UNIVERSITY OF

Semantics for core Concurrent ML using computation types

The resulting labelled transition system can be used as the basis of an equational theory of CML expressions, using *on* as equivalence.

Unfortunately, there are some problems with this semantics:

- It is complex, due to having to allow expressions in any evaluation context to reduce (for example requiring three rules for if-expressions rather than Reppy's two axiom schemas).
- It produces very long reductions, due to large numbers of 'book-keeping' steps (for example the long reduction in Table 9).
- The resulting equational theory does not have pleasant mathematical properties (for example neither β- nor η-conversion hold for the language).

In this paper we present a variant of CML using *o p on p*. These provide an explicit type constructor $\_\,\mathsf{comp}$ for computation, which means that the type system can distinguish between expressions which can perform computation (those of type $A\,\mathsf{comp}$) and those which are guaranteed to be in normal form (anything else). Differentiating by type between expressions which can and cannot perform reductions makes the operational semantics much simpler, for example the much shorter reduction in Table 16 and the simpler operational rules for if-expressions:

$$\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ f\ \mathsf{else}\ g \xrightarrow{\tau} f} \qquad \frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ f\ \mathsf{else}\ g \xrightarrow{\tau} g}$$

Computation types were originally proposed by Moggi (1991) in a denotational setting to provide models of non-trivial computation (such as CML communication) without losing pleasant mathematical properties (such as β- and η-reduction). Moggi provided a translation from the call-by-value λ-calculus into the language with computation types, which we can adapt for CML and prove to be correct up to weak bisimulation.

We can also use equational reasoning to transform inefficient programs (such as the translation of the long reduction in Table 9) into efficient ones (such as the short reduction in Table 16). We conjecture that such optimizations may make languages with explicit computation types simpler to optimize.

IN SECTION 2 we present a cut-down version of the operational semantics for CML presented in (Ferreira, Hennessy, and Jeffrey 1995), including a suitable definition of bisimulation for CML programs.

IN SECTION 3 we present the variant of CML with explicit computation types, and show that the resulting equational theory of bisimulation has better mathematical properties than that of CML. This is a variant of the language presented in (Jeffrey 1995a).

IN SECTION 4 we provide a translation from the first language into the second, and show that it is correct up to bisimulation.

This spawns send (a, v) off for concurrent execution, then evaluates accept a. These two processes can then communicate. In this paper, we are ignoring CML's *r* so spawn has type:

$$\mathtt{spawn} : (\mathtt{unit} \to A) \to \mathtt{unit}$$

CML does *no* provide a general 'external choice' operator such as CCS $+$. Instead, guarded choice is provided, and the type mechanism is used to ensure that choice is only ever used on guarded computation. The type A event is used as the type of guarded processes of type A, and CML allows for the creation of guarded input and output:

$$\mathtt{transmit}_A : (\mathtt{chan} * A) \to \mathtt{unit\,event} \qquad \mathtt{receive}_A : \mathtt{chan} \to A\,\mathtt{event}$$

and for guarded sequential computation:

$$\mathtt{wrap} : (A\,\mathtt{event} * (A \to B)) \to B\,\mathtt{event}$$

For example the guarded process which inputs a value on a and outputs it on b is given:

$$\mathtt{wrap}\,(\mathtt{receive\,a}, \mathtt{fn}\,x = \mathtt{send}\,(\mathtt{b}, x)) : \mathtt{unit\,event}$$

CML provides choice between guarded processes using choose. In CML this is defined on lists, but for simplicity we shall give it only for pairs:

$$\mathtt{choose} : (A\,\mathtt{event} * A\,\mathtt{event}) \to A\,\mathtt{event}$$

For example the guarded process which chooses between receiving a signal on a or b is:

$$\mathtt{choose}\,(\mathtt{receive}_A\,\mathtt{a}, \mathtt{receive}_A\,\mathtt{b}) : A\,\mathtt{event}$$

Guarded processes can be treated as any other process, using the function sync:

$$\mathtt{sync} : A\,\mathtt{event} \to A$$

For example, we can execute the above guarded process by saying:

$$\mathtt{sync}\,(\mathtt{choose}\,(\mathtt{receive}_A\,\mathtt{a}, \mathtt{receive}_A\,\mathtt{b})) : A$$

In fact, accept and send are not primitives in CML, and are defined:

$$\mathtt{accept}_A \stackrel{\mathrm{def}}{=} \mathtt{fn}\,x = \mathtt{sync}\,(\mathtt{receive}_A\,x)$$
$$\mathtt{send}_A \stackrel{\mathrm{def}}{=} \mathtt{fn}\,x = \mathtt{sync}\,(\mathtt{transmit}_A\,x)$$

This paper cannot provide a full introduction to CML, and the interested reader is referred to Reppy's papers (Reppy 1991; Reppy 1992) for further explanation.

