

Syntax and semantics of a GPU kernel programming language

John Wickerson

June 16, 2019

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	10

1 General purpose definitions and lemmas

theory *Misc* **imports**

Main

begin

A handy abbreviation when working with maps

abbreviation *make-map* :: 'a set \Rightarrow 'b \Rightarrow ('a \rightarrow 'b) ([- | \Rightarrow -])

where

[*ks* | \Rightarrow *v*] \equiv λk . if *k* \in *ks* then *Some v* else *None*

Projecting the components of a triple

definition $fst3 \equiv fst$

definition $snd3 \equiv fst \circ snd$

definition $thd3 \equiv snd \circ snd$

lemma $fst3\text{-simp}$ [*simp*]: $fst3 (a,b,c) = a$ $\langle proof \rangle$

lemma $snd3\text{-simp}$ [*simp*]: $snd3 (a,b,c) = b$ $\langle proof \rangle$

lemma $thd3\text{-simp}$ [*simp*]: $thd3 (a,b,c) = c$ $\langle proof \rangle$

end

2 Syntax of KPL

theory $KPL\text{-syntax}$ **imports**

$Misc$

begin

Locations of local variables

typedecl V

C strings

typedecl $name$

Procedure names

typedecl $proc\text{-}name$

Local-id, group-id

type-synonym $lid = nat$

type-synonym $gid = nat$

Fully-qualified thread-id

type-synonym $tid = gid \times lid$

Let (G, T) range over threadsets

type-synonym $threadset = gid\ set \times (gid \rightarrow lid\ set)$

Returns the set of tids in a threadset

fun $tids :: threadset \Rightarrow tid\ set$

where

$tids (G,T) = \{(i,j) \mid i\ j. i \in G \wedge j \in the (T\ i)\}$

type-synonym $word = nat$

datatype $loc =$

$Name\ name$

$| Var\ V$

Local expressions

```
datatype local-expr =  
  Loc loc  
| Gid  
| Lid  
| eTrue  
| eConj local-expr local-expr (infixl  $\wedge^*$  50)  
| eNot local-expr ( $\neg^*$ )
```

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. *wf-local-expr ns e* means that

- *e* does not mention any internal locations, and
- any name mentioned by *e* is in the set *ns*.

fun *wf-local-expr* :: *name set* \Rightarrow *local-expr* \Rightarrow *bool*

where

wf-local-expr ns (Loc (Var j)) = False
| *wf-local-expr ns (Loc (Name n)) = (n \in ns)*
| *wf-local-expr ns (e1 \wedge^* e2) =*
 (*wf-local-expr ns e1 \wedge wf-local-expr ns e2*)
| *wf-local-expr ns (\neg^* e) = wf-local-expr ns e*
| *wf-local-expr ns - = True*

Well-formed basic statements. *wf-basic-stmt ns b* means that

- *b* does not mention any internal locations, and
- any name mentioned by *b* is in the set *ns*.

fun *wf-basic-stmt* :: *name set* \Rightarrow *basic-stmt* \Rightarrow *bool*

where

wf-basic-stmt ns (Assign x e) = wf-local-expr ns e
| *wf-basic-stmt ns (Read x e) = wf-local-expr ns e*
| *wf-basic-stmt ns (Write e1 e2) =*
 (*wf-local-expr ns e1 \wedge wf-local-expr ns e2*)

Well-formed statements. *wf-stmt ns F S* means:

- *S* only calls procedures whose name is in *F*,
- *S* does not contain *WhileDyn*,
- *S* does not mention internal variables,
- *S* only mentions names in *ns*, and
- *S* does not declare the same name twice, e.g. *Local x (Local x foo)*.

