Syntax and semantics of a GPU kernel programming language

John Wickerson

May 28, 2015

Abstract

This document accompanies the article The Design and Implementation of a Verification Technique for GPU Kernels by Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson and John Wickerson [1]. It formalises all of the definitions provided in Sections 3 and 4 of the article.

Contents

1 General purpose definitions and lemmas 1
2 Syntax of KPL 2
3 Well-formedness of KPL kernels 3
4 Thread, group and kernel states 5
5 Execution rules for threads 7
6 Execution rules for groups 8
7 Execution rules for kernels 11

1 General purpose definitions and lemmas

theory Mise imports
  Main
begin

A handy abbreviation when working with maps

abbreviation make-map :: 'a set ⇒ 'b ⇒ ('a → 'b) ([ - |=> - ])
where
[k s |=> v] ≡ λk. if k ∈ ks then Some v else None
Projecting the components of a triple

**definition** \( \text{fst3} \equiv \text{fst} \)

**definition** \( \text{snd3} \equiv \text{fst} \circ \text{snd} \)

**definition** \( \text{thd3} \equiv \text{snd} \circ \text{snd} \)

**lemma** \( \text{fst3-simp} \ [\text{simp}] \): \( \text{fst3} \ (a,b,c) = a \) by (simp add: \( \text{fst3-def} \))

**lemma** \( \text{snd3-simp} \ [\text{simp}] \): \( \text{snd3} \ (a,b,c) = b \) by (simp add: \( \text{snd3-def} \))

**lemma** \( \text{thd3-simp} \ [\text{simp}] \): \( \text{thd3} \ (a,b,c) = c \) by (simp add: \( \text{thd3-def} \))

end

2 Syntax of KPL

theory **KPL-syntax** imports Misc begin

Locations of local variables

typedecl \( V \)

C strings

typedecl \( \text{name} \)

Procedure names

typedecl \( \text{proc-name} \)

Local-id, group-id

type-synonym \( \text{lid} = \text{nat} \)

type-synonym \( \text{gid} = \text{nat} \)

Fully-qualified thread-id

type-synonym \( \text{tid} = \text{gid} \times \text{lid} \)

Let \((G, T)\) range over threadsets

type-synonym \( \text{threadset} = \text{gid set} \times (\text{gid} \rightarrow \text{lid set}) \)

Returns the set of tids in a threadset

fun \( \text{tids :: threadset} \Rightarrow \text{tid set} \)

where

\[ \text{tids} \ (G,T) = \{ (i,j) \mid i, j \in G \land j \in \text{the} \ (T \ i) \} \]

type-synonym \( \text{word} = \text{nat} \)

datatype \( \text{loc} = \)

\[ \text{Name \ name} \]

\| \ Var \ V \]
Local expressions

datatype local-expr =
    Loc loc
    | Gid
    | Lid
    | eTrue
    | eConj local-expr local-expr (infixl ∧ 50)
    | eNot local-expr (¬)

Basic statements

datatype basic-stmt =
    Assign loc local-expr
    | Read loc local-expr
    | Write local-expr local-expr

Statements

datatype stmt =
    Basic basic-stmt
    | Seq stmt stmt (infixl ;; 50)
    | Local name stmt
    | If local-expr stmt stmt
    | While local-expr stmt
    | WhileDyn local-expr stmt
    | Call proc-name local-expr
    | Barrier
    | Break
    | Continue
    | Return

Procedures comprise a procedure name, parameter name, and a body statement

record proc =
    proc-name :: proc-name
    param :: name
    body :: stmt

Kernels

record kernel =
    groups :: nat
    threads :: nat
    procs :: proc list
    main :: stmt

end

3 Well-formedness of KPL kernels

theory KPL-wellformedness imports
KPL-syntax

begin

Well-formed local expressions. \textit{wf-local-expr ns e} means that

\begin{itemize}
  \item e does not mention any internal locations, and
  \item any name mentioned by e is in the set \textit{ns}.
\end{itemize}

\textbf{fun} \textit{wf-local-expr :: name set ⇒ local-expr ⇒ bool}

\textbf{where}
\begin{align*}
\textit{wf-local-expr ns (Loc (Var j))} &= \text{False} \\
\textit{wf-local-expr ns (Loc (Name n))} &= (n \in \textit{ns}) \\
\textit{wf-local-expr ns (e1 \& e2)} &= \left(\textit{wf-local-expr ns e1} \land \textit{wf-local-expr ns e2}\right) \\
\textit{wf-local-expr ns (¬e)} &= \textit{wf-local-expr ns e} \\
\textit{wf-local-expr ns } - &= \text{True}
\end{align*}

