiction for silent base redexes. If $m = m_1$ `>>=` f (m not a value), it is trivial from Lemma 4.6. If m is m_l `|||`$^p_*$ m_r then do case analysis on the resultant route. If it is true that $\mathtt{nextR}_{\mathfrak{s}}([b{:}g], c, w, m_l \ \mathtt{|||}^p_*\ m_r) = \text{REDEX } (\mathtt{[}L{:}r\mathtt{]}, x)$, then `pf` $p\ c = (c_l, c_r)$ for some c_l,c_r, and from Lemma 4.8, $\mathtt{nextR}_{\mathfrak{s}}(g, c_l, w, m_l) =$ REDEX (r, x). From IH, conclude `ap` $c_l\ a =$ TRUE, and by $\mathsf{pre}_{\mathfrak{s}}$, $c_l \sqsubseteq_{\mathfrak{s}} c$, and therefore `ap` $c\ a =$ TRUE. The proof for the right-hand side is symmetric. □

Lemma 4.17. *If $\mathsf{pre}_{\mathfrak{s}}$ and $w \Vdash m \longrightarrow_a^c w' \Vdash m'$ then $\mathtt{ap}\ c\ a =$ TRUE.*

Proof. Immediate from Lemma 4.16. □

The following lemmas relate successful single-step reduction of m on world w to reduction in a completely unrelated world w'. The key is that after reducing successfully we will have the correct route to that redex. The second time around this route is used as the new guess to guarantee that the same redex is actually found – some actions in m which were stalled in w (and therefore ignored) may have become unstalled in w', and it necessary to supply the exact route to make sure that we never encounter these unstalled actions, or any other redex.

Lemma 4.18. *If $\mathit{nextR}_{\mathfrak{s}}(g, c, w, m) = \text{REDEX } (r, \text{SILENT})$ then for all w', $\mathit{nextR}_{\mathfrak{s}}(g, c, w', m) = \text{REDEX } (r, \text{SILENT})$.*

Proof. Induction on m. Trivially true for any silent base redex, since they are deterministic. If $m = m_l \texttt{ ||| }^p_* m_r$ and $g = [b{:}g']$, then do case analysis on the route r. If $r = []$, it is a base redex. If $r = [L{:}r']$ then we can derive $\texttt{nextR}_\mathfrak{s}(g', c_l, w, m_l) = \text{REDEX } (r', \text{SILENT})$ with Lemma 4.8. With IH, prove $\texttt{nextR}_\mathfrak{s}(r', c_l, w', m_l) = \text{REDEX } (r', \text{SILENT})$, and therefore $\texttt{nextR}_\mathfrak{s}(r, c, w', m_l \texttt{ ||| }^p_* m_r) = \text{REDEX } (r, \text{SILENT})$. If the right-hand side is reduced, the proof is similar. Programs of the form $m_1 \texttt{ >>= } f$ are proved easily with induction. □

Lemma 4.19. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}^c_\bullet w \Vdash m'$ *then for all* w', $w' \Vdash m \xrightarrow{r \mapsto r}{}^c_\bullet w' \Vdash m'$.

Proof. Immediate, from Lemma 4.18. □

Lemma 4.20. *If* $\texttt{nextR}_\mathfrak{s}(g, c, w, m) = \text{REDEX } (r, \text{ACTION } a)$ *then for all* w_1, *if* $\texttt{wa}\ a\ w_1 = \text{FALSE}$ *then* $\texttt{nextR}_\mathfrak{s}(r, c, w_1, m) = \text{REDEX } (r, \text{ACTION } a)$.

Proof. Induction on m. Similar to the proof of Lemma 4.18 except the only valid base program is `action` a – all the others reduce silently. Since the action was already performed successfully in the action's local context c_1 which won't have changed, we know that $\texttt{ap}\ c_1\ a = \text{TRUE}$, and because we know $\texttt{wa}\ a\ w_1 = \text{FALSE}$ that action will definitely be returned as a legitimate, unstalled redex. □

Lemma 4.21. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}^c_a w' \Vdash m'$, *where* $\texttt{af}\ a\ w = (w', v)$, *then for all* w_1, *if* $\texttt{wa}\ a\ w_1 = \text{FALSE}$ *and* $\texttt{af}\ a\ w_1 = (w_1', v)$, *then* $w_1 \Vdash m \xrightarrow{r \mapsto r}{}^c_a w_1' \Vdash m'$.

Proof. Immediate, using Lemma 4.20. □

Lemma 4.22. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}^c_a w' \Vdash m'$, *then for all* w_1, *if* $\texttt{wa}\ a\ w_1 = \text{FALSE}$ *and* $\texttt{af}\ a\ w_1 = \bot$, *then* $w_1 \Vdash m\ ^{r \mapsto r}\!\uparrow^c_a$.

Proof. Immediate, using Lemma 4.20. □

Lemma 4.23. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}_c w' \Vdash m'$, *then* $w \Vdash m \xrightarrow{r \mapsto r}{}_c w' \Vdash m'$.

Proof. If the reduction is silent, apply Lemma 4.19. If it performs an action a, apply Lemma 4.21 – we know from Lemma 4.15 that $\texttt{wa}\ a\ w = \text{FALSE}$. □

Lemma 4.24. *If* $w \Vdash m\ ^{g \mapsto r}\!\uparrow^c_a$, *then* $w \Vdash m\ ^{r \mapsto r}\!\uparrow^c_a$.

Proof. Apply Lemma 4.22 – we know from Lemma 4.14 that $\texttt{wa}\ a\ w = \text{FALSE}$. □

We can now prove that reduction in disjoint contexts is order independent.

Theorem 4.1. *Assuming* $\mathsf{pre}_\mathfrak{s}$ *and* $c_l \lozenge_\mathfrak{s} c_r$, *then if both* $w \Vdash m_l \xrightarrow{g_l \mapsto r_l}{}_{c_l} w_l \Vdash m_l'$ *and* $w \Vdash m_r \xrightarrow{g_r \mapsto r_r}{}_{c_r} w_r \Vdash m_r'$, *then either*

- *there exists some* w_2 *such that both*

$$w_r \Vdash m_l \xrightarrow{r_l \mapsto r_l} c_l \; w_2 \Vdash m_l'$$

and

$$w_l \Vdash m_r \xrightarrow{r_r \mapsto r_r} c_r \; w_2 \Vdash m_r'$$

- *or it is the case that both* $w_r \Vdash m_l \overset{r_l \mapsto r_l}{\uparrow^{c_l}}$ *and* $w_l \Vdash m_r \overset{r_r \mapsto r_r}{\uparrow^{c_r}}$.

Proof. Case analysis on whether a reduction is silent.

- One is silent, say that of c_l: $w_l = w$, from Corollary 4.2, and both succeed. Let $w_2 = w_r$. Use Lemma 4.19 to prove that m_l still reduces to m_l' with world state w_r, and use Lemma 4.23 to prove that m_r will behave the same on world w with r_r as its guess instead of g_r. (The proof is symmetric if c_r's redex is silent.)

- Both are actions, say a_l and a_r: This means (Corollory 4.3) that $\texttt{af}\ a_l\ w = (w_l, v_l)$ and $\texttt{af}\ a_r\ w = (w_r, v_r)$. From Lemma 4.17 we know $\texttt{ap}\ c_l\ a_l = \textsc{True}$ and $\texttt{ap}\ c_r\ a_r = \textsc{True}$, and from Lemma 4.15, we also know that $\texttt{wa}\ a_l\ w = \textsc{False}$ and $\texttt{wa}\ a_r\ w = \textsc{False}$. Now, since $c_l \diamondsuit_{\mathfrak{s}} c_r$ holds this means $a_l \mid\mid\mid_{\mathfrak{s}} a_r$, $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$ and $\mathsf{ally}_{\mathfrak{s}}(a_r, a_l)$ are true. Using $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$ prove $\texttt{wa}\ a_r\ w_l = \textsc{False}$, using $\mathsf{ally}_{\mathfrak{s}}(a_r, a_l)$ prove $\texttt{wa}\ a_l\ w_r = \textsc{False}$, and because $c_l \mid\mid\mid_{\mathfrak{s}} c_r$, either:

  - There exists a w_2 such that $\texttt{af}\ a_l\ w_r = (w_2, v_l)$ and $\texttt{af}\ a_r\ w_l = (w_2, v_r)$. Apply Lemma 4.21 to both sides to prove that they both will succeed.
  - $\texttt{af}\ a_l\ w_r = \bot$ and $\texttt{af}\ a_r\ w_l = \bot$. Apply Lemma 4.22 to show they both fail.

□

4.4 Analysing Failure

The next important theorem is that given $\mathsf{PRE}_{\mathfrak{s}}$, if $w \Vdash m \uparrow^c$ then $w \Vdash m \Uparrow^c$. This is effectively a proof that denotational failure in the metalanguage always causes divergence in our language – if a single-step reduction can possibly fail then despite nondeterminism it is impossible that the program will ever converge to a normal form.

