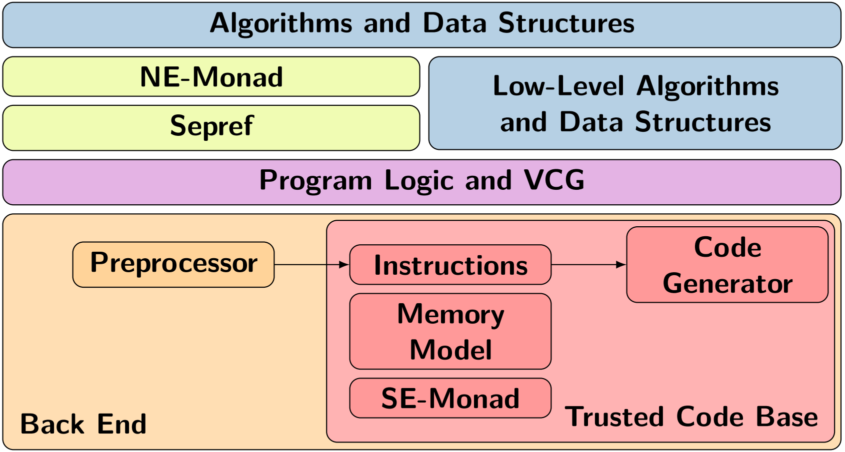
Refinement of Parallel Algorithms down to LLVM

Peter Lammich

University of Twente

Feb 2022

The Isabelle Refinement Framework



Isabelle LLVM Back End

Isabelle LLVM Back End

- Shallow embedding of small fragment of LLVM
  - just enough to express our programs
  - code generator translates to actual LLVM text

Isabelle LLVM Back End

- Shallow embedding of small fragment of LLVM
  - just enough to express our programs
  - code generator translates to actual LLVM text
- Simple memory model

```
datatype addr ≡ ADDR (bidx: nat) (idx: nat)
datatype ptr ≡ PTR_NULL | PTR_ADDR (the_addr: addr)
datatype val ≡ LL_INT lint | LL_STRUCT val list | LL_PTR ptr

datatype block ≡ FRESH | FREED | is_alloc: ALLOC (vals: val list)
typedef memory ≡ { μ :: nat ⇒ block. finite {b. μ b ≠ FRESH} }
```

Isabelle LLVM Back End

- Shallow embedding of small fragment of LLVM
  - just enough to express our programs
  - code generator translates to actual LLVM text
- Simple memory model

  datatype addr ≡ ADDR (bidx: nat) (idx: nat)
  datatype ptr ≡ PTR_NULL | PTR_ADDR (the_addr: addr)
  datatype val ≡ LL_INT lint | LL_STRUCT val list | LL_PTR ptr

  datatype block ≡ FRESH | FREED | is_alloc: ALLOC (vals: val list)
  typedef memory ≡ { μ :: nat ⇒ block. finite {b. μ b $\neq$ FRESH} }

- Using state-error monad

  α llM = memory ⇒ (FAIL | SUCC ($\alpha \times$ memory))

Shallow Embedding Example

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding Example

state/error monad

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding Example

state/error monad

types: word, pointer, struct

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding Example

state/error monad

types: word, pointer, struct

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

monad: bind, return

Shallow Embedding Example

state/error monad

types: word, pointer, struct

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

standard instructions (ll_<opcode>)

monad: bind, return

Shallow Embedding Example

state/error monad

types: word, pointer, struct

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

standard instructions (ll_<opcode>)

arguments: variables and constants

monad: bind, return

Shallow Embedding Example

state/error monad

types: word, pointer, struct

control flow (if, [optional: while])

standard instructions (ll_<opcode>)

arguments: variables and constants

monad: bind, return

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n1 ← ll_sub n 1;
      a  ← fib n1;
      n2 ← ll_sub n 2;
      b  ← fib n2;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding Example

state/error monad

types: word, pointer, struct

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

control flow (if, [optional: while])

standard instructions (ll_<opcode>)

function calls (rec. via fixp in ccpo)

arguments: variables and constants

monad: bind, return

Code Generation

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t

    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Code Generation

compiling control flow + pretty printing

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t

    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

```
define i64 @fib(i64 %n) {
  start:
    %t = icmp ule i64 %n, 1
    br i1 %t, label %then, label %else
  then:
    br label %ctd_if
  else:
    %n_1 = sub i64 %n, 1
    %a   = call i64 @fib (i64 %n_1)
    %n_2 = sub i64 %n, 2
    %b   = call i64 @fib (i64 %n_2)
    %c   = add i64 %a, %b
    br label %ctd_if
  ctd_if:
    %x1a = phi i64 [%n,%then], [%c,%else]
    ret i64 %x1a }
```

Preprocessor

- Only restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually
- Preprocessor transforms terms into restricted format
  - proves equality (via Isabelle kernel)
- Motto: Keep TCB small, preprocessor makes it usable