Well-formed basic statements. \textit{wf-basic-stmt ns b} means that

\begin{itemize}
  \item b does not mention any internal locations, and
  \item any name mentioned by b is in the set \textit{ns}.
\end{itemize}

\textbf{fun} \textit{wf-basic-stmt :: name set ⇒ basic-stmt ⇒ bool}

\textbf{where}
\begin{align*}
\textit{wf-basic-stmt ns (Assign x e)} &= \textit{wf-local-expr ns e} \\
\textit{wf-basic-stmt ns (Read x e)} &= \textit{wf-local-expr ns e} \\
\textit{wf-basic-stmt ns (Write e1 e2)} &= \left(\textit{wf-local-expr ns e1} \land \textit{wf-local-expr ns e2}\right)
\end{align*}

Well-formed statements. \textit{wf-stmt ns F S} means:

\begin{itemize}
  \item S only calls procedures whose name is in F,
  \item S does not contain \textit{WhileDyn},
  \item S does not mention internal variables,
  \item S only mentions names in \textit{ns}, and
  \item S does not declare the same name twice, e.g. \textit{Local x (Local x foo)}.
\end{itemize}

\textbf{fun} \textit{wf-stmt :: name set ⇒ proc-name set ⇒ stmt ⇒ bool}

\textbf{where}
\begin{align*}
\textit{wf-stmt ns F (Basic b)} &= \textit{wf-basic-stmt ns b} \\
\textit{wf-stmt ns F (S1 ; S2)} &= \left(\textit{wf-stmt ns F S1} \land \textit{wf-stmt ns F S2}\right) \\
\textit{wf-stmt ns F (Local n S)} &= (n \notin \textit{ns} \land \textit{wf-stmt} \left(\{n\} \cup \textit{ns}\right) F S) \\
\textit{wf-stmt ns F (If e S1 S2)} &= \left(\textit{wf-local-expr ns e} \land \textit{wf-stmt ns F S1} \land \textit{wf-stmt ns F S2}\right)
\end{align*}
\| \text{wf-stmt}\ ns\ F\ (\text{While}\ e\ S) = \\
(wx-local-expr\ ns\ e\ \land\ \text{wf-stmt}\ ns\ F\ S) \\
| \text{wf-stmt}\ ns\ F\ (\text{WhileDyn}\ -\ -) = \text{False} \\
| \text{wf-stmt}\ ns\ F\ (\text{Call}\ f\ e) = (f \in F \land wx-local-expr\ ns\ e) \\
| \text{wf-stmt}\ -\ -\ - = \text{True} \\

no-return\ S\ holds\ if\ S\ does\ not\ contain\ a\ \text{Return}\ statement

\textbf{fun no-return::}\ stmt \Rightarrow bool \\
\textbf{where} \\
no-return\ (S1 \;;\;;\ S2) = (no-return\ S1 \land no-return\ S2) \\
novation\ (Local\ n\ S) = no-return\ S \\
novation\ (If\ e\ S1\ S2) = (no-return\ S1 \land no-return\ S2) \\
novation\ (While\ e\ S) = (no-return\ S) \\
novation\ Return = False \\
novation\ - = \text{True} \\

\textbf{Well-formed\ kernel}

\textbf{definition} \text{wf-kernel::}\ kernel \Rightarrow bool \\
\textbf{where} \\
\text{wf-kernel}\ P \equiv \\
let\ F = \text{set}\ (\text{map}\ \text{proc-name}\ (\text{procs}\ P))\ in \\
\begin{align*} 
&(*)\ \text{The\ main\ statement\ must\ not\ refer\ to\ any}\ * \\
&\text{variable,\ except\ those\ it\ locally\ defines.}\ * \\
&\text{wf-stmt}\ \{\}\ F\ (\text{main}\ P) \\
\end{align*} \\
\begin{align*} 
&(*)\ \text{The\ main\ statement\ contains\ no\ return\ statement.}\ * \\
&\land\ \text{no-return}\ (\text{main}\ P) \\
\end{align*} \\
\begin{align*} 
&(*)\ A\ \text{procedure\ body\ may\ refer\ only\ to\ its\ argument.}\ * \\
&\land\ \text{list-all}\ (\lambda f.\ \text{wf-stmt}\ \{\text{param}\ f\}\ F\ (\text{body}\ f))\ (\text{procs}\ P) \\
\end{align*} \\
\textbf{end}