The fragment of CML we will consider here is missing much of CML's functionality, notably polymorphism, guards and thread identifiers. It is similar to the fragment of CML considered in (Ferreira, Hennessy, and Jeffrey 1995) except that for simplicity we do not consider the always command. We will call this subset 'core τ-free CML', or CML for short.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash ce : B}[c : A \to B] \qquad \frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash f : A \quad \Gamma \vdash g : A}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g : A}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash f : B}{\Gamma \vdash (e, f) : A * B} \qquad \frac{\Gamma \vdash e : A \quad \Gamma, x : A \vdash f : B}{\Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ f : B}$$

$$\frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash f : A}{\Gamma \vdash ef : B} \qquad \frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}[x \neq y]$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash k : \mathtt{chan}}$$

$$\frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{\Gamma, x : A \to B, y : A \vdash e : B}{\Gamma \vdash \mathtt{rec}\ x = \mathtt{fn}\ y = e : A \to B}$$

TABLE 1. Types for CML expressions

For simplicity, we will only use unit, bool, int and chan as base types, although other types such as lists could easily be added.

The *n r* are given by the grammar:

$$n ::= \cdots \mid \mathtt{-1} \mid \mathtt{0} \mid \mathtt{1} \mid \cdots$$

The *nn* are given by the grammar:

$$k ::= \mathtt{a} \mid \mathtt{b} \mid \cdots$$

The are given by the grammar:

$$v ::= \mathtt{true} \mid \mathtt{false} \mid n \mid k \mid () \mid \mathtt{rec}\ x = \mathtt{fn}\ x = e \mid x$$

The *pr on* are given by the grammar:

$$e ::= v \mid ce \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid (e, e) \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid ee$$

Finally, the *n on* are given by the grammar:

$$c ::= \mathtt{fst} \mid \mathtt{snd} \mid \mathtt{add} \mid \mathtt{mul} \mid \mathtt{leq} \mid \mathtt{transmit}_A \mid \mathtt{receive}_A$$
$$\mid \mathtt{choose} \mid \mathtt{spawn} \mid \mathtt{sync} \mid \mathtt{wrap} \mid \mathtt{never}$$

CML is a typed language, with a type system given by the grammar:

$$A ::= \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{chan} \mid A * A \mid A \to A \mid A\ \mathtt{event}$$

The type judgements for expressions are given as judgements $\Gamma \vdash e : A$, where Γ ranges over contexts of the form $x_1 : A_1, \ldots, x_n : A_n$. The type system is in Tables 1 and 2.

We can define syntactic sugar for CML definitions, writing $\mathtt{fn}\ x = e$ for $\mathtt{rec}\ y = \mathtt{fn}\ x = e$ when

$$
\begin{aligned}
\mathtt{fst} &: A * B \rightarrow A \\
\mathtt{snd} &: A * B \rightarrow B \\
\mathtt{add} &: \mathtt{int} * \mathtt{int} \rightarrow \mathtt{int} \\
\mathtt{mul} &: \mathtt{int} * \mathtt{int} \rightarrow \mathtt{int} \\
\mathtt{leq} &: \mathtt{int} * \mathtt{int} \rightarrow \mathtt{bool} \\
\mathtt{transmit}_A &: \mathtt{chan} * A \rightarrow \mathtt{unit}\ \mathtt{event} \\
\mathtt{receive}_A &: \mathtt{chan} \rightarrow A\ \mathtt{event} \\
\mathtt{choose} &: A\ \mathtt{event} * A\ \mathtt{event} \rightarrow A\ \mathtt{event} \\
\mathtt{spawn} &: \mathtt{unit} -\ \ \mathtt{unit} \rightarrow \mathtt{unit} \\
\mathtt{sync} &: A\ \mathtt{event} \rightarrow A \\
\mathtt{wrap} &: A\ \mathtt{event} * (A -\ \ B) \rightarrow B\ \mathtt{event} \\
\mathtt{never} &: \mathtt{unit} \rightarrow A\ \mathtt{event}
\end{aligned}
$$