Example: Preprocessing Euclid's Algorithm

```
euclid :: 64 word ⇒ 64 word ⇒ 64 word
euclid a b = do {
 (a,b) ← llc_while
   (λ(a,b) ⇒ ll_cmp (a ≠ b))
   (λ(a,b) ⇒ if (a≤b) then return (a,b−a) else return (a−b,b))
   (a,b);
 return a }
```

Example: Preprocessing Euclid's Algorithm

```
euclid :: 64 word ⇒ 64 word ⇒ 64 word
euclid a b = do {
 (a,b) ← llc_while
   (λ(a,b) ⇒ ll_cmp (a ≠ b))
   (λ(a,b) ⇒ if (a≤b) then return (a,b−a) else return (a−b,b))
   (a,b);
 return a }
```

preprocessor defines function euclid_0 and proves

```
euclid a b = do {
   ab ← ll_insert₁ init a; ab ← ll_insert₂ ab b;
   ab ← euclid₀ ab;
   ll_extract₁ ab  }
euclid₀ s = do {
 a ← ll_extract₁ s;
 b ← ll_extract₂ s;
 ctd ← ll_icmp_ne a b;
 llc_if ctd do {...; euclid₀ ...} }
```

Reasoning about LLVM Programs

- Separation Logic

  α :: memory $\rightarrow$ amemory :: sep_algebra
  wp c Q s $\equiv \exists$r s$'$. c s = SUCC r s$' \wedge$ Q r (α s$'$)
  $\{$P$\}$ c $\{$Q$\} \equiv \forall$F s. (P $*$ F) (α s) $\longrightarrow$ wp c (λr s$'$. (Q r $*$ F) s$'$) s

  - defined wrt. shallowly embedded semantics
  - proof rules are proved theorems!

Reasoning about LLVM Programs

- Separation Logic

  α :: memory $\rightarrow$ amemory :: sep_algebra
  $\mathsf{wp}\ \mathsf{c}\ \mathsf{Q}\ \mathsf{s} \equiv \exists \mathsf{r}\ \mathsf{s}'.\ \mathsf{c}\ \mathsf{s} = \mathsf{SUCC}\ \mathsf{r}\ \mathsf{s}' \wedge \mathsf{Q}\ \mathsf{r}\ (\alpha\ \mathsf{s}')$
  $\{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\} \equiv \forall \mathsf{F}\ \mathsf{s}.\ (\mathsf{P} * \mathsf{F})\ (\alpha\ \mathsf{s}) \longrightarrow \mathsf{wp}\ \mathsf{c}\ (\lambda \mathsf{r}\ \mathsf{s}'.\ (\mathsf{Q}\ \mathsf{r} * \mathsf{F})\ \mathsf{s}')\ \mathsf{s}$

  - defined wrt. shallowly embedded semantics
  - proof rules are proved theorems!
- Automation: VCG, frame inference, heuristics to discharge VCs
  - these prove theorems!

Reasoning about LLVM Programs

- Separation Logic

  α :: memory $\rightarrow$ amemory :: sep_algebra
  wp c Q s $\equiv \exists$r s$'$. c s = SUCC r s$'$ $\wedge$ Q r (α s$'$)
  {P} c {Q} $\equiv \forall$F s. (P $*$ F) (α s) $\longrightarrow$ wp c (λr s$'$. (Q r $*$ F) s$'$) s

  - defined wrt. shallowly embedded semantics
  - proof rules are proved theorems!
- Automation: VCG, frame inference, heuristics to discharge VCs
  - these prove theorems!
- Basic Data Structures: signed/unsigned integers, Booleans, arrays

Sepref

- Semi-automatic translation of functional to imperative program
- Data refinement to imperative DS
    - e.g. list to array
- Proves refinement theorem

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← while (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<|xs| ∧ h≤|xs| ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,|xs|);
  return l
}
```

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← while (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<|xs| ∧ h≤|xs| ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,|xs|);
  return l
}
```

invariant annotation

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← while (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<|xs| ∧ h≤|xs| ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,|xs|);
  return l
}
```

invariant annotation

hint for subsequent refinement

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← while (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<|xs| ∧ h≤|xs| ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,|xs|);
  return l
}
```

invariant annotation

overflow-safe midpoint computation

hint for subsequent refinement

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← while (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<|xs| ∧ h≤|xs| ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,|xs|);
  return l
}
```

invariant annotation

overflow-safe midpoint computation

hint for subsequent refinement