4 \textbf{Thread, group and kernel states}

theory \text{KPL-state} \\
\textbf{imports} \text{KPL-syntax} \\
\textbf{begin} \\
Thread\ state \\
\textbf{record} \text{thread-state} = \\
\begin{align*} 
&l :: V + \text{bool} \Rightarrow \text{word} \\
&sh :: \text{nat} \Rightarrow \text{word} \\
&R :: \text{nat set} \\
&W :: \text{nat set} \\
\end{align*} \\
\textbf{end}
abbreviation $GID \equiv \text{Inr True}$
abbreviation $LID \equiv \text{Inr False}$

Group state

record group-state =
  thread-states :: lid $\rightarrow$ thread-state (- $ts [1000] 1000)$
  R-group :: (lid $\times$ nat) set
  W-group :: (lid $\times$ nat) set

Valid group state

fun valid-group-state :: (gid $\rightarrow$ lid set) $\Rightarrow$ gid $\Rightarrow$ group-state $\Rightarrow$ bool
where
  valid-group-state $T$ i $\gamma$ =
  dom $(\gamma$ $ts)$ = the $(T$ i) $\land$
  $l$ (the $(\gamma$ $ts$ $j)$) $GID$ = i $\land$
  $l$ (the $(\gamma$ $ts$ $j)$) $LID$ = j)

Predicated statements

type-synonym pred-stmt = stmt $\times$ local-expr
type-synonym pred-basic-stmt = basic-stmt $\times$ local-expr

Kernel state

type-synonym kernel-state =
  (gid $\rightarrow$ group-state) $\times$ pred-stmt list $\times$ V list

Valid kernel state

fun valid-kernel-state :: threadset $\Rightarrow$ kernel-state $\Rightarrow$ bool
where
  valid-kernel-state $(G, T)$ ($\kappa$, ss, -) =
  dom $\kappa$ = G $\land$
  ($\forall$ i $\in$ G. valid-group-state $T$ i (the ($\kappa$ i))))

Valid initial kernel state

fun valid-initial-kernel-state :: strat $\Rightarrow$ threadset $\Rightarrow$ kernel-state $\Rightarrow$ bool
where
  valid-initial-kernel-state $S$ $(G, T)$ ($\kappa$, ss, vs) =
  valid-kernel-state $(G, T)$ ($\kappa$, ss, vs) $\land$
  (ss = [(S, eTrue)]) $\land$
  ($\forall$ i $\in$ G. R-group (the ($\kappa$ i)) = {} $\land$ W-group (the ($\kappa$ i)) = {}) $\land$
  ($\forall$ i $\in$ G. $\forall$ j $\in$ the $(T$ i). R (the ((the ($\kappa$ i)$ts$ $j)$)ts $j$)) = {} $\land$
  W (the ((the ($\kappa$ i)$ts$ $j)$)ts $j$)) = {} $\land$
  ($\forall$ i $\in$ G. $\forall$ j $\in$ the $(T$ i). $\forall$ v :: V.
    $l$ (the ((the ($\kappa$ i)$ts$ $j)$) (Int v) = 0) $\land$
  ($\forall$ i $\in$ G. $\forall$ i' $\in$ G. $\forall$ j $\in$ the $(T$ i). $\forall$ j' $\in$ the $(T$ i').
    sh (the ((the ($\kappa$ i)$ts$ $j)$)ts $j$)) =
    sh (the ((the ($\kappa$ i')$ts$ $j'$))) $\land$
5 Execution rules for threads

theory KPL-execution-thread imports KPL-state begin

Evaluate a local expression down to a word
fun eval-word :: local-expr ⇒ thread-state ⇒ word
where
eval-word (Loc (Var v)) τ = l τ (Inl v)
| eval-word Lid τ = l τ LID
| eval-word Gid τ = l τ GID
| eval-word eTrue τ = (eval-word e1 τ * eval-word e2 τ)
| eval-word (∼e) τ = (if eval-word e τ = 0 then 1 else 0)

Evaluate a local expression down to a boolean
fun eval-bool :: local-expr ⇒ thread-state ⇒ bool
where
eval-bool e τ = (eval-word e τ ≠ 0)

Abstraction level: none, equality abstraction, or adversarial abstraction
datatype abs-level = No-Abst | Eq-Abst | Adv-Abst