TABLE 2. Types for CML basic functions

shorthand for projections, and using $\stackrel{\text{def}}{=}$ as shorthand for recursive function declaration. For example, a one-place buffer can be defined:

$$
\begin{aligned}
\mathtt{cell}_A &: \mathtt{chan} * \mathtt{chan} -\ \ B \\
\mathtt{cell}_A(x, y) &\stackrel{\text{def}}{=} \mathtt{cell}_A(\mathtt{snd}(\mathtt{send}_A(y, \mathtt{accept}_A\ x), (
\end{aligned}
$$

we have:

$$\texttt{send (b, send)} \xrightarrow{\texttt{b!send}} \texttt{()}$$

and so we have the higher-order communication:

$$\texttt{send (b, send)} \parallel \texttt{accept b(a,0)} \Longrightarrow \texttt{()} \parallel \texttt{send (a,0)}$$

CML also allows communications of events, so we need to extend the language in a similar fashion to Reppy (1992) to include values of event type. These values are of the form $\texttt{[}ge\texttt{]}$ where ge is a CCS-style r , for example:

$$\begin{aligned}
\texttt{transmit (a,0)} &\Longrightarrow \texttt{[a!0]} \\
\texttt{receive a} &\Longrightarrow \texttt{[a?]} \\
\texttt{choose (transmit (a,0), receive a)} &\Longrightarrow \texttt{[a!0} \oplus \texttt{a?]} \\
\texttt{wrap (receive a, fn } x \texttt{ = } e\texttt{)} &\Longrightarrow \texttt{[a?} \Rightarrow \texttt{fn } x \texttt{ = } e\texttt{]}
\end{aligned}$$

This syntax is based on Reppy's, and is slightly different from that normally associated with process calculi, for example:

- we write $\texttt{a!0} \oplus \texttt{a?}$ rather than $\texttt{a!0} + \texttt{a?}$, and
- we write $\texttt{a?} \Rightarrow \texttt{fn } x \texttt{ = } e$ rather than $\texttt{a?}x.e$.

By extending the syntax of CML expressions to include guarded expressions, we get a particularly simple semantics for `sync` as just removing the outermost level of `[_]`, for example:

$$\begin{aligned}
&\texttt{send (a,0)} \\
&\quad \Longrightarrow \texttt{sync (transmit (a,0))} \\
&\quad \Longrightarrow \texttt{sync [a!0]} \\
&\quad \Longrightarrow \texttt{a!0} \\
&\quad \xrightarrow{\texttt{a!0}} \texttt{()}
\end{aligned}$$

In summary, we give the operational semantics for CML by first extending it to CML$^+$ by adding expressions:

$$e ::= \cdots \mid e \parallel e \mid ge$$

adding values:

$$v ::=$$

$$\frac{e \xrightarrow{\surd v} e'}{ef \xrightarrow{\tau} e' \,\|\, \mathtt{let}\ y = f\ \mathtt{in}\ g[v/x]}\,[v = \mathtt{rec}\ x = \mathtt{fn}\ y = \ g]$$

$$\frac{e \xrightarrow{\surd v} e'}{ce \xrightarrow{\tau} e' \,\|\, \delta(c, v)}$$

$$\frac{e \xrightarrow{\surd \mathtt{true}} e'}{\mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g \xrightarrow{\tau} e' \,\|\, f} \qquad \frac{e \xrightarrow{\surd \mathtt{false}} e'}{\mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g \xrightarrow{\tau} e' \,\|\, g}$$

$$\frac{e \xrightarrow{\surd v} e'}{(e, f) \xrightarrow{\tau} e' \,\|\, \mathtt{let}\ x = f\ \mathtt{in}\ \langle v, x\rangle} \qquad \frac{e \xrightarrow{\surd v} e'}{\mathtt{let}\ x = e\ \mathtt{in}\ f \xrightarrow{\tau} e' \,\|\, f[v/x]}$$

$$\frac{e \xrightarrow{k!_A v} e' \quad f \xrightarrow{k?_A x} f'}{e \,\|\, f \xrightarrow{\tau} e' \,\|\, f'[v/x]} \qquad \frac{e \xrightarrow{k?_A x} e' \quad f \xrightarrow{k!_A v} f'}{e \,\|\, f \xrightarrow{\tau} e'[v/x] \,\|\, f'}$$