```
lemma bin_search_correct:
  sorted xs ⟹ bin_search xs x ≤ spec i. i=find_index (≤y) xs
```

Example: Binary Search — Refinement

sepref_def $\text{bin\_search}_\dagger$ **is** bin_search
$:: (\text{array}_A\ \text{int}_A^{64})^k * (\text{int}_A^{64})^k \rightarrow \text{int}_A^{64}$
unfolding bin_search_def
apply (rule hfref_with_rdomI, annot_snat_const 64)
by sepref

Example: Binary Search — Refinement

sepref_def bin_search$_\dagger$ **is** bin_search
  $:: (\text{array}_A\ \text{int}_A^{64})^k * (\text{int}_A^{64})^k \rightarrow \text{int}_A^{64}$
  unfolding bin_search_def
  apply (rule hfref_with_rdomI, annot_snat_const 64)
  by sepref

hints for data refinement

Example: Binary Search — Refinement

sepref_def bin_search$_\dagger$ **is** bin_search
:: $(\text{array}_A\ \text{int}_A^{64})^k * (\text{int}_A^{64})^k \rightarrow \text{int}_A^{64}$
unfolding bin_search_def
apply (rule hfref_with_rdomI, annot_snat_const 64)
by sepref

hints for data refinement

automatic synthesis + proof

Example: Binary Search — Refinement

```
sepref_def bin_search† is bin_search
  :: (array_A int_A^64)^k * (int_A^64)^k → int_A^64
  unfolding bin_search_def
  apply (rule hfref_with_rdomI, annot_snat_const 64)
  by sepref
```

proves: $(\text{bin\_search}_\dagger, \text{bin\_search}) \in (\text{array}_A\ \text{int}_A^{64})^k * {\text{int}_A^{64}}^k \rightarrow \text{int}_A^{64}$

Example: Binary Search — Refinement

```
sepref_def bin_search† is bin_search
  :: (array_A int_A^64)^k * (int_A^64)^k → int_A^64
  unfolding bin_search_def
  apply (rule hfref_with_rdomI, annot_snat_const 64)
  by sepref
```

proves: $(\mathrm{bin\_search}_\dagger, \mathrm{bin\_search}) \in (\mathrm{array}_A\ \mathrm{int}_A^{64})^k * {\mathrm{int}_A^{64}}^k \to \mathrm{int}_A^{64}$
Combination with bin_search_correct yields:

theorem $\mathrm{bin\_search}_\dagger\mathrm{\_correct}$:

$$\{(\mathrm{array}_A\ \mathrm{int}_A^{64}\ xs\ xs_\dagger * \mathrm{int}_A^{64}\ x\ x_\dagger * \mathrm{sorted}\ xs)\}$$
$$(\mathrm{bin\_search}_\dagger\ xs_\dagger\ x_\dagger)$$
$$\{\lambda i_\dagger.\ \exists i.\ \mathrm{array}_A\ \mathrm{int}_A^{64}\ xs\ xs_\dagger * \mathrm{int}_A^{64}\ x\ x_\dagger * \mathrm{int}_A^{64}\ i\ i_\dagger * i{=}\mathrm{find\_index}\ ({\le}y)\ xs\}$$

Example: Binary Search — Generated Code

```
export_llvm bin_search† is int64_t bin_search(larray_t, elem_t)
defines
 typedef int64_t elem_t;
 typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Example: Binary Search — Generated Code

```
export_llvm bin_search† is int64_t bin_search(larray_t, elem_t)
defines
  typedef int64_t elem_t;
  typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Isabelle constant

Example: Binary Search — Generated Code

```
export_llvm bin_search† is int64_t bin_search(larray_t, elem_t)
defines
  typedef int64_t elem_t;
  typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Isabelle constant

LLVM name and signature

Example: Binary Search — Generated Code

```
export_llvm bin_search† is int64_t bin_search(larray_t, elem_t)
defines
  typedef int64_t elem_t;
  typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Isabelle constant

LLVM name and signature

C-style typedefs

Example: Binary Search — Generated Code

```
export_llvm bin_search† is int64_t bin_search(larray_t, elem_t)
defines
  typedef int64_t elem_t;
  typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Isabelle constant

LLVM name and signature

C-style typedefs

target file

Example: Binary Search — Generated Code

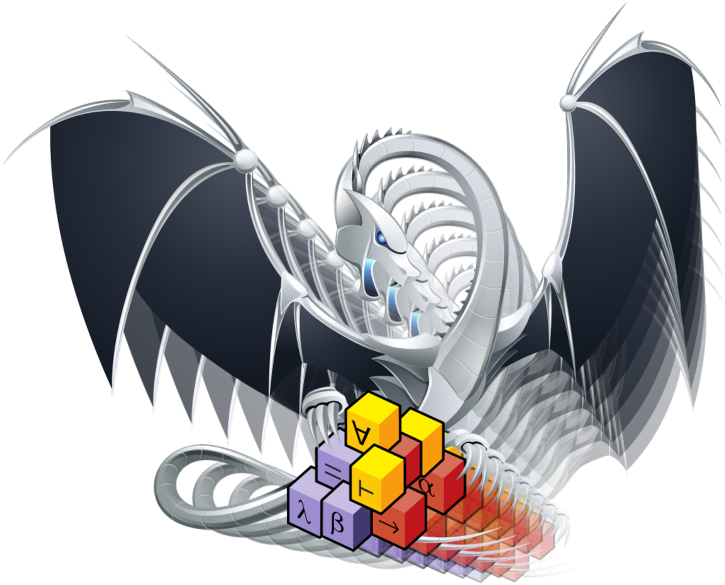
```
export_llvm bin_search† is int64_t bin_search(larray_t, elem_t)
defines
  typedef int64_t elem_t;
  typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Produces LLVM code and header file:

```
typedef int64_t elem_t;
typedef struct {
  int64_t len;
  elem_t*data;
} larray_t;

int64_t bin_search(larray_t,elem_t);
```

Isabelle-LLVM Parallel



Isabelle-LLVM Parallel

- Add parallel operator: $c_1 \parallel c_2$

Isabelle-LLVM Parallel

- Add parallel operator: $c_1 \parallel c_2$
  - execute in parallel, fail on data race

Isabelle-LLVM Parallel

- Add parallel operator: $c_1 \parallel c_2$
  - execute in parallel, fail on data race
- Shallow embedding: make program report memory accesses

$$
\begin{tikzcd}
 & \mu \arrow[dl, "c_1"'] \arrow[dr, "c_2"] & \\
\rho_1, \mu_1 \arrow[dr] & \substack{\text{check race} \\ \text{combine}} & \rho_2, \mu_2 \arrow[dl] \\
 & \mu' &
\end{tikzcd}
$$

Isabelle-LLVM Parallel

- Add parallel operator: $c_1 \parallel c_2$
  - execute in parallel, fail on data race
- Shallow embedding: make program report memory accesses

$$
\begin{array}{ll}
(\mathsf{c}_1 \parallel \mathsf{c}_2)\ \mu \equiv & \\
\quad (\mathsf{r}_1,\rho_1,\mu_1) \leftarrow \mathsf{c}_1\ \mu & \text{— execute first strand} \\
\quad (\mathsf{r}_2,\rho_2,\mu_2) \leftarrow \mathsf{c}_2\ \mu & \text{— execute second strand} \\
\quad \texttt{assert}\ \mathsf{no\_race}\ \rho_1\ \rho_2 & \text{— fail on data race} \\
\quad \mu' = \mathsf{combine}\ \rho_1\ \mu_1\quad \rho_2\ \mu_2 & \text{— combine states} \\
\quad \texttt{return}\ ((\mathsf{r}_1,\mathsf{r}_2),\ \rho_1 \cup \rho_2,\ \mu') &
\end{array}
$$

Isabelle-LLVM Parallel

- Add parallel operator: $c_1 \parallel c_2$
  - execute in parallel, fail on data race
- Shallow embedding: make program report memory accesses

$(\mathsf{c}_1 \parallel \mathsf{c}_2)\ \mu \equiv$

Access report

$(\mathsf{r}_1,\rho_1,\mu_1) \leftarrow \mathsf{c}_1\ \mu$ — execute first strand

$(\mathsf{r}_2,\rho_2,\mu_2) \leftarrow \mathsf{c}_2\ \mu$ — execute second strand

$\texttt{assert}\ \mathsf{no\_race}\ \rho_1\ \rho_2$ — fail on data race

$\mu' = \mathsf{combine}\ \rho_1\ \mu_1\quad \rho_2\ \mu_2$ — combine states

$\texttt{return}\ ((\mathsf{r}_1,\mathsf{r}_2),\ \rho_1 \cup \rho_2,\ \mu')$

Memory Allocation

- Currently: deterministic semantics
  - $c_1\ \mu$ and $c_2\ \mu$ will allocate same memory
  - cannot be combined!

Memory Allocation

- Currently: deterministic semantics
  - $c_1\ \mu$ and $c_2\ \mu$ will allocate same memory
  - cannot be combined!
- Use nondeterminism

$$\alpha\ \mathsf{llM} = \mathsf{memory} \Rightarrow \mathsf{FAIL} \mid \mathsf{SUCC}\ ((\alpha \times \mathsf{report} \times \mathsf{memory})\ \texttt{set})$$

  - malloc nondeterministically allocates some free address
  - on combination: exclude infeasible possibilities

Memory Allocation

- Currently: deterministic semantics
  - $c_1\ \mu$ and $c_2\ \mu$ will allocate same memory
  - cannot be combined!
- Use nondeterminism

$$\alpha\ \mathsf{llM} = \mathsf{memory} \Rightarrow \mathsf{FAIL} \mid \mathsf{SUCC}\ ((\alpha \times \mathsf{report} \times \mathsf{memory})\ \mathtt{set})$$

  - malloc nondeterministically allocates some free address
  - on combination: exclude infeasible possibilities

$(\mathsf{c}_1 \mid\mid \mathsf{c}_2)\ \mu \equiv$
$\quad(\mathsf{r}_1,\rho_1,\mu_1) \leftarrow \mathsf{c}_1\ \mu$
$\quad(\mathsf{r}_2,\rho_2,\mu_2) \leftarrow \mathsf{c}_2\ \mu$
$\quad\mathtt{assume}\ \rho_1.\mathsf{alloc} \cap \rho_2.\mathsf{alloc} = \emptyset$ — ignore infeasible combinations
$\quad\mathtt{assert}\ \mathsf{no\_race}\ \rho_1\ \rho_2$ — fail on data race
$\quad\mu' = \mathsf{combine}\ \rho_1\ \mu_1\quad \rho_2\ \mu_2$ — combine states
$\quad\mathtt{return}\ ((\mathsf{r}_1,\mathsf{r}_2),\ \rho_1 \cup \rho_2,\ \mu')$

Invariants

- We prove for IIM (enforced by subtype)
  - access reports are consistent with observed changes in memory
  - there is at least one possible result (no magic happens)
- Sanity check for semantics
- Allows us to prove symmetry of $\|$

$$c_1 \mid\mid c_2 \;=\; \text{swapres}\ (c_2 \mid\mid c_1)$$

$$\text{swapres}\ m \equiv (r_1,r_2) \leftarrow m;\ \texttt{return}\ (r_2,r_1)$$

Code generator

- We add $\text{llc\_par}\ f_1\ f_2\ x_1\ x_2 \equiv f_1\ x_1 \parallel f_2\ x_2$
  - f_1, f_2 must be functions

Code generator

- We add $\text{llc\_par}\ f_1\ f_2\ x_1\ x_2 \equiv f_1\ x_1 \parallel f_2\ x_2$
    - f_1, f_2 must be functions
- Code generator generates
    - type casting boilerplate
    - call to external parallel function