The rules of Figure 4, plus two additional rules for adversarial abstraction (Fig 7b)
inductive step-t :: abs-level ⇒ (thread-state × pred-basic-stmt) ⇒ thread-state ⇒ bool
where
T-Disabled:
¬ (eval-bool p τ) ⇒ step-t a (τ, (b, p)) τ
| T-Assign:
[ eval-bool p τ ; l' = (l τ) (Inl v := eval-word e τ) ]
⇒ step-t a (τ, (Assign (Var v) e, p)) (τ (| l := l' |))
| T-Read:
[ eval-bool p τ ; l' = (l τ) (Inl v := sh τ (eval-word e τ)) ;
R' = R τ ∪ { eval-word e τ } ; a ∈ {No-Abst, Eq-Abst} ]
⇒ step-t a (τ, (Read (Var v) e, p)) (τ (| l := l' , R := R' |))
| T-Write:
[ eval-bool p τ ;
sh' = (sh τ) (eval-word e1 τ := eval-word e2 τ) ;
W' = W τ ∪ { eval-word e1 τ } ; a ∈ {No-Abst, Eq-Abst} ]
\[ \quad \Rightarrow \text{step-t a} (\tau, (\text{Write } e_1 e_2, p)) (\tau (| \text{sh} := \text{sh}', W := W')); \]

\[ \mid \text{T-Read-Adv:} \]
\[ \quad [\text{eval-bool } p \tau; l' = l \tau (\text{Inl } v := \text{asterisk}); \]
\[ \quad \quad R' = R \tau \cup \{ \text{eval-word } e \tau \}] \]
\[ \quad \Rightarrow \text{step-t Adv-Abst} (\tau, (\text{Read} (\text{Var } v) e, p)) (\tau (| l := l', R := R')); \]

\[ \mid \text{T-Write-Adv:} \]
\[ \quad [\text{eval-bool } p \tau; W' = W \tau \cup \{ \text{eval-word } e \tau \}] \]
\[ \quad \Rightarrow \text{step-t Adv-Abst} (\tau, (\text{Write } e_1 e_2, p)) (\tau (| \ast \text{sh} := \text{sh}', \ast \text{W} := W')); \]

Rephrasing \text{T-Assign} to make it more usable

\[ \text{lemma T-Assign-helper:} \]
\[ \quad [\text{eval-bool } p \tau; l' = l \tau (\text{Inl } v := \text{eval-word } e \tau); \]
\[ \quad \quad \tau' = \tau (| l := l', R := R') \]
\[ \Rightarrow \text{step-t a} (\tau, (\text{Assign} (\text{Var } v) e, p)) \tau'; \]

by (auto simp add: step-t.T-Assign)

Rephrasing \text{T-Read} to make it more usable

\[ \text{lemma T-Read-helper:} \]
\[ \quad [\text{eval-bool } p \tau; W' = W \tau \cup \{ \text{eval-word } e \tau \}; \]
\[ \quad \quad a \in \{ \text{No-Abst, Eq-Abst} \}; \]
\[ \quad \quad \tau' = \tau (| l := l', R := R') \]
\[ \Rightarrow \text{step-t a} (\tau, (\text{Read} (\text{Var } v) e, p)) \tau'; \]

by (auto simp add: step-t.T-Read)

Rephrasing \text{T-Write} to make it more usable

\[ \text{lemma T-Write-helper:} \]
\[ \quad [\text{eval-bool } p \tau; \]
\[ \quad \quad \text{sh}' = (\text{sh } \tau) (\text{eval-word } e_1 \tau := \text{eval-word } e_2 \tau); \]
\[ \quad \quad W' = W \tau \cup \{ \text{eval-word } e_1 \tau \}; \]
\[ \quad \quad a \in \{ \text{No-Abst, Eq-Abst} \}; \]
\[ \quad \quad \tau' = \tau (| \ast \text{sh} := \text{sh}', \ast \text{W} := W'); \]
\[ \Rightarrow \text{step-t a} (\tau, (\text{Write } e_1 e_2, p)) \tau'; \]

by (auto simp add: step-t.T-Write)

end

6 Execution rules for groups

theory KPL-execution-group imports
KPL-execution-thread
begin

Intra-group race detection

\[ \text{definition group-race} \]
\[ := \text{lid set} \Rightarrow (\text{lid} \rightarrow \text{thread-state}) \Rightarrow \text{bool} \]

where group-race \text{T } \gamma \equiv \]
\[ \exists j \in \text{T}. \exists k \in \text{T}. j \neq k \land \]
\[ W \ (\text{the } (\gamma j)) \cap (R \ (\text{the } (\gamma k)) \cup W \ (\text{the } (\gamma k))) \neq \{\} \]

The constraints for the \text{merge} map

end
inductive `pre-merge`

```plaintext
:: lid set ⇒ (lid → thread-state) ⇒ nat ⇒ word ⇒ bool
where
[ j ∈ T ; z ∈ W (the (γ j)) ; dom γ = T ] ⇒
pre-merge T γ z (sh (the (γ j))) z
| [ ∀ j ∈ T. z /∈ W (the (γ j)) ; dom γ = T ] ⇒
pre-merge T γ z (sh (the (γ 0))) z
```

inductive-cases `pre-merge-inv [elim!]`: `pre-merge P γ z z'`

The `merge` map maps each nat to the word that satisfies the above constraints. The `merge-is-unique` lemma shows that there exists exactly one such word per nat, provided there are no group races.