fun *wf-stmt* :: *name set* \Rightarrow *proc-name set* \Rightarrow *stmt* \Rightarrow *bool*

where

wf-stmt ns F (Basic b) = wf-basic-stmt ns b
| *wf-stmt ns F (S1 ;; S2) = (wf-stmt ns F S1 \wedge wf-stmt ns F S2)*
| *wf-stmt ns F (Local n S) = (n \notin ns \wedge wf-stmt ($\{n\} \cup ns$) F S)*
| *wf-stmt ns F (If e S1 S2) =*
 (*wf-local-expr ns e \wedge wf-stmt ns F S1 \wedge wf-stmt ns F S2*)

```

| wf-stmt ns F (While e S) =
  (wf-local-expr ns e ∧ wf-stmt ns F S)
| wf-stmt ns F (WhileDyn -) = False
| wf-stmt ns F (Call f e) = (f ∈ F ∧ wf-local-expr ns e)
| wf-stmt - - - = True

```

no-return S holds if S does not contain a *Return* statement

```

fun no-return :: stmt ⇒ bool
where
  no-return (S1 ;; S2) = (no-return S1 ∧ no-return S2)
| no-return (Local n S) = no-return S
| no-return (If e S1 S2) = (no-return S1 ∧ no-return S2)
| no-return (While e S) = (no-return S)
| no-return Return = False
| no-return - = True

```

Well-formed kernel

definition *wf-kernel* :: kernel ⇒ bool

where

```

wf-kernel P ≡
let F = set (map proc-name (procs P)) in

```

— The main statement must not refer to *any* variable, except those it locally defines.

```

wf-stmt {} F (main P)

```

— The main statement contains no return statement.

```

∧ no-return (main P)

```

— A procedure body may refer only to its argument.

```

∧ list-all (λf. wf-stmt {param f} F (body f)) (procs P)

```

end

4 Thread, group and kernel states

theory *KPL-state* **imports**

```

  KPL-syntax

```

begin

Thread state

```

record thread-state =

```

```

  l :: V + bool ⇒ word

```

```

  sh :: nat ⇒ word

```

```

  R :: nat set

```

```

  W :: nat set

```

abbreviation $GID \equiv \text{Inr True}$

abbreviation $LID \equiv \text{Inr False}$

Group state

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

Valid group state

fun $\text{valid-group-state} :: (\text{gid} \rightarrow \text{lid set}) \Rightarrow \text{gid} \Rightarrow \text{group-state} \Rightarrow \text{bool}$
where

$\text{valid-group-state } T i \gamma = (
 \text{dom } (\gamma \text{ }_{ts}) = \text{the } (T i) \wedge
 (\forall j \in \text{the } (T i).
 l (\text{the } (\gamma \text{ }_{ts} j)) \text{ } GID = i \wedge
 l (\text{the } (\gamma \text{ }_{ts} j)) \text{ } LID = j))$

Predicated statements

type-synonym $\text{pred-stmt} = \text{stmt} \times \text{local-expr}$

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

Kernel state

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

Valid kernel state

fun $\text{valid-kernel-state} :: \text{threadset} \Rightarrow \text{kernel-state} \Rightarrow \text{bool}$

where

$\text{valid-kernel-state } (G, T) (\kappa, \text{ss}, -) = (
 \text{dom } \kappa = G \wedge
 (\forall i \in G. \text{valid-group-state } T i (\text{the } (\kappa i))))$

Valid initial kernel state

fun $\text{valid-initial-kernel-state} :: \text{stmt} \Rightarrow \text{threadset} \Rightarrow \text{kernel-state} \Rightarrow \text{bool}$