In previous subsections we defined two distinct types of failure:

- $w \Vdash m \overset{g \mapsto r}{\uparrow^c_a}$. A redex is found, $\texttt{action}\ a$, but the action a fails.
- $w \Vdash m \overset{g \mapsto \bot}{\uparrow^c}$. The search for a redex fails outright.

$$
\begin{array}{rcl}
\mathsf{Tree}\ \beta & = & \textsc{Branch}\ (\mathsf{Tree}\ \beta)\ (\mathsf{Tree}\ \beta) \mid \textsc{Leaf}\ \beta \\
\mathtt{redexTree_s} & :: & \varsigma \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Tree}\ (\mathsf{Redex}\ (\mathsf{Route}, (\varsigma, \mathsf{RxType}\ \alpha))) \\
\mathtt{redexList_s} & :: & \mathsf{Guess} \to \varsigma \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to [(\mathsf{Redex}\ (\mathsf{Route}, (\varsigma, \mathsf{RxType}\ \alpha)))] \\
\mathtt{check_s} & :: & \omega \to \mathsf{Redex}\ (\mathsf{Route}, (\varsigma, \mathsf{RxType}\ \alpha)) \to \mathsf{Redex}\ (\mathsf{Route}, \mathsf{RxType}\ \alpha) \\
\mathtt{shuffle} & :: & \mathsf{Guess} \to \mathsf{Tree}\ \beta \to \mathsf{Tree}\ \beta \\
\mathtt{preorder} & :: & \mathsf{Tree}\ \beta \to [\beta] \\
\mathtt{mapTree} & :: & (\beta \to \gamma) \to \mathsf{Tree}\ \beta \to \mathsf{Tree}\ \gamma \\
\mathtt{firstRx} & :: & \mathsf{Redex}\ \beta \to \mathsf{Redex}\ \beta \to \mathsf{Redex}\ \beta \\
\mathtt{addL}, \mathtt{addR} & :: & \mathsf{Redex}\ (\mathsf{Route}, \beta) \to \mathsf{Redex}\ (\mathsf{Route}, \beta)
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{nextR'_s}(g, c, w, m) & \triangleq & \mathtt{firstRxW_s}\ w\ (\mathtt{redexList_s}\ g\ c\ m) \\
\mathtt{firstRxW_s}\ w & \triangleq & \mathtt{foldl}\ \mathtt{firstRx}\ \textsc{NoRedex} \circ \mathtt{map}\ (\mathtt{check_s}\ w) \\
\mathtt{redexList_s}\ g\ c\ m & \triangleq & \mathtt{preorder}\ \$\ \mathtt{shuffle}\ g\ \$\ \mathtt{redexTree_s}\ c\ m
\end{array}
$$

Figure 11: Definition of $\mathtt{nextR_s}$

If a program fails in the first way, then proving it will diverge is relatively easy (Lemma 4.38). The second, however, is much more difficult since it forces us, like with $\mathtt{next_s}$, to rewrite $\mathtt{nextR_s}$, breaking the process of searching for a redex into more manageable chunks. This is required to find out exactly how and why searching for a redex might fail.

Deconstructing $\mathtt{nextR_s}$

There are obvious tensions at play when trying to re-implement $\mathtt{nextR_s}$. A tree structure perfectly describes the structure of redexes: a leaf node represents a definite redex or lack thereof, and a Branch indicates two concurrent programs, each of which may have their own redexes. Furthermore, the Guess used to coordinate the search for a redex is most "naturally" applied to a tree structure. On the other hand, we must search for a redex sequentially. A list is ideal for this, since each element in a list is indexed uniquely and sequentially by integers in a way that elements of a tree cannot be.

Our solution involved the construction of two temporary structures, a tree and a list, before the final redex is returned. By doing this we can then make use of existing theorems on lists and trees. An outline of the new function $\mathtt{nextR'_s}$ is given in Figure 11.

The first half is the function $\mathtt{redexList_s}$. This constructs a lazy list of potential redexes and has three parts. $\mathtt{redexTree_s}\ c\ m$ first builds a tree in which each leaf-node represents either NoRedex, indicating that no

reduction can take place, or REDEX $(r,(c_1,x))$, indicating that there is a potential redex at Route r, where c_1 is the local context for that redex. x can be either SILENT or ACTION a for some a. There are are only two situations where NOREDEX will occur in the tree: (1) if m is a value at the top-level, and (2) if a sub-program of m is of the form $\texttt{return}\ v_l\ |||^p_*\ m_r$ or $m_l\ |||^p_*\ \texttt{return}\ v_r$ and m_r/m_l aren't values. The function $\texttt{shuffle}\ g$ then rearranges the tree according to Guess g so that the left-to-right ordering of leaves respects the order in which $\texttt{nextR}_\mathfrak{s}$ originally would have scanned for redexes. Pre-ordering this tree with `preorder` then results in a list of potential redexes in the correct order.

$\texttt{firstRxW}_\mathfrak{s}$ is the second half and finds the first legitimate redex in this list. It has two parts. Mapping $\texttt{check}_\mathfrak{s}\ w$ across the list of potential redexes throws out any action redexes which (1) aren't permitted by their local context or (2) are permitted but are stalled in the current world w. Actions which aren't allowed result in $\bot$ ($\texttt{nextR}_\mathfrak{s}$ should only return valid actions – Lemma 4.16) but permitted actions which are currently stalled become NOREDEX. Having eliminated all spurious redexes from the list, we can simply scan the list looking for the first element which isn't NOREDEX. Folding `firstRx` across the list does exactly that.

The types of a few other internal functions are given in Figure 11. The function `mapTree` modifies each element of a tree with a given function and `addL` and `addR` prepend either a L or a R to the route contained within a redex.

It is useful to note that this re-implementation manages to separate the four arguments of $\texttt{nextR}'_\mathfrak{s}$ into three different sequential transformations. The context c and program m are supplied to $\texttt{redexTree}_\mathfrak{s}$, the tree is shuffled with the Guess g, and the world value w is only needed by $\texttt{check}_\mathfrak{s}$ to see if actions are stalled.

Unlike $\texttt{nextR}_\mathfrak{s}$, $\texttt{nextR}'_\mathfrak{s}$ is not defined directly in a recursive manner. We therefore need to prove that a direct recursive relationship exists. The following lemmas are all required to show that the behaviour of $\texttt{nextR}'_\mathfrak{s}$ for some program can be understood in terms of its behaviour for the sub-programs of that program.

Lemma 4.25.

$$
\begin{array}{lrcl}
(i) & \mathit{length} \circ \mathit{preorder} & = & \mathit{length} \circ \mathit{preorder} \circ \mathit{shuffle}\ g \\
(ii) & \mathit{shuffle}\ g \circ \mathit{shuffle}\ g & = & \mathit{id} \\
(iii) & \mathit{shuffle}\ g_1 \circ \mathit{shuffle}\ g_2 & = & \mathit{shuffle}\ g_2 \circ \mathit{shuffle}\ g_1 \\
(iv) & \mathit{shuffle}\ [L, L, ...] & = & \mathit{id} \\
(v) & \mathit{shuffle}\ g \circ \mathit{mapTree}\ f & = & \mathit{mapTree}\ f \circ \mathit{shuffle}\ g \\
(vi) & \mathit{map}\ f \circ \mathit{preorder} & = & \mathit{preorder} \circ \mathit{mapTree}\ f
\end{array}
$$

$$(vii) \quad (r1\ `\mathit{firstRx}`\ r2)\ `\mathit{firstRx}`\ r3 = r1\ `\mathit{firstRx}`\ (r2\ `\mathit{firstRx}`\ r3)$$

$$(viii) \quad \mathit{firstRxW}_\mathfrak{s}\ w\ (r_1 +\!\!\!+ r_2) = \mathit{firstRx}\ (\mathit{firstRxW}_\mathfrak{s}\ w\ r_1)\ (\mathit{firstRxW}_\mathfrak{s}\ w\ r_2)$$

$$
\begin{array}{lrcl}
(ix) & r & = & r \text{ `}\mathit{firstRx}\text{`} \textsc{NoRedex} \\
(x) & r & = & \textsc{NoRedex} \text{ `}\mathit{firstRx}\text{`} r \\
(xi) & \mathit{check}_{\mathfrak{s}}\ \omega \circ \mathit{addL} & = & \mathit{addL} \circ \mathit{check}_{\mathfrak{s}}\ \omega \\
(xii) & \mathit{check}_{\mathfrak{s}}\ \omega \circ \mathit{addR} & = & \mathit{addR} \circ \mathit{check}_{\mathfrak{s}}\ \omega \\
(xiii) & \mathit{firstRxW}_{\mathfrak{s}}\ w \circ \mathit{map}\ \mathit{addL} & = & \mathit{addL} \circ \mathit{firstRxW}_{\mathfrak{s}}\ w \\
(xiv) & \mathit{firstRxW}_{\mathfrak{s}}\ w \circ \mathit{map}\ \mathit{addR} & = & \mathit{addR} \circ \mathit{firstRxW}_{\mathfrak{s}}\ w
\end{array}
$$

Proof. All by straightforward induction or case analysis. □

Result (i) in Lemma 4.25 is a proof that shuffling a redex tree does not change the number of elements in that tree.

The results (ii)-(iv) in Lemma 4.25 show that `shuffle` g, for all g, forms the elements of an Abelian (or commutative) group under the $\circ$ operator. Function composition is always associative, the identity is `shuffle` $[L, L, ...]$ and each element is its own inverse.

Results (vii), (ix) and (x) show that `firstRx` is a monoidal operator with identity NoRedex. The final four results show that $\texttt{firstRxW}_{\mathfrak{s}}$ and `check` do not query the route at which a redex exists. Instead it is passed through these functions unchanged.