**Definition**

```plaintext
merge :: lid set ⇒ (lid → thread-state) ⇒ nat ⇒ word
where
merge T γ ≡ λ z. The (pre-merge T γ z)
```

**Lemma** `no-races-imp-no-write-overlap`:

```plaintext
¬ (group-race T γ) ⇒
∀ i ∈ T. ∀ j ∈ T.
  i ≠ j −→ W (the (γ i)) ∩ W (the (γ j)) = {} 
```

unfolding `group-race-def` by `blast`

**Lemma** `merge-is-unique`:

```plaintext
assumes dom γ = T
assumes ¬ (group-race T γ)
shows ∃! z'. pre-merge T γ z z'
apply (insert assms)
apply (drule no-races-imp-no-write-overlap)
apply (intro allI ex-ex1I)
apply (metis pre-merge.intros)
apply clarify
proof −
  fix z1 z2
  assume a: ∀ i ∈ dom γ. ∀ j ∈ dom γ. i ≠ j −→ W (the (γ i)) ∩ W (the (γ j)) = {}
  assume pre-merge (dom γ) γ z z1
  and pre-merge (dom γ) γ z z2
  thus z1 = z2
  apply (elim pre-merge-inv)
  apply (rename-tac j1 j2)
  apply (case-tac j1 = j2)
  apply auto[1]
  apply simp
  apply (subgoal-tac W (the (γ j1)) ∩ W (the (γ j2)) = {})
  apply auto[1]
  apply (auto simp add: a)
  done
qed
```
The rules of Figure 5, plus an additional rule for equality abstraction (Fig 7a), plus an additional rule for adversarial abstraction (Fig 7b)

**inductive step-g**

:: abs-level ⇒ gid ⇒ (gid → lid set) ⇒ (group-state × pred-stmt) ⇒ group-state option ⇒ bool

**where**

**G-Race:**

\[ \forall j \in \text{the } (T i). \text{step-t a } (\text{the } (\gamma_{ts} j), (s, p)) (\text{the } (\gamma'_{ts} j)) ; \text{group-race } (\text{the } (T i)) ((\gamma':: \text{group-state})_{ts}) \]

⇒ step-g a i T (γ, (Basic s, p)) None

| **G-Basic:**

\[ \forall j \in \text{the } (T i). \text{step-t a } (\text{the } (\gamma_{ts} j), (s, p)) (\text{the } (\gamma'_{ts} j)) ; \neg (\text{group-race } (\text{the } (T i)) (\gamma'_{ts})) ; \]

\[ R\text{-group } \gamma' = R\text{-group } \gamma \cup (\bigcup j \in \text{the } (T i). (\{j\} \times R (\text{the } (\gamma'_{ts} j)))) ; \]

\[ W\text{-group } \gamma' = W\text{-group } \gamma \cup (\bigcup j \in \text{the } (T i). (\{j\} \times W (\text{the } (\gamma'_{ts} j)))) \]

⇒ step-g a i T (γ, (Basic s, p)) (Some γ')

| **G-No-Op:**

\[ \forall j \in \text{the } (T i). \neg (\text{eval-bool } p (\text{the } (\gamma_{ts} j))) \]

⇒ step-g a i T (γ, (Barrier, p)) (Some γ)

| **G-Divergence:**

\[ \forall j \in \text{the } (T i). \neg (\text{eval-bool } p (\text{the } (\gamma_{ts} j))) ; \]

⇒ step-g a i T (γ, (Barrier, p)) None

| **G-Sync:**

\[ \forall j \in \text{the } (T i). \text{eval-bool } p (\text{the } (\gamma_{ts} j)) ; \]

\[ \forall j \in \text{the } (T i). (\gamma'_{ts} j) = (\text{the } (\gamma_{ts} j)) (| \]

\[ \text{sh := } \text{merge } P (\gamma_{ts}), R := \{\}, W := \{\{\} \} ) \]

⇒ step-g No-Abst i T (γ, (Barrier, p)) (Some γ')

| **G-Sync-Eq:**

\[ \forall j \in \text{the } (T i). \text{eval-bool } p (\text{the } (\gamma_{ts} j)) ; \]

\[ \forall j \in \text{the } (T i). (\gamma'_{ts} j) = (\text{the } (\gamma_{ts} j)) (| \]

\[ \text{sh := } \text{sh}', R := \{\}, W := \{\{\} \} ) \]