TABLE 6. CML operational semantics: silent reductions

$$\overline{v \xrightarrow{\surd v} \delta} \qquad \overline{k!_A v \xrightarrow{k!_A v} ()} \qquad \overline{k?_A \xrightarrow{k?_A x} x}$$

TABLE 7. CML operational semantics: axioms

$$\begin{aligned}
\delta(\mathtt{fst}, \langle v, w\rangle) &= v & \delta(\mathtt{transmit}_A, \langle k, v\rangle) &= [k!_A v]\\
\delta(\mathtt{snd}, \langle v, w\rangle) &= w & \delta(\mathtt{receive}_A, k) &= [k?_A]\\
\delta(\mathtt{add}, \langle m, n\rangle) &= m + n & \delta(\mathtt{choose}, \langle [ge_1], [ge_2]\rangle) &= [ge_1 \oplus ge_2]\\
\delta(\mathtt{mul}, \langle m, n\rangle) &= m \times n & \delta(\mathtt{wrap}, \langle [ge], v\rangle) &= [ge \Rightarrow v]\\
\delta(\mathtt{leq}, \langle m, n\rangle) &= m \leq n & \delta(\mathtt{spawn}, v) &= v() \,\|\, ()\\
\delta(\mathtt{sync}, [ge]) &= ge & \delta(\mathtt{never}, ()) &= [\delta]
\end{aligned}$$

TABLE 8. CML operational semantics: basic functions

$\mathtt{cell}_A \langle i, o\rangle$

$\xrightarrow{\tau}$ $\mathtt{let}\ x = \langle i, o\rangle\ \mathtt{in}\ \mathtt{cell}_A\ (\mathtt{snd}\ (\mathtt{send}_A\ (\mathtt{snd}\ x, \mathtt{accept}_A$ [redacted]

$\xrightarrow{\tau}$ $\mathtt{cell}_A\ (\mathtt{snd}\ (\mathtt{send}_A\ (\mathtt{snd}\langle i, o\rangle, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle))$ [redacted]

$\xrightarrow{\tau}$ $\mathtt{let}\ x = \mathtt{snd}\ (\mathtt{send}_A\ (\mathtt{snd}\langle i, o\rangle, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle)),$ [redacted]

$\mathtt{in}\ \mathtt{cell}_A\ x$

$\xrightarrow{\tau}$ $\mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = (\mathtt{snd}\langle i, o\rangle, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle))$

$, o \quad \langle i, o$ $\mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y)$

[redacted]

$\xrightarrow{\tau}$ [redacted] $(\mathtt{fst}\langle i, o\rangle))$

[redacted] $y)$

[redacted]

obvious.

Let a *o p , n* relation $\mathcal{R}$ be an open type-indexed relation where Γ is everywhere the empty context, and can therefore be elided.

For any closed type-indexed relation $\mathcal{R}$, let its *op n n on* $\mathcal{R}^\circ$ be defined as:

pleted:

$$\begin{tikzcd} \mathsf{e}_1 \arrow[d, "1"'] & \mathcal{R} & \mathsf{e}_2 & \text{as} & \mathsf{e}_1 \arrow[d, "1"'] & \mathcal{R} & \mathsf{e}_2 \arrow[d, Rightarrow, "\hat{2}"'] \\ \mathsf{e}_1' & & & & \mathsf{e}_1' & \mathcal{R}^\circ & \mathsf{e}_2' \end{tikzcd} \quad \text{where } 1\,\mathcal{R}\,2$$

A r or r on is a higher-order weak simulation whose inverse is also a higher-order weak simulation. Let $\approx$ be the largest higher-order weak bisimulation.