```
void parallel(void (*f1)(void*), void (*f2)(void*), void *x1, void *x2)
```

Code generator

- We add $\text{llc\_par } f_1\ f_2\ x_1\ x_2 \equiv f_1\ x_1 \parallel f_2\ x_2$
  - f_1, f_2 must be functions
- Code generator generates
  - type casting boilerplate
  - call to external parallel function

```
void parallel(void (*f1)(void*), void (*f2)(void*), void *x1, void *x2)
```

- For example, implemented using TBB:

```
{
  tbb::parallel_invoke([=]{f1(x1);}, [=]{f2(x2);});
}
```

Amending higher layers of IRF

- Prove concurrency rule

$$\begin{aligned} & \{\mathsf{P}_1\}\ \mathsf{c}_1\ \{\mathsf{Q}_1\} \quad \wedge \quad \{\mathsf{P}_2\}\ \mathsf{c}_2\ \{\mathsf{Q}_2\} \\ \implies & \{\mathsf{P}_1 * \mathsf{P}_2\}\ \mathsf{c}_1 \parallel \mathsf{c}_2\ \{\lambda(\mathsf{r}_1,\mathsf{r}_2).\ \mathsf{Q}_1\ \mathsf{r}_1 * \mathsf{Q}_2\ \mathsf{r}_2\} \end{aligned}$$

Amending higher layers of IRF

- Prove concurrency rule

$$\begin{aligned} & \{P_1\}\ c_1\ \{Q_1\} \quad \wedge \quad \{P_2\}\ c_2\ \{Q_2\} \\ \Longrightarrow\ & \{P_1 * P_2\}\ c_1 \parallel c_2\ \{\lambda(r_1,r_2).\ Q_1\ r_1 * Q_2\ r_2\} \end{aligned}$$

- Sepref refines sequential to parallel execution

  $\mathsf{npar}\ f_1\ f_2\ x_1\ x_2 \equiv r_1 \leftarrow f_1\ x_1;\ r_2 \leftarrow f_2\ x_2;\ \texttt{return}\ (r_1,r_2)$

  refined to llc_par.

Amending higher layers of IRF

- Prove concurrency rule

$$\{P_1\}\ c_1\ \{Q_1\} \quad \wedge \quad \{P_2\}\ c_2\ \{Q_2\}$$
$$\Longrightarrow\ \{P_1 * P_2\}\ c_1 \parallel c_2\ \{\lambda(r_1,r_2).\ Q_1\ r_1 * Q_2\ r_2\}$$

- Sepref refines sequential to parallel execution

  $\mathsf{npar}\ f_1\ f_2\ x_1\ x_2 \equiv r_1 \leftarrow f_1\ x_1;\ r_2 \leftarrow f_2\ x_2;\ \texttt{return}\ (r_1,r_2)$

  refined to llc_par.
- Backwards compatible with sequential Sepref!
  - Easy porting of existing algorithms

Parallel Quicksort (basic)