Lemma 4.26. *If m is not a value then* $\mathit{redexList}_{\mathfrak{s}}\ g\ c\ (m \texttt{ >>= } f) = \mathit{redexList}_{\mathfrak{s}}\ g\ c\ m$.

Proof. Immediate. □

Lemma 4.27. *If m_l, m_r are not both values and* $\mathit{pf}\ p\ c = (c_l, c_r)$ *then*

$$
\begin{array}{c}
\mathit{redexList}_{\mathfrak{s}}\ [L{:}g]\ c\ (m_l\ \texttt{|||}^p_*\ m_r) = \\
\mathit{map}\ \mathit{addL}\ (\mathit{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l) +\!\!+ \mathit{map}\ \mathit{addR}\ (\mathit{redexList}_{\mathfrak{s}}\ g\ c_r\ m_r)
\end{array}
$$

Proof. Equational.

$$
\begin{array}{c}
\texttt{redexList}_{\mathfrak{s}}\ [L{:}g]\ c\ m_l\ \texttt{|||}^p_*\ m_r \\
= (\texttt{redexList}_{\mathfrak{s}}\text{, defn.}) \\
\texttt{preorder \$ shuffle}\ [L{:}g]\ \texttt{\$}\ \texttt{redexTree}_{\mathfrak{s}}\ c\ (m_l\ \texttt{|||}^p_*\ m_r) \\
= (\texttt{redexTree}_{\mathfrak{s}}\text{ defn.}) \\
\texttt{preorder \$ shuffle}\ [L{:}g]\ \texttt{\$ Branch (mapTree addL \$}\ \texttt{redexTree}_{\mathfrak{s}}\ c_l\ m_l)\ (\cdots) \\
= (\texttt{shuffle}\text{ defn.}) \\
\texttt{preorder \$ Branch (shuffle}\ g\ \texttt{\$ mapTree addL \$}\ \texttt{redexTree}_{\mathfrak{s}}\ c_l\ m_l)\ (\cdots) \\
= (\texttt{preorder}\text{ defn.}) \\
(\texttt{preorder \$ shuffle}\ g\ \texttt{\$ mapTree addL \$}\ \texttt{redexTree}_{\mathfrak{s}}\ c_l\ m_l)\ +\!\!+\ (\cdots) \\
= (\text{Lemma 4.25, (v)}) \\
(\texttt{preorder \$ mapTree addL \$ shuffle}\ g\ \texttt{\$}\ \texttt{redexTree}_{\mathfrak{s}}\ c_l\ m_l)\ +\!\!+\ (\cdots) \\
= (\text{Lemma 4.25, (vi)}) \\
(\texttt{map addL \$ preorder \$ shuffle}\ g\ \texttt{\$}\ \texttt{redexTree}_{\mathfrak{s}}\ c_l\ m_l)\ +\!\!+\ (\cdots) \\
= (\texttt{redexList}_{\mathfrak{s}}\text{, defn.}) \\
\texttt{map addL}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l) +\!\!+ \texttt{map addR}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_r\ m_r)
\end{array}
$$

□

Lemma 4.28. *If m_l, m_r are not both values and* `pf` $p\ c = (c_l, c_r)$ *then*

$$\texttt{redexList}_{\mathfrak{s}}\ [R{:}g]\ c\ (m_l\ \texttt{|||}^p_*\ m_r) = \texttt{map addR}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_r\ m_r) +\!\!+ \texttt{map addL}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l)$$

Proof. Symmetric to that of Lemma 4.27. □

Lemma 4.29. *If m is not a value then* $\texttt{nextR}'_{\mathfrak{s}}(g, c, w, m\ \texttt{>>=}\ f) = \texttt{nextR}'_{\mathfrak{s}}(g, c, w, m)$

Proof. Immediate from Lemma 4.26 □

Lemma 4.30. *If* `pf` $p\ c = (c_l, c_r)$ *and m_l,m_r are not both values then*

$$\texttt{nextR}'_{\mathfrak{s}}([L{:}g], c, w, m_l\ \texttt{|||}^p_*\ m_r) = \texttt{firstRx}\ (\texttt{addL}\ (\texttt{nextR}'_{\mathfrak{s}}(g, c_l, w, m_l)))\ (\texttt{addR}\ (\texttt{nextR}'_{\mathfrak{s}}(g, c_r, w, m_r)))$$

This states that if the the initial Guess *says to look on the left-hand side first, then do*

Proof. Equational.

$$\begin{array}{c}
\texttt{nextR}'_{\mathfrak{s}}([L{:}g], c, w, m_l\ \texttt{|||}^p_*\ m_r) \\
= (\texttt{nextR}'_{\mathfrak{s}}\ \text{defn.}) \\
\texttt{firstRxW}_{\mathfrak{s}}\ w\ (\texttt{redexList}_{\mathfrak{s}}\ [L{:}g]\ c\ (m_l\ \texttt{|||}^p_*\ m_r)) \\
= (\text{Lemma 4.27}) \\
\texttt{firstRxW}_{\mathfrak{s}}\ w\ (\texttt{map addL}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l) +\!\!+ \texttt{map addR}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_r\ m_r)) \\
= (\text{Lemma } \texttt{firstRxW}_{\mathfrak{s}}\ w\ (rs_1 +\!\!+ rs_2)) \\
\texttt{firstRx}\ (\texttt{firstRxW}_{\mathfrak{s}}\ w\ (\texttt{map addL}\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l)))\ (\cdots) \\
= (\text{Lemma 4.25, (xiii) and (xiv)}) \\
\texttt{firstRx}\ (\texttt{addL}\ (\texttt{firstRxW}_{\mathfrak{s}}\ w\ (\texttt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l)))\ (\texttt{addR}\ (\cdots)) \\
= (\texttt{nextR}'_{\mathfrak{s}}\ \text{defn.}) \\
\texttt{firstRx}\ (\texttt{addL}\ (\texttt{nextR}'_{\mathfrak{s}}(g, c_l, w, m_l)))\ (\texttt{addR}\ (\texttt{nextR}'_{\mathfrak{s}}(g, c_r, w, m_r)))
\end{array}$$

□

Lemma 4.31. *If* `pf` $p\ c = (c_l, c_r)$ *and m_l,m_r are not both values then*

$$\texttt{nextR}'_{\mathfrak{s}}([R{:}g], c, w, m_l\ \texttt{|||}^p_*\ m_r) = \texttt{firstRx}\ (\texttt{addR}\ (\texttt{nextR}'_{\mathfrak{s}}(g, c_r, w, m_r)))\ (\texttt{addL}\ (\texttt{nextR}'_{\mathfrak{s}}(g, c_l, w, m_l)))$$

Proof. Similar to proof of Lemma 4.30. □

Lemma 4.32. $\texttt{nextR}_{\mathfrak{s}} = \texttt{nextR}'_{\mathfrak{s}}$.

Proof. Induction on program structure. All base terms are deterministic, so the Guess is irrelevant and it is a simple matter of showing that $\texttt{check}_{\mathfrak{s}}$ and $\texttt{redexTree}_{\mathfrak{s}}$ together behave the same as $\texttt{nextR}_{\mathfrak{s}}$. If the program is of the form m `>>=` f, m not a value, then Lemma 4.6 and Lemma 4.26 together convert $\texttt{nextR}_{\mathfrak{s}}$ and $\texttt{nextR}'_{\mathfrak{s}}$ respectively to the form of the IH. If the program

$$\mathsf{wf}_{\mathfrak{s}} : \varsigma \to (\mathsf{Prog}\ \nu\ \alpha\ \rho) \to \mathbb{B}$$

$$\mathsf{wf}_{\mathfrak{s}}(c, m) \triangleq \mathtt{length}\ (\mathtt{preorder}\ (\mathtt{redexTree}_{\mathfrak{s}}\ c\ m)) \neq \bot$$

$$\mathsf{wf}_{\mathfrak{s}}(c, \mathtt{return}\ v)\ \ \mathsf{wf}_{\mathfrak{s}}(c, \mathtt{action}\ a)\ \ \mathsf{wf}_{\mathfrak{s}}(c, \mathtt{test}\ a\ m_t\ m_f)\ \ \neg\mathsf{wf}_{\mathfrak{s}}(c, \bot)$$

$$\mathsf{wf}_{\mathfrak{s}}(c, m\ \mathtt{>>=}\ f) \iff \mathsf{wf}_{\mathfrak{s}}(c, m)$$

$$\text{if } \mathtt{pf}\ p\ c = \bot, \text{ then } \neg\mathsf{wf}_{\mathfrak{s}}(c, m_l\ \mathtt{|||}^p_*\ m_r)$$

$$\text{if } \mathtt{pf}\ p\ c = \mathtt{(}c_l\mathtt{,}c_r\mathtt{)}, \text{ then } \mathsf{wf}_{\mathfrak{s}}(c, m_l\ \mathtt{|||}^p_*\ m_r) \iff (\mathsf{wf}_{\mathfrak{s}}(c_l, m_l) \wedge \mathsf{wf}_{\mathfrak{s}}(c_r, m_r))$$

Figure 12: Well-Formedness of Programs

is $m_l\ \mathtt{|||}^p_*\ m_r$, m_l and m_r not both values, then depending on whether the Guess is of the form $[L{:}g]$ or $[R{:}g]$ use Lemma 4.27 or Lemma 4.28. With the IH for m_l and m_r one can then show that $\mathtt{nextR}'_{\mathfrak{s}}$ preserves (1) the order in which the implementation of $\mathtt{nextR}_{\mathfrak{s}}$ searches for redexes and (2) how the resultant route, if there is one, has either L or R prepended to it depending on the location of the redex. □

Well-formedness of Programs

For the divergence proof we found it necessary to define a notion of a well-formed program with respect to some context. The purpose of this is to distinguish programs which fail as a result of their structure (programs which aren't well-formed) and those that fail because of actions and how they interact with world state (programs that are well-formed.) The key to the introduction of well-formedness is the proof that if a program is not well-formed then it will always diverge, *even though it's possible that it may never fail after a finite number of steps.* (Failure, as always, means immediate failure, ↑.)