Proposition 3. $\approx$ *on r n -*

Proof. Given in (Ferreira, Hennessy, and Jeffrey 1995), using a variant of Gordon's (1995) presentation Howe's (1989) proof technique. Note that this proof relies on the fact that we are considering the subset of CML without `always`, and hence do not have to consider initial τ-actions in summations, which present the same problems as in the first-order case (Milner 1989). $\square$

Unfortunately, this equivalence does not have many pleasant mathematical properties. For example none of the usual equations for products are true:

$$\begin{aligned} \mathtt{fst}\,(e, f) &\not\approx e \\ \mathtt{snd}\,(e, f) &\not\approx f \\ (\mathtt{fst}\,e, \mathtt{snd}\,e) &\not\approx e \end{aligned}$$

(For each counter-example consider an expression with side-effects, such as `cell`.)

In the next section we shall consider a variant of CML which uses a restrictive type system to provide more pleasant mathematical properties of programs. We shall then show a translation from CML into the restricted language, which is correct up to weak bisimulation.

3 Concurrent monadic ML

In the previous section, we showed how to define an operational semantics for CML which can be used as the basis of a bisimulation equivalence between programs. Unfortunately, this equivalence does not have tssTf17.048TLs6.81111(a)-4.11265

s)11.696(t)-5.01912(e)-4.11265(r)26.3929(-)2.811.696(5.01912(n)4.71816(i)9401067.56(f)2.b50596(o7

Using an explicit type constructor for computation has the advantage that the only terms which perform computation are those of type $A\,\mathsf{comp}$, and that an expression of any other type is guaranteed to be in normal for

$$\frac{e \xrightarrow{\alpha} e'}{\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f \xrightarrow{\alpha} \mathsf{let}\, x \Leftarrow e' \,\mathsf{in}\, f}$$

$$\frac{e \xrightarrow{\alpha} e'}{e \,||\, f \xrightarrow{\alpha} e' \,||\, f} \qquad \frac{f \longrightarrow f'}{e \,||\, f \longrightarrow e \,||\, f'} \qquad \frac{e \xrightarrow{\tau} e'}{e \,\square\, f \xrightarrow{\tau} e' \,\square\, f} \qquad \frac{f \xrightarrow{\tau} f'}{e \,\square\, f \xrightarrow{\tau} e \,\square\, f'}$$

TABLE 12. CMML operational semantics: static rules

In the operational semantics of CML, terms in many contexts can reduce, whereas there are far fewer reduction contexts in CMML. In fact, looking at the sequential sub-language of CMML (without $||$ or $\square$) the only reduction context is let:

$$\frac{e \xrightarrow{\alpha} e'}{\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f \xrightarrow{\alpha} \mathsf{let}\, x \Leftarrow e' \,\mathsf{in}\, f}$$

Many of the operational rules in CML require spawning off concurrent processes, whereas in CMML the main rule which produces extra concurrent processes is β-reduction for let-expressions:

e

ular we require the lts to be r n :

$$e \xrightarrow{\checkmark f} e'$$

if then = and ' = ''

$$[\![\texttt{true}]\!] = \mathsf{true}$$
$$[\![\texttt{false}]\!] = \mathsf{false}$$
$$[\![n$$

$$
\begin{array}{ll}
[\![\texttt{cell}]\!] \approx & \mathsf{rec}\, x_1 = \mathsf{fn}\, x_2 \Rightarrow \\
& \mathsf{let}\, x_3 \Leftarrow [x_1] \\
& \mathsf{in}\, \mathsf{let}\, x_4 \Leftarrow \mathsf{let}\, x_5 \Leftarrow \mathsf{let}\, x_6 \Leftarrow \mathsf{let}\, x_8 \Leftarrow \mathsf{let}\, x_9 \Leftarrow \mathsf{let}\, x_{11} \Leftarrow [x_2] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, [x_{11}.\mathsf{r}] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, \mathsf{let}\, x_{10} \Leftarrow \mathsf{let}\, x_{12} \Leftarrow \mathsf{let}\, x_{13} \Leftarrow [x_2] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, [x_{13}.\mathsf{l}] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, x_{12}? \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, [\langle x_9, x_{10} \rangle] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, x_8.\mathsf{l}!x_8.\mathsf{r} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, \mathsf{let}\, x_7 \Leftarrow [x_2] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\, [\langle x_6, x_7 \rangle] \\
& \qquad\qquad\qquad\qquad \mathsf{in}\, [x_5.\mathsf{r}] \\
& \quad \mathsf{in}\, x_3\, x_4
\end{array}
$$

TABLE 20. Example translation of CML$^+$ into CMML

ciency. This suggests that CMML may be a suitable virtual machine language for a CML compiler, where verifiable peephole optimizations can be performed.