where

$\text{valid-initial-kernel-state } S (G, T) (\kappa, \text{ss}, \text{vs}) = (
 \text{valid-kernel-state } (G, T) (\kappa, \text{ss}, \text{vs}) \wedge
 (\text{ss} = [(S, \text{eTrue})]) \wedge
 (\forall i \in G. R\text{-group } (\text{the } (\kappa i)) = \{\}) \wedge W\text{-group } (\text{the } (\kappa i)) = \{\}) \wedge
 (\forall i \in G. \forall j \in \text{the } (T i). R (\text{the } ((\text{the } (\kappa i))_{ts} j)) = \{\}
 \wedge W (\text{the } ((\text{the } (\kappa i))_{ts} j)) = \{\}) \wedge
 (\forall i \in G. \forall j \in \text{the } (T i). \forall v :: V.
 l (\text{the } ((\text{the } (\kappa i))_{ts} j)) (\text{Inl } v) = 0) \wedge
 (\forall i \in G. \forall i' \in G. \forall j \in \text{the } (T i). \forall j' \in \text{the } (T i').
 sh (\text{the } ((\text{the } (\kappa i))_{ts} j)) =
 sh (\text{the } ((\text{the } (\kappa i'))_{ts} j'))) \wedge$

($vs = []$)

end

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* \Rightarrow *thread-state* \Rightarrow *word*

where

eval-word (*Loc* (*Var* v)) $\tau = l \tau$ (*Inl* v)

| *eval-word* *Lid* $\tau = l \tau$ *LID*

| *eval-word* *Gid* $\tau = l \tau$ *GID*

| *eval-word* *eTrue* $\tau = 1$

| *eval-word* ($e1 \wedge^* e2$) $\tau =$

(*eval-word* $e1 \tau * \text{eval-word } e2 \tau$)

| *eval-word* ($\neg^* e$) $\tau = (\text{if } \text{eval-word } e \tau = 0 \text{ then } 1 \text{ else } 0)$

Evaluate a local expression down to a boolean

fun *eval-bool* :: *local-expr* \Rightarrow *thread-state* \Rightarrow *bool*

where

eval-bool $e \tau = (\text{eval-word } e \tau \neq 0)$

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* \Rightarrow (*thread-state* \times *pred-basic-stmt*) \Rightarrow *thread-state* \Rightarrow *bool*

where

T-Disabled:

$\neg (\text{eval-bool } p \tau) \Longrightarrow \text{step-t } a (\tau, (b, p)) \tau$

| *T-Assign*:

$\llbracket \text{eval-bool } p \tau ; l' = (l \tau) (\text{Inl } v := \text{eval-word } e \tau) \rrbracket$
 $\Longrightarrow \text{step-t } a (\tau, (\text{Assign } (\text{Var } v) e, p)) (\tau (| l := l' |))$

| *T-Read*:

$\llbracket \text{eval-bool } p \tau ; l' = (l \tau) (\text{Inl } v := \text{sh } \tau (\text{eval-word } e \tau)) ;$
 $R' = R \tau \cup \{ \text{eval-word } e \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} \rrbracket$
 $\Longrightarrow \text{step-t } a (\tau, (\text{Read } (\text{Var } v) e, p)) (\tau (| l := l', R := R' |))$

| *T-Write*:

$\llbracket \text{eval-bool } p \tau ;$
 $\text{sh}' = (\text{sh } \tau) (\text{eval-word } e1 \tau := \text{eval-word } e2 \tau) ;$
 $W' = W \tau \cup \{ \text{eval-word } e1 \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} \rrbracket$

$\implies \text{step-}t \ a \ (\tau, (\text{Write } e1 \ e2, p)) \ (\tau \ (| \ sh := sh', W := W' |))$
| *T-Read-Adv*:
 $\llbracket \text{eval-bool } p \ \tau ; l' = (l \ \tau) \ (\text{Inl } v := \text{asterisk}) ;$
 $R' = R \ \tau \cup \{ \text{eval-word } e \ \tau \} \rrbracket$
 $\implies \text{step-}t \ \text{Adv-Abst} \ (\tau, (\text{Read } (\text{Var } v) \ e, p)) \ (\tau \ (| \ l := l', R := R' |))$
| *T-Write-Adv*:
 $\llbracket \text{eval-bool } p \ \tau ; W' = W \ \tau \cup \{ \text{eval-word } e1 \ \tau \} \rrbracket$
 $\implies \text{step-}t \ \text{Adv-Abst} \ (\tau, (\text{Write } e1 \ e2, p)) \ (\tau \ (| \ W := W' |))$