If a program m is well-formed with respect to context c then this is written as $\mathsf{wf}_{\mathfrak{s}}(c, m)$. The definition and some basic properties of well-formedness is given in Figure 12 – a program is well-formed if the list of potential redexes returned by $\mathtt{redexTree}_{\mathfrak{s}}$ is of defined length. If $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$ then we know m is partial (contains ⊥s), infinite or the context c was not split successfully in m. The properties are all simple consequences of the implementation, requiring just a few basic results such as

$$\mathtt{length}\ (xs +\!\!+ ys) = \bot \iff (\mathtt{length}\ xs = \bot\ \vee \mathtt{length}\ ys = \bot)$$

To explain why we need this new terminology one needs to understand why an attempt to prove the theorem relating failure to divergence

would fail without it. Divergence, $\Uparrow$, expresses the property that for a given world/program pair no reduction sequence of finite length can result in convergence to some normal form. The way that one proves that a world/program pair diverges is to prove, by inducting over i, that if it converged to normal form after i steps one can show a contradiction.

It was originally thought that the following lemma would suffice: If $\mathsf{PRE}_{\mathfrak{s}}$ and $w \Vdash m \longrightarrow^c w' \Vdash m'$, then $w \Vdash m \uparrow^c$ implies $w' \Vdash m' \uparrow^c$. In other words, if a program can possibly fail, then it cannot escape from that possibility by reducing to a different world/program pair. But this proof strategy doesn't hold for some programs which are badly-formed, and the following counter-example shows why: (to be run in model `lock` with any context; world state should be TRUE (the mutex is locked); `f` is irrelevant.)

```
let  waits = par () (action WAIT) waits f
in   par () (action UNLOCK) waits f
```

The program `waits` attempts to create an infinite number of processes each of which performs a single WAIT action. The entire program above places an UNLOCK in parallel with `waits`. If the mutex is locked and we first look for a redex on the right-hand side then it will always fail ($\uparrow$) – there are an infinite number of stalled actions within `waits` and the search for a redex simply won't terminate. After executing (`action` UNLOCK), however, each of the infinite number of WAITs are released, and there will always be a redex, and thus no (immediate) failure can occur[2]. By proving that badly-formed programs always diverge, we solve the problem with the `waits` counter-example. Since it always diverges, it is obviously true that if it can fail it will diverge.

It is worth noting that some partial or infinite programs *are* well-formed. Take the `reads` example in Section 3: `reads` itself isn't well-formed, but the program (`test` READI `reads` (`return` (-1))), although "syntactically" infinite, *is* well-formed. It is also worth noting that well-formedness is a property of programs for a given context, not world/program pairs.

[2]One may ask, though, that if the single-step reduction function is given a Guess which tells it always to first try the right-hand side of a parallel reduction, it would also fail immediately, never observing an action. That might solve it for this example, but the problem runs much deeper. What's of real interest is that when we said it made the theorem unprovable, *we didn't mean it made it false*! The issue is that our denotational, domain theoretic model of computable programs contains denotations of non-existent programs - which is an example of the famous full abstraction problem [Ong95]. In the above example, to cause it to fail, the right-hand side must be constantly traversed. But what if the side with an infinite number of actions alternates from left to right? The domain of Prog $\nu\ \alpha\ \rho$ admits any infinite sequence of left/right, but, since an infinite list of Bool can encode any real number, and it's well known that certain real numbers are uncomputable, one cannot always compute a guess which would always pick the correct reduction path to guarantee failure. And the guess *must* be computable, because of the admissibility constraints mentioned.

4.5 Failure implies Divergence

We are now in a position to begin proving the following key theorem.

Theorem 4.2. *If* $\mathsf{PRE}_{\mathfrak{s}}$*, then* $w \Vdash m \uparrow^c$ *implies* $w \Vdash m \Uparrow^c$*.*

Proof. If $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$, use Lemma 4.35. If $w \Vdash m \overset{g\mapsto r}{\uparrow}{}^c_a$, use Lemma 4.38. Otherwise it must be true that $\mathsf{wf}_{\mathfrak{s}}(c, m)$ and $w \Vdash m \overset{g\mapsto \bot}{\uparrow}{}^c$, so use Lemma 4.51. □

It's proof is effectively the merging of three separate lemmas, and these three lemmas are proved in the following three respective sub-subsections.

Badly-Formed Programs

We prove that badly-formed programs diverge by showing that a program cannot reduce out of a badly-formed state, and that no badly-formed program is in normal form.

Lemma 4.33. *If* $w \Vdash m \downarrow^c$ *then* $\mathsf{wf}_{\mathfrak{s}}(c, m)$*.*

Proof. Induction on m. A value is well-formed, as is a stalled action (or any action, for that matter). The other base terms are not in normal form, so don't apply. This is easily proved by induction if $m = m' \texttt{ >>= } f$. When $m = m_l \texttt{ ||| }^p_* m_r$ is in normal form, m_l and m_r must be as well. Therefore, by IH, $\mathsf{wf}_{\mathfrak{s}}(c_l, m_l)$ and $\mathsf{wf}_{\mathfrak{s}}(c_r, m_r)$ and since the concatenation of two finite lists is a finite list, $\mathsf{wf}_{\mathfrak{s}}(c, m)$.

□

Lemma 4.34. *If* $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $w \Vdash m \longrightarrow^c w' \Vdash m'$*, then* $\neg\mathsf{wf}_{\mathfrak{s}}(c, m')$*.*

Proof. Induction on m.

- Base terms. The only base program which isn't well-formed is $\bot$, and it cannot reduce successfully, so doesn't apply.
- $m = m_1 \texttt{ >>= } f$: It must be true that $\neg\mathsf{wf}_{\mathfrak{s}}(c, m_1)$ and $w \Vdash m_1 \longrightarrow^c w_1' \Vdash w'$ so, by IH, $\neg\mathsf{wf}_{\mathfrak{s}}(c, m_1')$. Therefore m_1' cannot be a value, and $\neg\mathsf{wf}_{\mathfrak{s}}(c, m_1 \texttt{ >>= } f)$.
- $m = m_l \texttt{ ||| }^p_* m_r$: Since $\neg\mathsf{wf}_{\mathfrak{s}}(c, m_l \texttt{ ||| }^p_* m_r)$, that means at least one of the following is true: $\neg\mathsf{wf}_{\mathfrak{s}}(c_l, m_l)$ or $\neg\mathsf{wf}_{\mathfrak{s}}(c_r, m_r)$. Now, say m_l contains the reduced redex. This means $w \Vdash m_l \longrightarrow^{c_l} w_l' \Vdash m'$ and $m' = m_l' \texttt{ ||| }^p_* m_r$, for some m_l'. If it was m_r which was badly-formed then the resultant program will still be badly-formed. If it was m_l that was badly formed then, by IH, $\neg\mathsf{wf}_{\mathfrak{s}}(c_l, m_l')$ and the resultant program remains badly-formed. (The proof is symmetric if m_r was reduced, not m_l.)

□

Lemma 4.35. *If* $\neg \mathsf{wf}_{\mathfrak{s}}(c, m)$*, then for all* w*,* $w \Vdash m \Uparrow^c$*.*

Proof. Induction on the number of reduction steps it might take to reduce to normal form. Base case (0): from Lemma 4.33, since m is badly-formed it can't be in normal form. Inductive case: It can't converge to normal form after $i+1$ steps because after one step it is still badly-formed (Lemma 4.34) and it can't converge after i steps (IH). □

Failure of Actions

We prove that the failure of an action implies $w \Vdash m \Uparrow^c$ by showing that even if $w \Vdash m$ can also reduce successfully then it can never "escape" from the action that originally failed. This is the property we originally wanted to prove for all programs, before we showed a counter-example. That it is true for actions which fail is the result of the definition of $|||_{\mathfrak{s}}$. It preserves the fact that $\texttt{af}\ a\ w = \bot$ after the world state is modified by another action.