```
psort xs ≡
  if |xs|≤1 then return xs                          — trivially sorted
  else
    (xs,m) ← partition_spec xs;                      — partition
    (_,xs) ← with_split m xs (λxs₁ xs₂.
      npar psort psort xs₁ xs₂                       — recursively sort partitions
    );
    return xs

with_split i xs f ≡
  assert (i < |xs|);                                 — split point must be in list
  (xs₁,xs₂) ← f (take i xs) (drop i xs);             — execute f with halfs
  assert (|xs₁| = i ∧ |xs₂| = |xs| − i);             — length of halfs must not change
  return (xs₁@xs₂)                                   — return both halfs
```

Parallel Quicksort (refined)

```
psort xs n ≡
 assert n=|xs|;
 if n≤1 then return xs
 else psort_aux xs n (log2 n * 2)                    — recursion depth limit

psort_aux xs n d ≡
 assert n=|xs|                                       — extra parameter for length
 if d=0 ∨ n<100000 then sort_spec xs                 — fallback to seq-sort
 else
   (xs,m) ← partition_spec xs;
   let bad = m<n div 8 ∨ (n−m < n div 8)             — check unbalanced partition
   (_,xs) ← with_split m xs (λxs₁ xs₂.
     if bad then                                     — sequentially recurse for unbalanced
       nseq psort_aux psort_aux (xs₁,m,d−1) (xs₂,n−m,d−1)
     else                                            — recurse in parallel for balanced
       npar psort_aux psort_aux (xs₁,m,d−1) (xs₂,n−m,d−1)
   );
   return xs
```

Parallel Quicksort (Sepref + Code Export)

- Sepref generates imperative program
  - using existing sequential pdqsort for fallback
  - using (new) sampling partitioner (proved correct + refined separately)

Parallel Quicksort (Sepref + Code Export)

- Sepref generates imperative program
  - using existing sequential pdqsort for fallback
  - using (new) sampling partitioner (proved correct + refined separately)
- Correctness theorem:

  $\{\mathsf{arr}_A\ \mathsf{xs}\ \mathsf{xs}_\dagger * \mathsf{idx}_A\ \mathsf{n}\ \mathsf{n}_\dagger * \mathsf{n} = |\mathsf{xs}|\}$
  $(\mathsf{psort}_\dagger\ \mathsf{xs}_\dagger\ \mathsf{n}_\dagger)$
  $\{\lambda \mathsf{r}.\ \mathsf{r}{=}\mathsf{xs}_\dagger * \exists\ \mathsf{xs}'.\ \mathsf{arr}_A\ \mathsf{xs}'\ \mathsf{xs}_\dagger * \mathsf{sorted}\ \mathsf{xs}' * \mathsf{mset}\ \mathsf{xs}' = \mathsf{mset}\ \mathsf{xs}\}$

Parallel Quicksort (Sepref + Code Export)

- Sepref generates imperative program
  - using existing sequential pdqsort for fallback
  - using (new) sampling partitioner (proved correct + refined separately)
- Correctness theorem:

$\{\mathrm{arr}_A\ xs\ xs_\dagger * \mathrm{idx}_A\ n\ n_\dagger * n = |xs|\}$
$(\mathrm{psort}_\dagger\ xs_\dagger\ n_\dagger)$
$\{\lambda r.\ r{=}xs_\dagger * \exists\, xs'.\ \mathrm{arr}_A\ xs'\ xs_\dagger * \mathrm{sorted}\ xs' * \mathrm{mset}\ xs' = \mathrm{mset}\ xs\}$

- Instantiation to concrete weak ordering + code export

interpretation unat: pcmp (λ_. <) (λ_. ll_icmp_ult) unat_A^{64} ⟨proof⟩
interpretation str: pcmp (λ_. <) (λ_. strcmp) str_A^{64} ⟨proof⟩