Rephrasing *T-Assign* to make it more usable

lemma *T-Assign-helper*:

$\llbracket \text{eval-bool } p \ \tau ; l' = (l \ \tau) \ (\text{Inl } v := \text{eval-word } e \ \tau) ; \tau' = \tau \ (| \ l := l' |) \rrbracket$
 $\implies \text{step-}t \ a \ (\tau, (\text{Assign } (\text{Var } v) \ e, p)) \ \tau'$
 $\langle \text{proof} \rangle$

Rephrasing *T-Read* to make it more usable

lemma *T-Read-helper*:

$\llbracket \text{eval-bool } p \ \tau ; l' = (l \ \tau) \ (\text{Inl } v := sh \ \tau \ (\text{eval-word } e \ \tau)) ;$
 $R' = R \ \tau \cup \{ \text{eval-word } e \ \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} ;$
 $\tau' = \tau \ (| \ l := l', R := R' |) \rrbracket$
 $\implies \text{step-}t \ a \ (\tau, (\text{Read } (\text{Var } v) \ e, p)) \ \tau'$
 $\langle \text{proof} \rangle$

Rephrasing *T-Write* to make it more usable

lemma *T-Write-helper*:

$\llbracket \text{eval-bool } p \ \tau ;$
 $sh' = (sh \ \tau) \ (\text{eval-word } e1 \ \tau := \text{eval-word } e2 \ \tau) ;$
 $W' = W \ \tau \cup \{ \text{eval-word } e1 \ \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} ;$
 $\tau' = \tau \ (| \ sh := sh', W := W' |) \rrbracket$
 $\implies \text{step-}t \ a \ (\tau, (\text{Write } e1 \ e2, p)) \ \tau'$
 $\langle \text{proof} \rangle$

end

6 Execution rules for groups

theory *KPL-execution-group* **imports**

KPL-execution-thread

begin

Intra-group race detection

definition *group-race*

$:: \text{lid set} \Rightarrow (\text{lid} \rightarrow \text{thread-state}) \Rightarrow \text{bool}$

where *group-race* $T \ \gamma \equiv$

$\exists j \in T. \exists k \in T. j \neq k \wedge$

$W \ (\text{the } (\gamma \ j)) \cap (R \ (\text{the } (\gamma \ k)) \cup W \ (\text{the } (\gamma \ k))) \neq \{ \}$

The constraints for the *merge* map

inductive *pre-merge*

$:: \text{lid set} \Rightarrow (\text{lid} \rightarrow \text{thread-state}) \Rightarrow \text{nat} \Rightarrow \text{word} \Rightarrow \text{bool}$

where

$\llbracket j \in T ; z \in W (\text{the } (\gamma j)) ; \text{dom } \gamma = T \rrbracket \Longrightarrow$
 $\text{pre-merge } T \gamma z (\text{sh } (\text{the } (\gamma j)) z)$
 $| \llbracket \forall j \in T. z \notin W (\text{the } (\gamma j)) ; \text{dom } \gamma = T \rrbracket \Longrightarrow$
 $\text{pre-merge } T \gamma z (\text{sh } (\text{the } (\gamma 0)) z)$

inductive-cases *pre-merge-inv* [*elim!*]: $\text{pre-merge } P \gamma z z'$

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 *merge* $:: \text{lid set} \Rightarrow (\text{lid} \rightarrow \text{thread-state}) \Rightarrow \text{nat} \Rightarrow \text{word}$
where $\text{merge } T \gamma \equiv \lambda z. \text{The } (\text{pre-merge } T \gamma z)$

lemma *no-races-imp-no-write-overlap*:

$\neg (\text{group-race } T \gamma) \Longrightarrow$
 $\forall i \in T. \forall j \in T.$
 $i \neq j \longrightarrow W (\text{the } (\gamma i)) \cap W (\text{the } (\gamma j)) = \{\}$
<proof>

lemma *merge-is-unique*:

assumes $\text{dom } \gamma = T$
assumes $\neg (\text{group-race } T \gamma)$
shows $\exists ! z'. \text{pre-merge } T \gamma z z'$
<proof>