Lemma 4.36. *Given* $\mathsf{pre}_{\mathfrak{s}}$*,* $c_l \Diamond_{\mathfrak{s}} c_r$*, then if* $w \Vdash m_r \ ^{g_r \mapsto r_r} \uparrow^{c_r}_{a_r}$ *and* $w \Vdash m_l \longrightarrow^{c_l} w_l \Vdash m_l'$*, then* $w_l \Vdash m_r \ ^{r_r \mapsto r_r} \uparrow^{c_r}_{a_r}$*.*

Proof. We know that the action a_r failed: $\texttt{af}\ a_r\ w = \bot$. If m_l reduces silently then $w_l = w$, and Lemma 4.24 can be used. If m_l performs some action a_l then, from Lemma 4.15, $\texttt{wa}\ a_l\ w = \textsc{False}$, from Lemma 4.17, $\texttt{ap}\ c_l\ a = \textsc{True}$, and from the definition, $\texttt{af}\ a_l\ w = (w_l, v_l)$, for some v_l. With $c_l \Diamond_{\mathfrak{s}} c_r$, then derive $a_l \ |||_{\mathfrak{s}}\ a_r$ and $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$. Using these two facts we can prove $\texttt{af}\ a_r\ w_l = \bot$ and $\texttt{wa}\ a_r\ w_l = \textsc{False}$ respectively. Apply Lemma 4.22 to show that a_r will fail when executed in world w_l. □

Lemma 4.37. *If* $\mathsf{PRE}_{\mathfrak{s}}$*,* $w \Vdash m \ ^{g \mapsto r} \uparrow^c_a$ *and* $w \Vdash m \longrightarrow^c w' \Vdash m'$*, then* $w' \Vdash m' \ ^{g \mapsto r} \uparrow^c_a$*.*

Proof. Induction on m. True by contradiction for all base programs – they are deterministic, and therefore cannot fail *and* successfully reduce. For $m_1 \texttt{>>=} f$ (where m_1 isn't a value) apply IH. If m is of the form $m_l \ |||^p_*\ m_r$ (and m_l and m_r aren't both values) then do case analysis on whether the failure and the reduction were on the same side or different sides. If both failure and success are on the same side, apply the IH. If failure and success are on opposite sides, apply Lemma 4.36 to prove that the side that fails will still fail after a successful reduction on the opposite side. (Since reduction can succeed, $\texttt{pf}\ p\ c = (c_l, c_r)$ for some c_l, c_r, and from $\mathsf{PRE}_{\mathfrak{s}}$ this means $\mathsf{pre}_{\mathfrak{s}}$ and $c_l \Diamond_{\mathfrak{s}} c_r$). □

Lemma 4.38. *If* $\mathsf{PRE}_{\mathfrak{s}}$*, then if* $w \Vdash m \ ^{g \mapsto r} \uparrow^c_a$*, then* $w \Vdash m \Uparrow^c$*.*

Proof. Induction on the number of reduction steps it might take to reach normal form. Base case (0): Since $w \Vdash m \uparrow^c$, by Lemma 4.13 it can't be in normal form. Inductive case: It can't reach normal form after $i+1$ steps because, by Lemma 4.37, after one step it can still fail, and by IH it can't converge to normal form after i steps. □

Well-formed failure of $\texttt{nextR}_\mathfrak{s}$

Finally, we must prove that, assuming $\mathsf{PRE}_\mathfrak{s}$ and $\mathsf{wf}_\mathfrak{s}(c, m)$, if $w \Vdash m \overset{g \mapsto \bot}{\uparrow^c}$ then $w \Vdash m \Uparrow^c$. The structure to the proof is rather similar to our proof of Lemma 4.38 at a high level but there are considerably more technical details.

The well-formedness of m has a direct consequence: it means that there are a finite list of potential redexes, and we can therefore induct over the length of the list without admissibility constraints. One can therefore prove existential properties by induction, namely that if $\texttt{nextR}_\mathfrak{s}$ results in $\bot$ then it failed for some *specific* potential redex identified by an integer. Once we have an identifer for problematic redexes we can then prove admissible properties about a lazy structure like $\mathsf{Prog}\ \nu\ \alpha\ \rho$.

This subsection of the proof is not pretty and requires us to get our hands dirty manipulating the elements of lists of redexes. (Our approach was influenced very much by the existence of other pre-proved theorems relating to lists.) Also, for the amount of effort required, its relevance is disappointingly small. As we show below, if a program fails in the above manner it is for one of two rather uninteresting reasons. We include the proof for absolute completeness and as a testament to the rigour a proof-assistant imposes on its user.

We begin by proving three lemmas relating lists of potential redexes of programs of the form $m_l \mid\mid\mid^p_* m_r$ to that of m_l and m_r.

Lemma 4.39. *If* $\texttt{pf}\ p\ c = (c_l, c_r)$, $\mathsf{wf}_\mathfrak{s}(c, m_l \mid\mid\mid^p_* m_r)$ *and it is true that* $\texttt{redexList}_\mathfrak{s}\ [b{:}g]\ c\ (m_l \mid\mid\mid^p_* m_r)\ !!\ i = \textsc{Redex}\ ([L{:}r], x)$ *then for some* i_l $\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l\ !!\ i_l = \textsc{Redex}\ (r, x)$.

Proof. First, m_l and m_r cannot both be values since this would means $\texttt{redexList}_\mathfrak{s}\ [b{:}g]\ c\ (m_l \mid\mid\mid^p_* m_r) = [\textsc{Redex}\ ([], (c, \textsc{Silent}))]$, and $[L{:}r] \neq []$. Cases analysis on b:

- $b = L$: Apply Lemma 4.27. Because $\mathsf{wf}_\mathfrak{s}(c, m_l \mid\mid\mid^p_* m_r)$, there are a finite, defined number of potential redexes for both m_l and m_r. If $i < \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l)$ then $\texttt{map addL \$ redexList}_\mathfrak{s}\ g\ c_l\ m_l\ !!\ i = \textsc{Redex}\ ([L{:}r], x)$ and therefore $\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l\ !!\ i = \textsc{Redex}\ (r, x)$, so let $i_l = i$. If $i \geq \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l)$ then it is true that $\texttt{map addR \$ redexList}_\mathfrak{s}\ g\ c_r\ m_r\ !!\ i = \textsc{Redex}\ ([L{:}r], x)$, and this is a contradiction since addR cannot modify the resultant route so it becomes $[L{:}r]$.

- $b = R$: Apply Lemma 4.28. Similar to the above except m_r is searched first. If $i < \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$ then contradiction. Otherwise $i \geq \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$, so let $i_l = i - \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$.

□

Lemma 4.40. *If* $\texttt{pf}\ p\ c = (c_l, c_r)$, $\mathsf{wf}_\mathfrak{s}(c, m_l\ /\!/\!/^p_*\ m_r)$ *and*

$$\texttt{redexList}_\mathfrak{s}\ [b{:}g]\ c\ (m_l\ /\!/\!/^p_*\ m_r)\ !!\ i = \textsc{Redex}\ (\lceil R{:}r \rceil, x)$$

then for some i_r $\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r\ !!\ i_r = \textsc{Redex}\ (r, x)$.

Proof. Similar to the proof of Lemma 4.39. □

Lemma 4.41. *If* $\texttt{pf}\ p\ c = (c_l, c_r)$ *and* $\mathsf{wf}_\mathfrak{s}(c, m_l\ /\!/\!/^p_*\ m_r)$ *then*

$$\texttt{redexList}_\mathfrak{s}\ g\ c\ (m_l\ /\!/\!/^p_*\ m_r)\ !!\ i = \textsc{Redex}\ (\lceil\rceil, x)$$

for some i *if and only if* m_l *and* m_r *are both values.*

Proof. If: By contradiction. If m_l and m_r were not both values then, depending on whether g is $[L{:}g']$ or $[R{:}g']$ for some g', use either Lemma 4.27 or Lemma 4.28 to show that the resultant route must be either $[L{:}r']$ or $[R{:}r']$ for some r'. A route of value [] is impossible.

Only if: both m_l and m_r are values, so by the implementation there is just one silent reduction at route []. □

The next step is to identify how and why $w \Vdash m\ ^{g \mapsto \bot}\!\uparrow^c$. The following three lemmas together show that if $w \Vdash m\ ^{g \mapsto \bot}\!\uparrow^c$ then for some action a either

- $\texttt{ap}\ c\ a \neq \textsc{True}$, or
- $\texttt{ap}\ c\ a = \textsc{True}$ and $\texttt{wa}\ a\ w = \bot$

Lemma 4.42. *If a list* rxs *is finite and* $\texttt{firstRxW}_\mathfrak{s}\ w\ rxs = \bot$ *then there exists some* i, $i \geq 0$, $i < \texttt{length}\ rxs$ *such that* $\texttt{check}_\mathfrak{s}\ w\ (rxs\ !!\ i) = \bot$.

Proof. Induction on the length of rxs. If $rxs = []$, $\texttt{firstRxW}_\mathfrak{s}\ w\ rxs \neq \bot$. If $rxs = [rx : rxs']$ then either $\texttt{check}_\mathfrak{s}\ w\ rx = \bot$ (let $i = 0$), or $\texttt{firstRxW}_\mathfrak{s}\ w\ rxs' = \bot$, in which case let $i = i' + 1$, where i' is the index from the IH. □

Lemma 4.43. *If* $\mathsf{wf}_\mathfrak{s}(c, m)$ *and* $w \Vdash m\ ^{g \mapsto \bot}\!\uparrow^c$ *then there exists an integer* i *such that* $(\texttt{redexList}_\mathfrak{s}\ g\ c\ m)\ !!\ i = \textsc{Redex}\ (r, (c_1, \textsc{Action}\ a))$ *and* $\texttt{check}_\mathfrak{s}\ w\ (\textsc{Redex}\ (r, (c_1, \textsc{Action}\ a))) = \bot$.