⇒ step-g Eq-Abst i T (γ, (Barrier, p)) (Some γ')

| **G-Sync-Adv:**

\[ \forall j \in \text{the } (T i). \text{eval-bool } p (\text{the } (\gamma_{ts} j)) ; \]

\[ \forall j \in \text{the } (T i). \exists \text{sh}'. (\gamma'_{ts} j) = (\text{the } (\gamma_{ts} j)) (| \]

\[ \text{sh := } \text{sh}', R := \{\}, W := \{\{\} \} ) \]

⇒ step-g Adv-Abst i T (γ, (Barrier, p)) (Some γ')

**Rephrasing G-No-Op to make it more usable**

**lemma G-No-Op-helper:**

\[ \forall j \in \text{the } (T i). \neg (\text{eval-bool } p (\text{the } (\gamma_{ts} j))) ; \gamma = \gamma' \]

⇒ step-g a i T (γ, (Barrier, p)) (Some γ')

**by (simp add: step-g.G-No-Op)**

end
7 Execution rules for kernels

theory KPL-execution-kernel imports KPL-execution-group begin

Inter-group race detection

definition kernel-race :: gid set ⇒ (gid ⇒ group-state) ⇒ bool
where kernel-race G κ ≡
∃ i ∈ G. ∃ j ∈ G. i ≠ j ∧
(snd ⋃ (W-group (the (κ i)))) ∩
(snd ⋃ (R-group (the (κ j)))) ∪ snd ⋃ (W-group (the (κ j)))) ≠ {}

Replaces top-level Break with v := true
fun belim :: stmt ⇒ V ⇒ stmt
where
belim (Basic b) v = Basic b
| belim (S1 ;; S2) v = (belim S1 v ;; belim S2 v)
| belim (Local n S) v = Local n (belim S v)
| belim (If e S1 S2) v = If e (belim S1 v) (belim S2 v)
| belim (While e S) v = While e S
| belim (Call f e) v = Call f e
| belim Break v = Basic (Assign (Var v) eTrue)
| belim Continue v = Continue
| belim Return v = Return

Replaces top-level Continue with v := true
fun celim :: stmt ⇒ V ⇒ stmt
where
celim (Basic b) v = Basic b
| celim (S1 ;; S2) v = (celim S1 v ;; celim S2 v)
| celim (Local n S) v = Local n (celim S v)
| celim (If e S1 S2) v = If e (celim S1 v) (celim S2 v)
| celim (While e S) v = While e S
| celim (Call f e) v = Call f e
| celim Break v = Break
| celim Continue v = Basic (Assign (Var v) eTrue)
| celim Return v = Return

subst-basic-stmt n v loc replaces n with v inside loc
fun subst-loc :: name ⇒ V ⇒ loc ⇒ loc
where
subst-loc n v (Var w) = Var w
| subst-loc n v (Name m) = (if n = m then Var v else Name m)
subst-local-expr \( n \) \( v \) \( e \) replaces \( n \) with \( v \) inside \( e \)

fun subst-local-expr :: name ⇒ V ⇒ local-expr ⇒ local-expr
where
  subst-local-expr \( n \) \( v \) (Loc \( \text{loc} \)) = Loc (subst-loc \( n \) \( v \) \( \text{loc} \))
  subst-local-expr \( n \) \( v \) Gid = Gid
  subst-local-expr \( n \) \( v \) Lid = Lid
  subst-local-expr \( n \) \( v \) \( e \)True = eTrue
  subst-local-expr \( n \) \( v \) \( e \)1 \&\& \( e \)2 =
    (subst-local-expr \( n \) \( v \) \( e \)1 \&\& subst-local-expr \( n \) \( v \) \( e \)2)
  subst-local-expr \( n \) \( v \) ¬\&\& \( e \) = ¬\&\& (subst-local-expr \( n \) \( v \) \( e \))

subst-basic-stmt \( n \) \( v \) \( b \) replaces \( n \) with \( v \) inside \( b \)

fun subst-basic-stmt :: name ⇒ V ⇒ basic-stmt ⇒ basic-stmt
where
  subst-basic-stmt \( n \) \( v \) (Assign \( \text{loc} \) \( e \)) =
    Assign (subst-loc \( n \) \( v \) \( \text{loc} \)) (subst-local-expr \( n \) \( v \) \( e \))
  subst-basic-stmt \( n \) \( v \) (Read \( \text{loc} \) \( e \)) =
    Read (subst-loc \( n \) \( v \) \( \text{loc} \)) (subst-local-expr \( n \) \( v \) \( e \))
  subst-basic-stmt \( n \) \( v \) (Write \( e \)1 \( e \)2) =
    Write (subst-local-expr \( n \) \( v \) \( e \)1) (subst-local-expr \( n \) \( v \) \( e \)2)