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*

$:: \text{abs-level} \Rightarrow \text{gid} \Rightarrow (\text{gid} \rightarrow \text{lid set}) \Rightarrow (\text{group-state} \times \text{pred-stmt}) \Rightarrow \text{group-state option} \Rightarrow \text{bool}$

where

G-Race:

$\llbracket \forall j \in \text{the } (T i). \text{step-t } a (\text{the } (\gamma_{ts} j), (s, p)) (\text{the } (\gamma'_{ts} j)) ;$
 $\text{group-race } (\text{the } (T i)) ((\gamma' :: \text{group-state})_{ts}) \rrbracket$
 $\Longrightarrow \text{step-g } a i T (\gamma, (\text{Basic } s, p)) \text{None}$

| *G-Basic*:

$\llbracket \forall j \in \text{the } (T i). \text{step-t } a (\text{the } (\gamma_{ts} j), (s, p)) (\text{the } (\gamma'_{ts} j)) ;$
 $\neg (\text{group-race } (\text{the } (T i)) (\gamma'_{ts})) ;$
 $R\text{-group } \gamma' = R\text{-group } \gamma \cup (\bigcup j \in \text{the } (T i). (\{j\} \times R (\text{the } (\gamma'_{ts} j)))) ;$
 $W\text{-group } \gamma' = W\text{-group } \gamma \cup (\bigcup j \in \text{the } (T i). (\{j\} \times W (\text{the } (\gamma'_{ts} j)))) \rrbracket$
 $\Longrightarrow \text{step-g } a i T (\gamma, (\text{Basic } s, p)) (\text{Some } \gamma')$

| *G-No-Op*:

$\forall j \in \text{the } (T i). \neg (\text{eval-bool } p (\text{the } (\gamma_{ts} j)))$
 $\Longrightarrow \text{step-g } a i T (\gamma, (\text{Barrier}, p)) (\text{Some } \gamma)$

| *G-Divergence*:

$\llbracket j \neq k ; j \in \text{the } (T i) ; k \in \text{the } (T i) ;$

$$\llbracket \text{eval-bool } p \text{ (the } (\gamma \text{ }_{ts} j)) ; \neg (\text{eval-bool } p \text{ (the } (\gamma \text{ }_{ts} k))) \rrbracket$$

$$\implies \text{step-g a i T } (\gamma, (\text{Barrier}, p)) \text{ None}$$

| *G-Sync*:

$$\llbracket \forall j \in \text{the } (T \text{ } i). \text{eval-bool } p \text{ (the } (\gamma \text{ }_{ts} j)) ;$$

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

$$\text{sh} := \text{merge } P \text{ } (\gamma \text{ }_{ts}), R := \{\}, W := \{\} \text{ |)} \rrbracket$$

$$\implies \text{step-g No-Abst i T } (\gamma, (\text{Barrier}, p)) (\text{Some } \gamma')$$

| *G-Sync-Eq*:

$$\llbracket \forall j \in \text{the } (T \text{ } i). \text{eval-bool } p \text{ (the } (\gamma \text{ }_{ts} j)) ;$$

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

$$\text{sh} := \text{sh}', R := \{\}, W := \{\} \text{ |)} \rrbracket$$

$$\implies \text{step-g Eq-Abst i T } (\gamma, (\text{Barrier}, p)) (\text{Some } \gamma')$$

| *G-Sync-Adv*:

$$\llbracket \forall j \in \text{the } (T \text{ } i). \text{eval-bool } p \text{ (the } (\gamma \text{ }_{ts} j)) ;$$

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

$$\text{sh} := \text{sh}', R := \{\}, W := \{\} \text{ |)} \rrbracket$$

$$\implies \text{step-g Adv-Abst i T } (\gamma, (\text{Barrier}, p)) (\text{Some } \gamma')$$