4- *Corr n o r n on*

We will now show that the translation of CML$^+$ into CMML is correct up to bisimulation. We will do this by defining an appropriate notion of weak bisimulation between CML and CMML programs. This proof uses Milner and Sangiorgi's (1992) technique of 'bisimulation up to'.

A *o p , n r on n* CML

such that the following diagrams can be completed:

$$\begin{tikzcd} e_1 \arrow[d, "1"'] & \mathcal{R} & e_2 \\ e_1' & & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "1"'] & \mathcal{R} & e_2 \arrow[d, Rightarrow, "2"'] \\ e_1' & \mathcal{R}^\circ & e_2' \end{tikzcd} \quad \text{where } {}_1 \mathcal{R} {}_2$$

and:

$$\begin{tikzcd} e_1 & \mathcal{R} & e_2 \arrow[d, "2"'] \\ & & e_2' \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "\hat{1}"'] & \mathcal{R} & e_2 \arrow[d, "2"'] \\ e_1' & \mathcal{R}^\circ & e_2' \end{tikzcd} \quad \text{where } {}_1 \mathcal{R} {}_2$$

Let $\lesssim$ be the largest expansion.

Proposition 6. $\lesssim$ *pr on r n on* CML *n* CMML-

Proof. Similar to Proposition 3. □

For example, the preorder $\leq_\beta$ given by β-reducing in all contexts is an expansion:

$$\frac{}{e\,f \geq_\beta g[f/y][e/x]}[e = (\mathsf{rec}\, x = \mathsf{fn}\, y \Rightarrow g)] \qquad \frac{}{\mathsf{let}\, x \Leftarrow [e]\, \mathsf{in}\, f \geq_\beta f[e/x]}$$

$$\frac{}{\mathsf{if\,true\,then}\, f\, \mathsf{else}\, g \geq_\beta f} \qquad \frac{}{\mathsf{if\,false\,then}\, f\, \mathsf{else}\, g \geq_\beta g}$$

$$\frac{}{e \geq_\beta e} \qquad \frac{e \geq_\beta f \geq_\beta g}{e \geq_\beta g} \qquad \frac{e \geq_\beta f}{C[e] \geq_\beta C[f]}$$

Proposition 7. $e \leq_\beta f$ *n* $e \lesssim f$-

Proof. Show that each of the axioms forms an expansion. The result then follows from Proposition 6. □

We can use the proof technique of strong bisimulation up to $(\leq, \sqsubseteq)$ to show that the translation from CML to CMML forms a weak bisimulation.

Proposition 8. *An ron on p o* $(\,, \lesssim)$ *on-*

Proof. An adaptation of the results in (Sangiorgi and Milner 1992). □

Proposition 9. *r n on o* CML$^+$ *n o* CMML *ron* , *on p o* $(\geq_\beta, \leq_\beta)$-

Proof. Let $\mathcal{R}$ be:

$$\mathcal{R}_A = \{(e, E[\![e]\!]) \mid \vdash e : A\} \qquad \mathcal{R}_A = \{(v, \ [\![v]\!]) \mid \vdash v : A\}$$

and:

$$
\begin{aligned}
&\mathsf{let}\, x \Leftarrow e' \,\mathsf{in} \quad [\![v]\!]\, x \\
&\quad \geq_\beta \mathsf{let}\, x \Leftarrow E[\![\mathtt{e}']\!] \,\mathsf{in} \quad [\![v]\!]\, x \\
&\quad \geq_\beta \mathsf{let}\, x \Leftarrow E[\![\mathtt{e}']\!] \,\mathsf{in}\, E[\![g]\!][x/z][\quad [\![v]\!]/y] \\
&\quad = E[\![\mathtt{let}\, z = \mathtt{e}' \,\mathtt{in}\, g[v/y]]\!]
\end{aligned}
$$

The other cases are similar. □

Proposition 10. e r o $E[\![\mathtt{e}]\!]$-

Proof. Follows from Propositions 7, 8 and 9. □