Proof. $w \Vdash m \overset{g \mapsto \bot}{\uparrow^c}$ is defined as $\texttt{nextR}_\mathfrak{s}(g, c, w, m) = \bot$, which is equivalent to $\texttt{firstRxW}_\mathfrak{s}\ w\ (\texttt{redexList}_\mathfrak{s}\ g\ c\ m) = \bot$. Since $\mathsf{wf}_\mathfrak{s}(c, m)$ there are only a finite number of potential redexes, so apply Lemma 4.42 to prove that there is some i and such that $\texttt{check}_\mathfrak{s}\ w\ (\texttt{redexList}_\mathfrak{s}\ g\ c\ m\ !!\ i) = \bot$. $\texttt{redexList}_\mathfrak{s}\ g\ c\ m\ !!\ i$ must be of the form REDEX $(r, (c_1, \text{ACTION } a))$ because $\texttt{check}_\mathfrak{s}$ never fails for silent redexes or NOREDEX. (There is also a separate and entirely uninteresting proof that $\texttt{redexList}_\mathfrak{s}$ cannot return a list with $\bot$ as an element. We omit this entirely). □

Lemma 4.44. *$\texttt{check}_\mathfrak{s}\ w$ (REDEX $(r, (c, \text{ACTION } a))) = \bot$ if and only if either $\texttt{ap}\ c\ a \neq$ TRUE or $\texttt{ap}\ c\ a =$ TRUE $\wedge$ $\texttt{wa}\ a\ w = \bot$.*

Proof. Immediate. □

We temporarily replace any reference to divergence with properties of $\texttt{check}_\mathfrak{s}$ and $\texttt{redexList}_\mathfrak{s}$ and continue the proof in the style of previous subsections. Lemma 4.46, Lemma 4.47 and Lemma 4.48 are really just re-workings of Lemma 4.17, Lemma 4.36 and Lemma 4.37 respectively.

Lemma 4.45. *If $\texttt{check}_\mathfrak{s}\ w$ (REDEX $(r, x)) = \bot$ then for all r_1 it is true that $\texttt{check}_\mathfrak{s}\ w$ (REDEX $(r_1, x)) = \bot$.*

Proof. A direct consequence of the implementation. $\texttt{check}_\mathfrak{s}$ never examines the route r. □

Lemma 4.46. *If $\mathsf{wf}_\mathfrak{s}(c, m)$ and $\mathsf{pre}_\mathfrak{s}$, then if $\texttt{redexList}_\mathfrak{s}\ g\ c\ m\ !!\ i =$ REDEX $(r, (c_1, \text{ACTION } a))$ then $\texttt{ap}\ c_1\ a =$ TRUE implies $\texttt{ap}\ c\ a =$ TRUE.*

Proof. Induction on m. For base terms, only $\texttt{action}\ a$ meets the precondition and $\texttt{ap}\ c\ a =$ TRUE implies $\texttt{ap}\ c\ a =$ TRUE. If $m = m_1\ \texttt{>>=}\ f$ then apply Lemma 4.26 and IH. If $m = m_l\ |||^p_*\ m_r$ then apply either Lemma 4.39 or Lemma 4.40 depending on the route r. If $r = [L{:}r']$ we can prove that $\texttt{redexTree}_\mathfrak{s}\ g'\ c_l\ m_l\ !!\ i_l =$ REDEX $(r', (c_1, \text{ACTION } a))$ for some i_l. With IH, prove $\texttt{ap}\ c_1\ a =$ TRUE implies $\texttt{ap}\ c_l\ a =$ TRUE and with $\mathsf{pre}_\mathfrak{s}$ prove that $c_l \sqsubseteq_\mathfrak{s} c$ and therefore $\texttt{ap}\ c\ a =$ TRUE. The proof for the right-hand side is similar. □

Lemma 4.47. *If $\mathsf{pre}_\mathfrak{s}$, $c_l \lozenge_\mathfrak{s} c_r$, $\mathsf{wf}_\mathfrak{s}(c_r, m_r)$, $\texttt{redexTree}_\mathfrak{s}\ g_r\ c_r\ m_r\ !!\ i =$ REDEX $(r_r, (c_{r1}, \text{ACTION } a_r))$, $\texttt{check}_\mathfrak{s}\ w$ (REDEX $(r_r, (c_{r1}, \text{ACTION } a_r))) = \bot$ and $w \Vdash m_l \longrightarrow^{c_l} w_l \Vdash m_l'$, then $\texttt{check}_\mathfrak{s}\ w_l$ (REDEX $(r_r, (c_{r1}, \text{ACTION } a_r))) = \bot$.*

Proof. From Lemma 4.44, either $\texttt{ap}\ c_{r1}\ a_r \neq$ TRUE or $\texttt{ap}\ c_{r1}\ a_r =$ TRUE $\wedge$ $\texttt{wa}\ a_r\ w = \bot$. If it is the former then this will be unaffected by a reduced action, so regardless of w_l, $\texttt{check}_\mathfrak{s}$ will still fail. If $\texttt{ap}\ c_{r1}\ a_r =$ TRUE and $\texttt{wa}\ a_r\ w = \bot$, then apply Lemma 4.46 to prove $\texttt{ap}\ c\ a_r =$ TRUE. Next examine the reduction of m_l. If it's silent, then $w_l = w$, so m_r can fail in

the same way. If reducing m_l performs an action a_l then apply Lemma 4.15 to prove $\texttt{wa}\ a_l\ w = \textsc{False}$ and, from Lemma 4.17, $\texttt{ap}\ c_l\ a = \textsc{True}$. With $c_l \diamondsuit_{\mathfrak{s}} c_r$ derive $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$, which guarantees that $\texttt{wa}\ a_r\ w_l = \bot$. This, by Lemma 4.44, proves that $\texttt{check}_{\mathfrak{s}}$ will also fail with world w_l. □

Lemma 4.48. *If* $\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $\mathsf{PRE}_{\mathfrak{s}}$ *then*

$$\begin{aligned} \mathit{redexTree}_{\mathfrak{s}}\ g_1\ c\ m\ !!\ i &= \textsc{Redex}\ (r_1, (c_1, \textsc{Action}\ a)) \\ \mathit{check}_{\mathfrak{s}}\ w\ (\textsc{Redex}\ (r_1, (c_1, \textsc{Action}\ a))) &= \bot \end{aligned}$$

and $w \Vdash m \xrightarrow{g_2 \mapsto r_2}_c w' \Vdash m'$ *implies*

$$\mathit{check}_{\mathfrak{s}}\ w'\ (\textsc{Redex}\ (r_1, (c_1, \textsc{Action}\ a))) = \bot$$

Proof. Induction on m.

- True by contradiction for all base redexes, since, being deterministic, they cannot both reduce and fail to reduce.
- For $m = m_1\ \texttt{>>=}\ f$ (m_1 not a value), from Lemma 4.26

  $$\texttt{redexTree}_{\mathfrak{s}}\ g_1\ c\ (m_1\ \texttt{>>=}\ f) = \texttt{redexTree}_{\mathfrak{s}}\ g_1\ c\ m_1$$

  and from the language semantics we can derive $w \Vdash m_1 \xrightarrow{g_2 \mapsto r_2}_c w' \Vdash m_1'$ and $m' = m_1'\ \texttt{>>=}\ f$ from $w \Vdash m_1\ \texttt{>>=}\ f \xrightarrow{g_2 \mapsto r_2}_c w' \Vdash m'$. Apply IH.
- When $m = m_l\ |||^p_*\ m_r$, where m_l and m_r are not both values, perform case analysis on the routes r_1 and r_2. Neither can be [] (Lemma 4.41, Lemma 4.5).
  - If both redexes are on the same side, say the left-hand side, making $r_1 = [L{:}r_1']$ and $r_2 = [L{:}r_2']$, apply Lemma 4.39 and Lemma 4.10. With IH one can now prove that

    $$\texttt{check}_{\mathfrak{s}}\ w'\ (\textsc{Redex}\ (r_1', (c_1, \textsc{Action}\ a))) = \bot$$

    Apply Lemma 4.45 to prove the same is true with route $[L{:}r_1']$.
  - If the redexes are on opposite sides then from $\texttt{pf}\ p\ c = (c_l, c_r)$ and $\mathsf{PRE}_{\mathfrak{s}}$ we can prove $\mathsf{pre}_{\mathfrak{s}}$ and $c_l \diamondsuit_{\mathfrak{s}} c_r$ (and $c_r \diamondsuit_{\mathfrak{s}} c_l$ by symmetry). Apply Lemma 4.47.

□

We are finally in a position to "repackage" divergence. Lemma 4.49 shows that re-using a previous route as a Guess will mean the redex at that route will be the first to be chosen (a re-working of the proof of both Lemma 4.18 and Lemma 4.20). Lemma 4.50 uses this to tidy up Lemma 4.48, and Lemma 4.51 follows exactly the same procedure as Lemma 4.38 to prove failure implies divergence.

Lemma 4.49. *If* $\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $\texttt{redexList}_{\mathfrak{s}}\ g\ c\ m\ !!\ i = \textsc{Redex}\ (r,x)$ *then* $\texttt{redexList}_{\mathfrak{s}}\ r\ c\ m\ !!\ 0 = \textsc{Redex}\ (r,x)$.