```
export_llvm
  unat.psort† is uint64_t* psort(uint64_t*, int64_t)
  str.psort† is llstring* str_psort(llstring*, int64_t)
  defines
    typedef struct {int64_t sz; struct {int64_t cap; char *data;};} llstring;
  file psort.ll
```

Parallel Quicksort (Sepref + Code Export)

- Sepref generates imperative program
  - using existing sequential pdqsort for fallback
  - using (new) sampling partitioner (proved correct + refined separately)
- Correctness theorem:

  $\{\mathrm{arr}_A\ \mathrm{xs}\ \mathrm{xs}_\dagger * \mathrm{idx}_A\ \mathrm{n}\ \mathrm{n}_\dagger * \mathrm{n} = |\mathrm{xs}|\}$
  $(\mathrm{psort}_\dagger\ \mathrm{xs}_\dagger\ \mathrm{n}_\dagger)$
  $\{\lambda r.\ r{=}\mathrm{xs}_\dagger * \exists\ \mathrm{xs}'.\ \mathrm{arr}_A\ \mathrm{xs}'\ \mathrm{xs}_\dagger * \mathrm{sorted}\ \mathrm{xs}' * \mathrm{mset}\ \mathrm{xs}' = \mathrm{mset}\ \mathrm{xs}\}$

- Instantiation to concrete weak ordering + code export

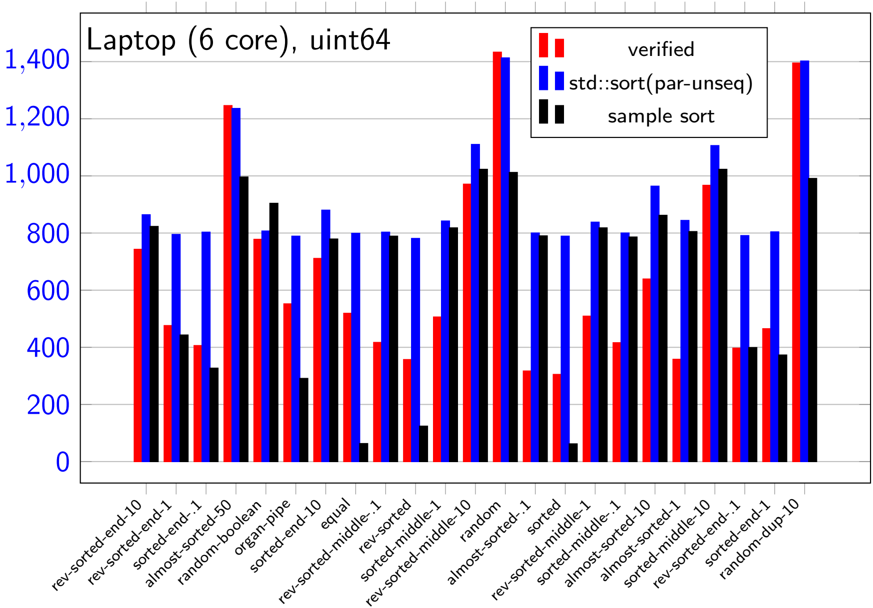
  interpretation unat: pcmp (λ_. $<$) (λ_. ll_icmp_ult) unat_A^{64} ⟨proof⟩