subst-stmt \( n \) \( v \) \( s \) \( t \) holds if \( t \) is the result of replacing \( n \) with \( v \) inside \( s \)

inductive subst-stmt :: name ⇒ V ⇒ stmt ⇒ stmt ⇒ bool
where
  subst-stmt \( n \) \( v \) (Basic \( b \)) (Basic (subst-basic-stmt \( n \) \( v \) \( b \)))
  [[ subst-stmt \( n \) \( v \) \( S \)1 \( S \)1' ; subst-stmt \( n \) \( v \) \( S \)2 \( S \)2' ] ⇒
    subst-stmt \( n \) \( v \) \( S \)1 ; subst-stmt \( n \) \( v \) \( S \)2 ; subst-stmt \( n \) \( v \) \( S \)1' ; subst-stmt \( n \) \( v \) \( S \)2' ] ⇒
  subst-stmt \( n \) \( v \) (Local \( m \) \( S \)) (Local \( m \) \( S \)')
  [[ subst-stmt \( n \) \( v \) \( S \)1 \( S \)1' ; subst-stmt \( n \) \( v \) \( S \)2 \( S \)2' ] ⇒
    subst-stmt \( n \) \( v \) (If \( e \) \( S \)1 \( S \)2) (If \( e \) \( S \)1' \( S \)2')
  subst-stmt \( n \) \( v \) \( S \)1' ⇒ subst-stmt \( n \) \( v \) (While \( e \) \( S \)) (While \( e \) \( S \)')
  subst-stmt \( n \) \( v \) (Call \( f \) \( e \)) (Call \( f \) \( e \))
  subst-stmt \( n \) \( v \) Barrier Barrier
  subst-stmt \( n \) \( v \) Break Break
  subst-stmt \( n \) \( v \) Continue Continue
  subst-stmt \( n \) \( v \) Return Return

param-subst \( f \) \( u \) replaces \( f \)’s parameter with \( u \)

definition param-subst :: proc list ⇒ proc-name ⇒ V ⇒ stmt
where
  param-subst \( f s \) \( f u \) ≡
    let \( \text{proc} = \text{THE proc. proc} \in \text{set fs} \land \text{proc-name proc} = f \) in
      THE \( S \)' , subst-stmt (param proc) \( u \) (body proc) \( S \)'

Replace Return with \( v := \text{true} \)

fun relim :: stmt ⇒ V ⇒ stmt
where

\[
\text{relim (Basic } b \text{) } v = \text{Basic } b
\]

\[
\begin{align*}
\text{relim (S1 ; S2) } v &= (\text{relim S1 } v ; \text{ relim S2 } v) \\
\text{relim (Local } n \text{ S) } v &= \text{Local } n \text{ (relim S } v) \\
\text{relim (If } e \text{ S1 S2) } v &= \text{If } e \text{ (relim S1 } v) \text{ (relim S2 } v) \\
\text{relim (While } e \text{ S) } v &= \text{While } e \text{ (relim S } v)
\end{align*}
\]

\[
\begin{align*}
\text{relim (Call } f \text{ e) } v &= \text{Call } f \text{ e} \\
\text{relim Barrier } v &= \text{Barrier} \\
\text{relim Break } v &= \text{Break} \\
\text{relim Continue } v &= \text{Continue} \\
\text{relim Return } v &= \text{Basic (Assign (Var } v \text{) eTrue)}
\end{align*}
\]

Fresh variables

definition fresh :: \(V \Rightarrow V\) list \(\Rightarrow\) bool

where fresh \(v\) \(\in\) \(\equiv\) \(v \notin\) \(\set\) \(\text{vs}\)

The rules of Figure 6

inductive \(\text{step-k}\) :

\(\text{abs-level} \Rightarrow \text{proc list} \Rightarrow \text{threadset} \Rightarrow \text{kernel-state} \Rightarrow \text{kernel-state} \Rightarrow \text{option} \Rightarrow \text{bool}\)

where

\(\text{K-Inter-Group-Race}:\)

\[
\forall i \in G. \text{step-g a i } T \text{ (the (}\kappa \text{ i), (Basic b, p)) (Some (the (}\kappa' \text{ i))) ;} \quad \text{kernel-race P } \kappa' \text{ } \Rightarrow \text{step-k a fs (G, } T \text{ (}\kappa, \text{ (Basic b, p) } \# \text{ ss, } \text{vs) None}}
\]

\(\text{K-Intra-Group-Race}:\)

\[
\forall i \in G. \text{step-g a i } T \text{ (the (}\kappa \text{ i), (Basic s, p)) None } \Rightarrow \text{step-k a fs (G, } T \text{ (}\kappa, \text{ (Basic s, p) } \# \text{ ss, } \text{vs) None}}
\]