Proof. Induction on m. For base terms, $\bot$ is not well-formed and otherwise there is just one potential redex, the Guess is ignored and i must be 0 already. If $m = m_1\ \texttt{>>=}\ f$ where m_1 is not a value, use Lemma 4.26 and IH. If $m = m_l\ \texttt{|||}^p_*\ m_r$, where m_l and m_r are not both values, then do case analysis on r. If $r = [L{:}r']$, apply Lemma 4.39 to prove that for some i_l, $\texttt{redexList}_{\mathfrak{s}}\ g'\ c_l\ m_l\ !!\ i_l = \textsc{Redex}\ (r',x)$, where $g = [b{:}g']$. By IH, this implies $\texttt{redexList}_{\mathfrak{s}}\ r'\ c_l\ m_l\ !!\ 0 = \textsc{Redex}\ (r',x)$, and by Lemma 4.27 this can be shown to imply $\texttt{redexList}_{\mathfrak{s}}\ [L{:}r']\ c\ (m_l\ \texttt{|||}^p_*\ m_r)\ !!\ 0 = \textsc{Redex}\ ([L{:}r'],x)$. That is: if x is the first potential redex in m_l with guess r' then it will be the first potential redex in $m_l\ \texttt{|||}^p_*\ m_r$ with guess $[L{:}r']$, since it will be m_l that is searched first. If $r = [R{:}r']$ then the proof is similar. □

Lemma 4.50. *If* $\mathsf{PRE}_{\mathfrak{s}}$, $\mathsf{wf}_{\mathfrak{s}}(c, m)$, $w \Vdash m\ ^{g \mapsto \bot}\!\uparrow^c$ *and* $w \Vdash m \longrightarrow^c w' \Vdash m'$, *then for some* g_1 $w' \Vdash m'\ ^{g_1 \mapsto \bot}\!\uparrow^c$.

Proof. Apply Lemma 4.43 to prove that some i $\texttt{redexList}_{\mathfrak{s}}\ g\ c\ m\ !!\ i = \textsc{Redex}\ (r,(c_1,\textsc{Action}\ a))$ and $\texttt{check}_{\mathfrak{s}}\ w\ \textsc{Redex}\ (r,(c_1,\textsc{Action}\ a)) = \bot$. Use Lemma 4.48 to prove $\texttt{check}_{\mathfrak{s}}\ w'\ \textsc{Redex}\ (r,(c_1,\textsc{Action}\ a)) = \bot$ (a can still cause failure in world w'). Apply 4.49 to prove $\texttt{redexList}_{\mathfrak{s}}\ r\ c\ m\ !!\ 0 = \textsc{Redex}\ (r,(c_1,\textsc{Action}\ a))$ – if we chose Guess r the second time then the first redex to be chosen will be action a. It can be shown easily that this means $\texttt{nextR}_{\mathfrak{s}}(r, c, w', m') = \bot$, and therefore, letting $g_1 = r$, $w' \Vdash m'\ ^{g_1 \mapsto \bot}\!\uparrow^c$. □

Lemma 4.51. *If* $\mathsf{PRE}_{\mathfrak{s}}$, *then if* $\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $w \Vdash m\ ^{g \mapsto \bot}\!\uparrow^c$, *then* $w \Vdash m \Uparrow^c$.

Proof. Induction on the number of reduction steps it might take to reach normal form. Base case (0): Since $w \Vdash m \uparrow^c$, by Lemma 4.13 it can't be in normal form. Inductive case: It can't converge to normal form after $i + 1$ steps because, by Lemma 4.50, after one step it can still fail, and by IH it can't converge to normal form after i steps. □

4.6 Confluence of Reduction

The confluence result is finally proved in this subsection. As is standard for confluence proofs, this is achieved by proving a so-called "diamond property". This is a proof that if two different non-deterministic reductions are possible, then there exists a common reduct to which both sides can then reduce.

We assume for all of this sub-subsection that $\mathsf{PRE}_{\mathfrak{s}}$ holds.

Lemma 4.52. *If* $w \Vdash m \ {}^{1}\!\downarrow^{c} \ w' \Vdash m'$ *(i.e.* $w \Vdash m \longrightarrow^{c} w' \Vdash m'$ *and* $w' \Vdash m' \downarrow^{c}$*), then* $w \Vdash m \ {}^{1}\!\Downarrow^{c} \ w' \Vdash m'$*. (This is a proof of confluence for programs which require one reduction step.)*

Proof. Induction on m.

- All the base programs either can't reduce or reduce deterministically.
- With m_1 `>>=` f (m_1 not a value), we know that $w \Vdash m_1 \longrightarrow^{c} w' \Vdash m_1'$, $w' \Vdash m_1' \downarrow^{c}$ and $m' = m_1'$ `>>=` f. Because m' is stalled, m_1' can't be a value, so apply IH.
- With m_l `|||`$^{p}_{*}$ m_r first prove that neither side can fail (if one side did fail, then the whole program could fail, and, by Theorem 4.2, the whole program could never reduce to a normal form – but it does, after one step.) Next, prove that it's impossible for *both* sides to be successfully reduced (if both were, then by Theorem 4.1, after single-step reducing one particular side the program couldn't be in normal form – the opposite side could still either be reduced or fail.) So any two reductions must be both in m_l or both in m_r. Apply IH to prove that both reductions on that side will have the same outcome.

□

Lemma 4.53. *Diamond Property. This is the proof that if we know that* $w \Vdash m \xrightarrow{g' \mapsto r'}_{c} w' \Vdash m'$ *and* $w \Vdash m \xrightarrow{g'' \mapsto r''}_{c} w'' \Vdash m''$ *then either*

- *The same redex was reduced:* $w' = w''$ *and* $m' = m''$*.*
- *There is a common reduct: For some* w''' *and* m'''*,* $w' \Vdash m' \xrightarrow{r'' \mapsto r''}_{c} w''' \Vdash c'''$ *and* $w'' \Vdash m'' \xrightarrow{r' \mapsto r'}_{c} w''' \Vdash c'''$*.*
- *Both sides will diverge:* $w' \Vdash m' \Uparrow^{c}$ *and* $w'' \Vdash m'' \Uparrow^{c}$*.*

Actually, in the last case we prove something slightly more specific which always implies divergence. This because divergence is inadmissable, requiring existential quantification and negation. We prove instead that they either fail or reduce to a badly-formed program. Both of these imply divergence, and it is easy to prove that if a `>>=` or a `par` diverge in this special way, the whole program will also diverge in this way.

Proof. Induction on m.

- All base reduction rules either can't reduce at all (contradiction), or deterministically reduce to the same redex.

- $w \Vdash m_1 \texttt{ >>= } f \longrightarrow^c w' \Vdash m_1' \texttt{ >>= } f$ and $w \Vdash m_1 \texttt{ >>= } f \longrightarrow^c w'' \Vdash m_1'' \texttt{ >>= } f$. If m_1' or m_1'' are stalled, then by Lemma 4.52, $w' = w''$ and $m_1' = m_1''$. This case rules out either m_1' or m_1'' being values, so the next reduction must be within m_1' and m_1'', and not the application of f to something. Apply IH.

- Both
$$w \Vdash m_l \mid\mid\mid^p_* m_r \longrightarrow^c w' \Vdash m_l' \mid\mid\mid^p_* m_r'$$
and
$$w \Vdash m_l \mid\mid\mid^p_* m_r \longrightarrow^c w'' \Vdash m_l'' \mid\mid\mid^p_* m_r''$$
The two reductions may have reduced different sides, or the same side.

  - If the same side was reduced, say that of m_l, use Lemma 4.52 to account for the possibility of m_l' or m_l'' being stalled or values (which would cause reduction to flip to the other side the second time around.) If either side was, then both reductions would have been the same. Apply IH.
  - If one reduction was in m_l and the other was in m_r, then apply Theorem 4.1. Either both reduction orders can then fail (which entails failure for the whole program and therefore divergence by Theorem 4.2), or both reduction orders can reduce once more to get the same result by reducing the opposite side to that initially reduced. However, if one side reduces to a program which is $\bot$ (which is still a successful reduction), then depending on the ordering, the parallel execution may accidentally force the program's evaluation.

    This is why we need divergence at all, and is a consequence of the side condition that $m_l \neq \bot$ and $m_r \neq \bot$ in the single-step semantics. As an example, consider the program

    $$\texttt{par}\ \textsc{LeftWr}\ (\texttt{action}\ (\textsc{WriteI}\ 6))\ (\texttt{return}\ 2 \texttt{ >>= } \lambda\_.\bot)\ \texttt{f}$$

    The left-hand side reduces to (`return` 0) and the right-hand side reduces to $\bot$. Reducing the left side then the right side yields `par` LeftWr (`return` 0) $\bot$ `f`, a badly-formed program (which diverges), but reducing the right side and then the left will fail outright (which causes divergence.)

□

Theorem 4.3. *Confluence. If* $w \Vdash m \ {}^i\!\downarrow^c \ w_1 \Vdash m_1$ *then* $w \Vdash m \ {}^i\!\Downarrow^c w_1 \Vdash m_1$.