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

lemma *G-No-Op-helper*:

$$\llbracket \forall j \in \text{the } (T \text{ } i). \neg (\text{eval-bool } p \text{ (the } (\gamma \text{ }_{ts} j))) ; \gamma = \gamma' \rrbracket$$

$$\implies \text{step-g a i T } (\gamma, (\text{Barrier}, p)) (\text{Some } \gamma')$$

<proof>

end

7 Execution rules for kernels

theory *KPL-execution-kernel* **imports**

KPL-execution-group

begin

Inter-group race detection

definition *kernel-race*

$:: \text{gid set} \Rightarrow (\text{gid} \rightarrow \text{group-state}) \Rightarrow \text{bool}$

where *kernel-race* $G \kappa \equiv$

$\exists i \in G. \exists j \in G. i \neq j \wedge$

$(\text{snd } ' (W\text{-group } (\text{the } (\kappa \text{ } i)))) \cap$

$(\text{snd } ' (R\text{-group } (\text{the } (\kappa \text{ } j)))) \cup \text{snd } ' (W\text{-group } (\text{the } (\kappa \text{ } j)))) \neq \{\}$

Replaces top-level *Break* with $v := \text{true}$

fun *belim* $:: \text{stmt} \Rightarrow V \Rightarrow \text{stmt}$

where

belim (*Basic* b) $v = \text{Basic } b$

| *belim* (*S1* ;; *S2*) $v = (\text{belim } S1 \text{ } v \text{ ;; } \text{belim } S2 \text{ } v)$

| *belim* (*Local* n *S*) $v = \text{Local } n \text{ } (\text{belim } S \text{ } v)$

| *belim* (*If* e *S1* *S2*) $v = \text{If } e \text{ } (\text{belim } S1 \text{ } v) \text{ } (\text{belim } S2 \text{ } v)$

| *belim* (*While e S*) *v* = *While e S*

| *belim* (*Call f e*) *v* = *Call f e*

| *belim* *Barrier v* = *Barrier*

| *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* \Rightarrow *V* \Rightarrow *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* *Barrier v* = *Barrier*

| *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* \Rightarrow *V* \Rightarrow *loc* \Rightarrow *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* \Rightarrow *V* \Rightarrow *local-expr* \Rightarrow *local-expr*

where

subst-local-expr n v (Loc loc) = *Loc (subst-loc n v loc)*

| *subst-local-expr n v Gid* = *Gid*

| *subst-local-expr n v Lid* = *Lid*

| *subst-local-expr n v eTrue* = *eTrue*

| *subst-local-expr n v (e1 \wedge * e2)* =

 (*subst-local-expr n v e1* \wedge * *subst-local-expr n v e2*)

| *subst-local-expr n v (\neg * e)* = \neg * (*subst-local-expr n v e*)

subst-basic-stmt n v b replaces *n* with *v* inside *b*

fun *subst-basic-stmt* :: *name* \Rightarrow *V* \Rightarrow *basic-stmt* \Rightarrow *basic-stmt*

where

subst-basic-stmt n v (Assign loc e) =

Assign (subst-loc n v loc) (subst-local-expr n v e)

| *subst-basic-stmt n v (Read loc e)* =

Read (subst-loc n v loc) (subst-local-expr n v e)

| $\text{subst-basic-stmt } n \ v \ (\text{Write } e1 \ e2) =$
 $\text{Write } (\text{subst-local-expr } n \ v \ e1) \ (\text{subst-local-expr } n \ v \ e2)$

$\text{subst-stmt } n \ v \ s \ t$ holds if t is the result of replacing n with v inside s

inductive $\text{subst-stmt} :: \text{name} \Rightarrow V \Rightarrow \text{stmt} \Rightarrow \text{stmt} \Rightarrow \text{bool}$
where