\(\text{K-Basic}:\)

\[
\forall i \in G. \text{step-g a i } T \text{ (the (}\kappa \text{ i), (Basic b, p)) (Some (the (}\kappa' \text{ i))) ;} \quad \neg \text{ (kernel-race G } \kappa' \text{ ) } \Rightarrow \text{step-k a fs (G, } T \text{ (}\kappa, \text{ (Basic b, p) } \# \text{ ss, } \text{vs) (Some (}\kappa' \text{,ss, vs))}}
\]

\(\text{K-Divergence}:\)

\[
\forall i \in G. \text{step-g a i } T \text{ (the (}\kappa \text{ i), (Barrier, p)) None } \Rightarrow \text{step-k a fs (G, } T \text{ (}\kappa, \text{ (Barrier, p) } \# \text{ ss, } \text{vs) None}}
\]

\(\text{K-Sync}:\)

\[
\forall i \in G. \text{step-g a i } T \text{ (the (}\kappa \text{ i), (Barrier, p)) (Some (the (}\kappa' \text{ i))) ;} \quad \neg \text{ (kernel-race G } \kappa' \text{ ) } \Rightarrow \text{step-k a fs (G, } T \text{ (}\kappa, \text{ (Barrier, p) } \# \text{ ss, } \text{vs) (Some (}\kappa' \text{,ss, vs))}}
\]

\(\text{K-Seq}:\)

\[
\text{step-k a fs (G, } T \text{ (}\kappa, \text{ (S1 ; } S2, \text{ p) } \# \text{ ss, } \text{vs) (Some (}\kappa, \text{ (S1, p) } \# \text{ (S2, p) } \# \text{ ss, vs))}}
\]

\(\text{K-Var}:\)

\[
\text{fresh } v \text{ vs } \Rightarrow \text{step-k a fs (G, } T \text{ (}\kappa, \text{ (Local } n \text{ S, p) } \# \text{ ss, } \text{vs) (Some (}\kappa, \text{ (THE } S'. \text{ subst-stmt n } v \text{ S } S', \text{ p) } \# \text{ ss, } v \# \text{ vs))}}
\]

\(\text{K-If}:\)

\[
\text{fresh } v \text{ vs } \Rightarrow
\]

13
step-k a fs \((G, T) (\kappa, (If \ e \ S1 \ S2, \ p) \# ss, vs) (Some (\kappa, (Basic (Assign (Var v) e), p)) \# (S1, p \&\& \ Loc (Var v)) \# (S2, p \&\& \neg \ Loc (Var v)) \# ss, v \# vs))\)

<table>
<thead>
<tr>
<th>K-Open:</th>
</tr>
</thead>
<tbody>
<tr>
<td>fresh v vs \implies \step-k a fs ((G, T) (\kappa, (While e S, p) # ss, vs) (Some (\kappa, (Basic (Assign (Var v) e) p)) # (\text{WhileDyn e S1}, p) # ss, v # vs)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K-Iter:</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
&\forall i \in G; j \in \text{the (T i)}; \\
&eavl-bool (p \&\& e) \text{ (the ((the (\kappa i))ts j))}; \\
&\text{fresh u vs ; fresh v vs; u \neq v } \implies \step-k a fs \((G, T) (\kappa, (\text{WhileDyn e S}, p) \# ss, vs) (Some (\kappa, (Basic (Assign (Var u) e) p)) \# (\text{WhileDyn e S1}, p) \# ss, u \# v \# vs))
\end{align*}
\] |

<table>
<thead>
<tr>
<th>K-Done:</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
&\forall i \in G. \forall j \in \text{the (T i)}. \\
&\neg (eavl-bool (p \&\& e) \text{ (the ((the (\kappa i))ts j))) } \implies \step-k a fs \((G, T) (\kappa, (\text{WhileDyn e S}, p) \# ss, vs) (Some (\kappa, ss, vs))
\end{align*}
\] |

<table>
<thead>
<tr>
<th>K-Call:</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
&\# \text{fresh u vs ; fresh v vs; u \neq v ; s = param-subst fs f u } \implies \step-k a fs \((G, T) (\kappa, (\text{Call f e}, p) \# ss, vs) \) (Some (\kappa, (Basic (Assign (Var u) e) ; relim s v, p \&\& \neg \ Loc (Var v)) \# ss, u \# v \# vs))
\end{align*}
\] |

end

theory Kernel-programming-language imports

Misc

KPL-syntax

KPL-wellformedness

KPL-state

KPL-execution-thread

KPL-execution-group

KPL-execution-kernel

begin

end
References