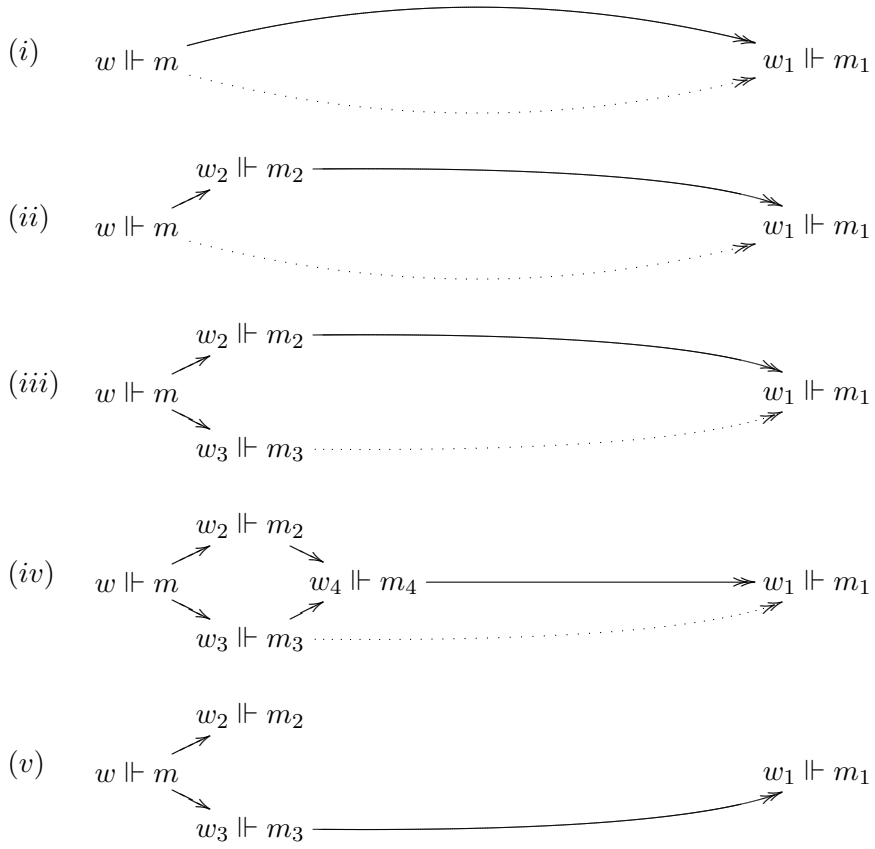

Figure 13: Proving Confluence using the Diamond Property

Proof. Induction on i. If $i = 0$, then apply Lemma 4.13. In the inductive case ($i = k + 1$), first prove for $k = 0$ using Lemma 4.52. Figure 13 demonstrates the sequence of steps required to prove the full inductive case. Diagram (i) is the initial proof obligation.

Assuming $k \geq 1$ (and $i > 1$), extract the first of the $k + 1$ reduction steps to yield $w \Vdash m \longrightarrow^c w_2 \Vdash m_2$ and $w_2 \Vdash m_2 \ {}^{k}\downarrow^c \ w_1 \Vdash m_1$, for some w_2, m_2 (diagram (ii)). To prove confluence, we must show that for any arbitrary reduction of $w \Vdash m$, it will eventually reduce to $w_1 \Vdash m_1$. First prove that despite the nondeterminism, $w \Vdash m$ cannot fail or be in normal form. (It can't be in normal form because it can reduce – Lemma 4.13; it can't fail because it does eventually converge to normal form – Theorem 4.2.) Therefore, the only alternative to reducing to $w_2 \Vdash m_2$ is that there is some other $w_3 \Vdash m_3$ such that $w \Vdash m \longrightarrow^c w_3 \Vdash m_3$ (diagram (iii)). We must now show that $w_3 \Vdash m_3 \ {}^{k}\Downarrow^c \ w_1 \Vdash m_1$. Apply IH twice in a row: first in a forwards style to prove that we know $w_2 \Vdash m_2 \ {}^{k}\Downarrow^c \ w_1 \Vdash m_1$; second in a backwards style to show that we need only prove $w_3 \Vdash m_3 \ {}^{k}\downarrow^c \ w_1 \Vdash m_1$. From the diamond property (Lemma 4.53), one of the following is true:

- $w_2 = w_3$ and $m_2 = m_3$: Trivial.
- There is a common reduct $w_4 \Vdash m_4$ (diagram (iv) in Figure 13) such that $w_2 \Vdash m_2 \longrightarrow^c w_4 \Vdash m_4$ and $w_3 \Vdash m_3 \longrightarrow^c w_4 \Vdash m_4$: From the confluence of reduction of $w_2 \Vdash m_2$, prove that $w_4 \Vdash m_4 \ {}^{k-1}\downarrow^c \ w_1 \Vdash m_1$. Thus $w_3 \Vdash m_3 \ {}^{k}\downarrow^c \ w_1 \Vdash m_1$.
- $w_2 \Vdash m_2 \Uparrow^c$ and $w_3 \Vdash m_3 \Uparrow^c$: Proof by contradiction (we know $w_2 \Vdash m_2$ can converge to normal form.)

□

Corollary 4.54. *Either $w \Vdash m \Uparrow^c$ or there exists a w',m' such that $w \Vdash m \Downarrow^c w' \Vdash m'$ (and not both.)*

Proof. The definition $w \Vdash m \Uparrow^c$ is that it's not the case that $w \Vdash m \downarrow^c w' \Vdash m'$, and $w \Vdash m \downarrow^c w' \Vdash m'$, by confluence, is equivalent to $w \Vdash m \Downarrow^c w' \Vdash m'$.

□

A Sparkle Proof Sections

The machine-readable form of the proofs may be obtained from the following URL:

`http://www.cs.tcd.ie/research_groups/fmg/archive/CURIO-Proofs.html`

It isn't always straightforward to simply list the Sparkle theorem which corresponds to each individual lemma. Almost all proofs which required induction over program structure needed to be proved in such a way that Sparkle could show it was admissible. These results – most of which have the letters `_adms` appended to their name – contain the guts of the proof. Usually a theorem was then proved (without the "adms") which expressed the result in a more natural style. Also, many definedness results were omitted entirely in this document. These are uninteresting but still affect what was actually proved and our ability to form a direct link between presented lemmas and the Sparkle theorems.

Below is a select list of results in this document for which there are equivalent Sparkle theorems.

Result	Sparkle Section	Theorem Name
Lemma 2.1	`curio_examples`	`PRE_Buffer`
Lemma 2.2	`curio_examples`	`PRE_Lock`
Lemma 2.3	`curio_examples`	`PRE_IVar`
Lemma 3.1	`dountil`	`dountil_main_theorem`
R.E. Proof	`curio_rdce`	`rdce_run`
Lemma 4.1	`curio_prelude`	`next_separated`
Lemma 4.8	`curio_prelude`	`nextRedex_Par_False`
Lemma 4.9	`curio_prelude`	`nextRedex_Par_True`
Lemma 4.12	`curio_reduction`	`nextRedex_stalled_any_guess`
Lemma 4.14	`curio_reduction`	`nextRedex_wa`
Lemma 4.16	`curio_reduction`	`nextRedex_ap`
Lemma 4.18	`curio_reduction`	`nextRedex_interfere_Silent`
Lemma 4.20	`curio_reduction`	`nextRedex_interfere_Action`
Theorem 4.1	`curio_reduction`	`next_disjoint_with_nextRedex`
Lemma 4.25 (i)	`curio_failure_internals`	`length_preorder_shuffle`
Lemma 4.25 (v)	`curio_failure_internals`	`shuffle_mapTree`
Lemma 4.25 (vi)	`curio_failure_internals`	`preorder_mapTree`
Lemma 4.25 (viii)	`curio_failure_internals`	`firstRedex_++`
Lemma 4.32	`curio_failure`	`nextRedex_separated`
Theorem 4.2	`curio_failure`	`next_failure_divergence`
Lemma 4.34	`curio_failure`	`next_badlyformed_failure`
Lemma 4.35	`curio_failure`	`next_badlyformed_divergence`
Lemma 4.52	`curio_confluence`	`next_final_step`
Lemma 4.53	`curio_confluence`	`tidy_diamond`
Theorem 4.3	`curio_confluence`	`rdce_CONFLUENCE`

References

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, Revised Edition, 1984.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[DBvE05] Malcolm Dowse, Andrew Butterfield, and Marko van Eekelen. Reasoning about deterministic concurrent functional I/O. In Clemens Grelck and Frank Huch, editors, *Proceedings of IFL 2004*, volume LNCS3474, pages 177–194. Springer-Verlag, 2005.

[dMvEP01] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers. In Thomas Arts and Markus Mohnen, editors, *Proceedings of the 13th International Workshop, IFL2001*, number 2312 in LNCS, pages 55–71. Springer-Verlag, 2001.

[Hue94] Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994. Preliminary version available as INRIA Technical Report 2009, August 1993.

[Iga74] Shigeru Igarashi. Admissibility of fixed-point induction in first-order logic of typed theories. In *Proceedings of the International Symposium on Theoretical Programming*, pages 344–383, London, UK, 1974. Springer-Verlag.

[Nip96] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *LNCS*, pages 733–747. Springer, 1996.

[Ong95] C.-H. L. Ong. Correspondence between operational and denotational semantics: the full abstraction problem for PCF. *Handbook of logic in computer science (vol. 4): semantic modelling*, pages 269–356, 1995.

[Pau87] Lawrence C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1987.

[Pey01] Simon Peyton Jones. Tackling the awkward squad – monadic input/output, concurrency, exceptions, and foreign language calls in Haskell. In CAR Hoare, M Broy, and R Stein-brueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, pages 47–96. IOS Press, 2001.

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In ACM, editor, *POPL '96: Florida, 21–24 January 1996*, pages 295–308, New York, NY, USA, 1996. ACM Press.

[PHWH03] Simon Peyton Jones, Paul Hudak, Philip Wadler, and John Hughes, editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[PvE01] Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean version 2.0 language report. `http://www.cs.kun.nl/~clean/`, December 2001.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.