$\text{subst-stmt } n \ v \ (\text{Basic } b) \ (\text{Basic } (\text{subst-basic-stmt } n \ v \ b))$
| $\llbracket \text{subst-stmt } n \ v \ S1 \ S1' ; \text{subst-stmt } n \ v \ S2 \ S2' \rrbracket \Longrightarrow$
 $\text{subst-stmt } n \ v \ (S1 ;; S2) \ (S1' ;; S2')$
| $\llbracket m \neq n ; \text{subst-stmt } n \ v \ S \ S' \rrbracket \Longrightarrow$
 $\text{subst-stmt } n \ v \ (\text{Local } m \ S) \ (\text{Local } m \ S')$
| $\llbracket \text{subst-stmt } n \ v \ S1 \ S1' ; \text{subst-stmt } n \ v \ S2 \ S2' \rrbracket \Longrightarrow$
 $\text{subst-stmt } n \ v \ (\text{If } e \ S1 \ S2) \ (\text{If } e \ S1' \ S2')$
| $\text{subst-stmt } n \ v \ S \ S' \Longrightarrow \text{subst-stmt } n \ v \ (\text{While } e \ S) \ (\text{While } e \ S')$

| $\text{subst-stmt } n \ v \ (\text{Call } f \ e) \ (\text{Call } f \ e)$
| $\text{subst-stmt } n \ v \ \text{Barrier} \ \text{Barrier}$
| $\text{subst-stmt } n \ v \ \text{Break} \ \text{Break}$
| $\text{subst-stmt } n \ v \ \text{Continue} \ \text{Continue}$
| $\text{subst-stmt } n \ v \ \text{Return} \ \text{Return}$

$\text{param-subst } f \ u$ replaces f 's parameter with u

definition $\text{param-subst} :: \text{proc list} \Rightarrow \text{proc-name} \Rightarrow V \Rightarrow \text{stmt}$
where $\text{param-subst } fs \ f \ u \equiv$
 $\text{let } \text{proc} = \text{THE } \text{proc}. \text{proc} \in \text{set } fs \wedge \text{proc-name } \text{proc} = f \text{ in}$
 $\text{THE } S'. \text{subst-stmt } (\text{param } \text{proc}) \ u \ (\text{body } \text{proc}) \ S'$

Replace Return with $v := \text{true}$

fun $\text{relim} :: \text{stmt} \Rightarrow V \Rightarrow \text{stmt}$
where

$\text{relim } (\text{Basic } b) \ v = \text{Basic } b$
| $\text{relim } (S1 ;; S2) \ v = (\text{relim } S1 \ v ;; \text{relim } S2 \ v)$
| $\text{relim } (\text{Local } n \ S) \ v = \text{Local } n \ (\text{relim } S \ v)$
| $\text{relim } (\text{If } e \ S1 \ S2) \ v = \text{If } e \ (\text{relim } S1 \ v) \ (\text{relim } S2 \ v)$
| $\text{relim } (\text{While } e \ S) \ v = \text{While } e \ (\text{relim } S \ v)$
| $\text{relim } (\text{Call } f \ e) \ v = \text{Call } f \ e$
| $\text{relim } \text{Barrier} \ v = \text{Barrier}$
| $\text{relim } \text{Break} \ v = \text{Break}$
| $\text{relim } \text{Continue} \ v = \text{Continue}$
| $\text{relim } \text{Return} \ v = \text{Basic } (\text{Assign } (\text{Var } v) \ e\text{True})$

Fresh variables

definition $\text{fresh} :: V \Rightarrow V \text{ list} \Rightarrow \text{bool}$
where $\text{fresh } v \ vs \equiv v \notin \text{set } vs$

The rules of Figure 6

inductive *step-k*

$:: \text{abs-level} \Rightarrow \text{proc list} \Rightarrow \text{threadset} \Rightarrow \text{kernel-state} \Rightarrow \text{kernel-state option} \Rightarrow \text{bool}$

where

K-Inter-Group-Race:

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

| *K-Intra-Group-Race:*

$\llbracket i \in G; \text{step-g } a \ i \ T \ (\text{the } (\kappa \ i), (\text{Basic } s, p)) \ \text{None} \rrbracket \Longrightarrow$
 $\text{step-k } a \ fs \ (G, T) \ (\kappa, (\text{Basic } s, p) \# \ ss, \ vs) \ \text{None}$

| *K-Basic:*

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

| *K-Divergence:*

$\llbracket i \in G; \text{step-g } a \ i \ T \ (\text{the } (\kappa \ i), (\text{Barrier}, p)) \ \text{None} \rrbracket \Longrightarrow$
 $\text{step-k } a \ fs \ (G, T) \ (\kappa, (\text{Barrier}, p) \# \ ss, \ vs) \ \text{None}$

| *K-Sync:*

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

| *K-Seq:*

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

| *K-Var:*

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

| *K-If:*

$\text{fresh } v \ vs \Longrightarrow$
 $\text{step-k } a \ fs \ (G, T) \ (\kappa, (\text{If } e \ S1 \ S2, p) \# \ ss, \ vs) \ (\text{Some } (\kappa,$
 $(\text{Basic } (\text{Assign } (\text{Var } v) \ e), p)$
 $\# \ (S1, p \wedge^* \text{Loc } (\text{Var } v)))$
 $\# \ (S2, p \wedge^* \neg^* (\text{Loc } (\text{Var } v))) \# \ ss, \ v \ \# \ vs))$

| *K-Open:*

$\text{fresh } v \ vs \Longrightarrow$
 $\text{step-k } a \ fs \ (G, T) \ (\kappa, (\text{While } e \ S, p) \# \ ss, \ vs) \ (\text{Some } (\kappa,$
 $(\text{WhileDyn } e \ (\text{belim } S \ v), p \wedge^* \neg^* (\text{Loc } (\text{Var } v))) \# \ ss, \ v \ \# \ vs))$

| *K-Iter:*

$\llbracket i \in G \ ; \ j \in \text{the } (T \ i) \ ;$
 $\text{eval-bool } (p \wedge^* e) \ (\text{the } ((\text{the } (\kappa \ i))_{ts} \ j)) \ ;$
 $\text{fresh } u \ vs \ ; \ \text{fresh } v \ vs; \ u \neq v \rrbracket \Longrightarrow$
 $\text{step-k } a \ fs \ (G, T) \ (\kappa, (\text{WhileDyn } e \ S, p) \# \ ss, \ vs) \ (\text{Some } (\kappa,$
 $(\text{Basic } (\text{Assign } (\text{Var } u) \ e), p)$
 $\# \ (\text{celim } S \ v, p \wedge^* \text{Loc } (\text{Var } u) \wedge^* \neg^* (\text{Loc } (\text{Var } v)))$
 $\# \ (\text{WhileDyn } e \ S, p) \# \ ss, \ u \ \# \ v \ \# \ vs))$

| *K-Done:*

$\forall i \in G. \forall j \in \text{the } (T \ i).$
 $\neg (\text{eval-bool } (p \wedge^* e) \ (\text{the } ((\text{the } (\kappa \ i))_{ts} \ j))) \Longrightarrow$

$step\text{-}k\ a\ fs\ (G,T)\ (\kappa, (WhileDyn\ e\ S, p)\ \# ss, vs)\ (Some\ (\kappa, ss, vs))$
| *K-Call*:
 $\llbracket fresh\ u\ vs\ ;\ fresh\ v\ vs\ ;\ u\ \neq\ v\ ;\ s = param\text{-}subst\ fs\ f\ u \rrbracket \implies$
 $step\text{-}k\ a\ fs\ (G,T)\ (\kappa, (Call\ f\ e, p)\ \# ss, vs)$
 $(Some\ (\kappa, (Basic\ (Assign\ (Var\ u)\ e)\ ;\ relim\ s\ v,$
 $p\ \wedge^* \neg^* (Loc\ (Var\ v)))\ \# ss, u\ \# v\ \# vs))$

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

- [1] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels, 2014. Under submission.