  interpretation str: pcmp (λ_. $<$) (λ_. strcmp) str_A^{64} ⟨proof⟩

  export_llvm
   $\mathrm{unat.psort}_\dagger$ **is** uint64_t* psort(uint64_t*, int64_t)
   $\mathrm{str.psort}_\dagger$ **is** llstring* str_psort(llstring*, int64_t)
   defines
    typedef struct {int64_t sz; struct {int64_t cap; char *data;};} llstring;
   file psort.ll

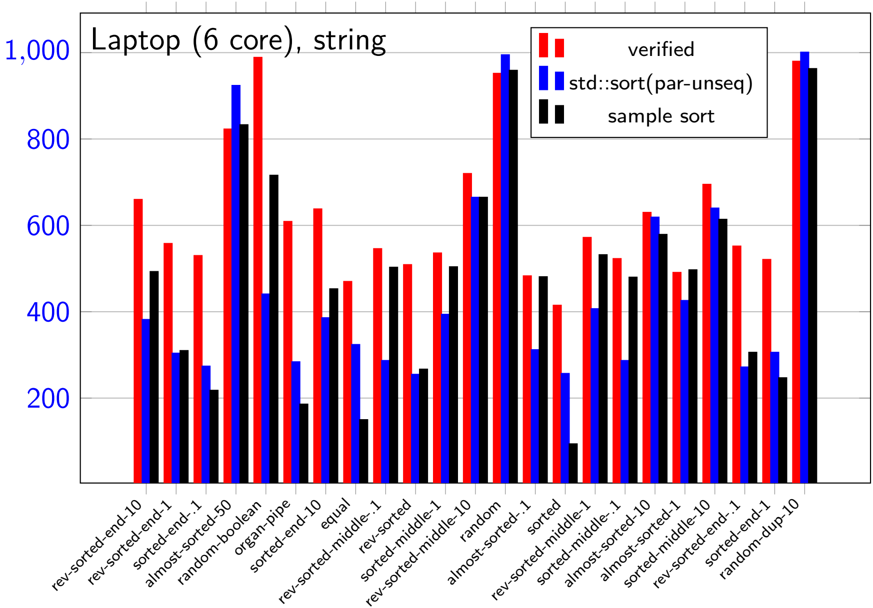
- Link against C++ benchmark driver

  clang++ [. . .] lib_isabelle_llvm.cpp psort.ll benchmark.cpp

Benchmarks

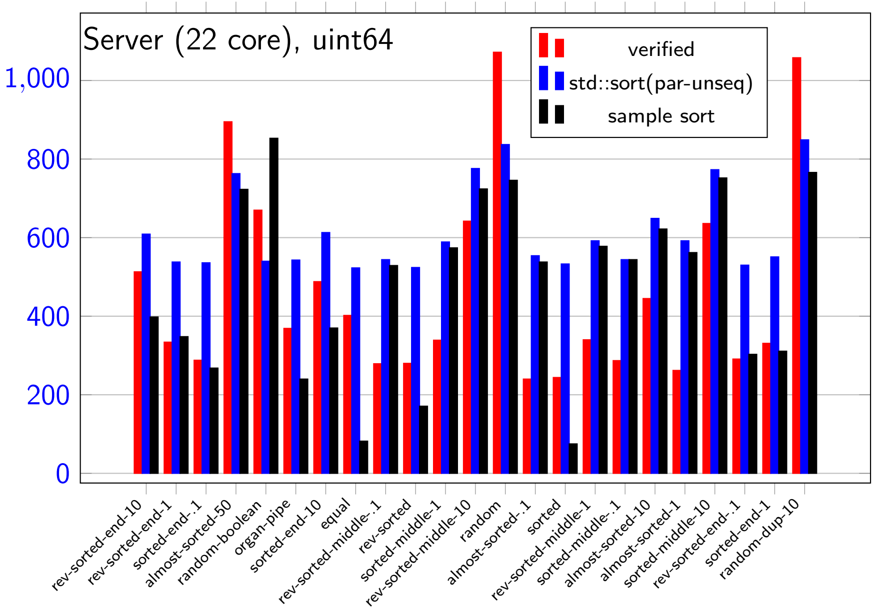

Laptop (6 core), uint64
verified
std::sort(par-unseq)
sample sort
0
200
400
600
800
1,000
1,200
1,400
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

Benchmarks



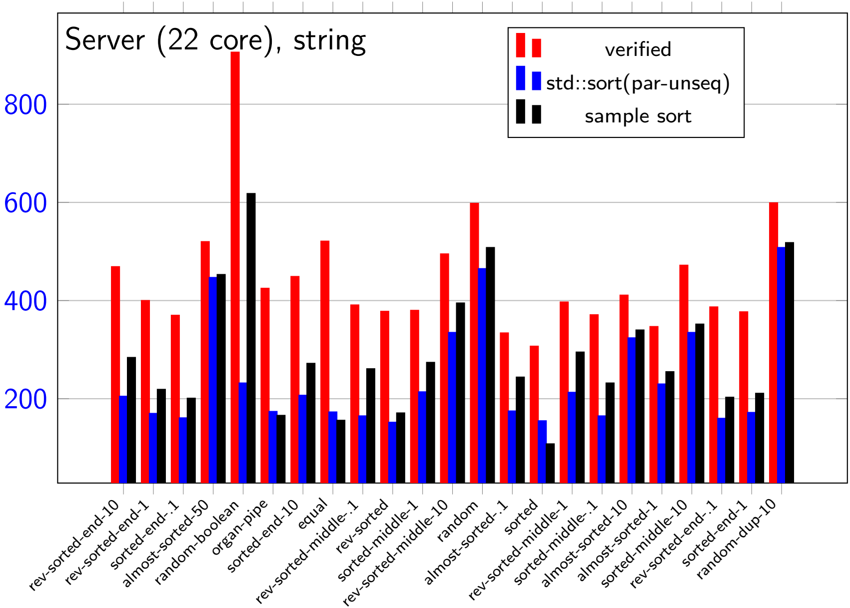
Laptop (6 core), string
verified
std::sort(par-unseq)
sample sort
1,000
800
600
400
200
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

Benchmarks



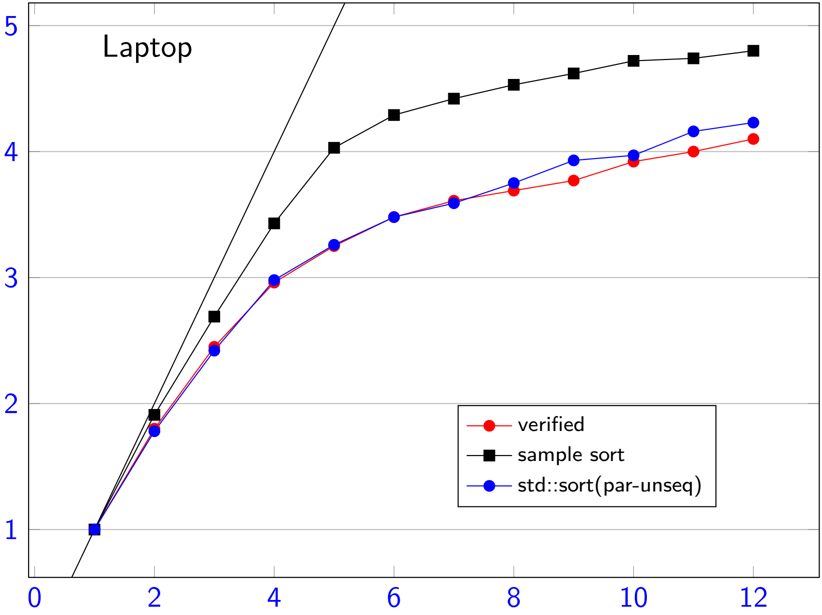
Server (22 core), uint64
verified
std::sort(par-unseq)
sample sort
0
200
400
600
800
1,000
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

Benchmarks



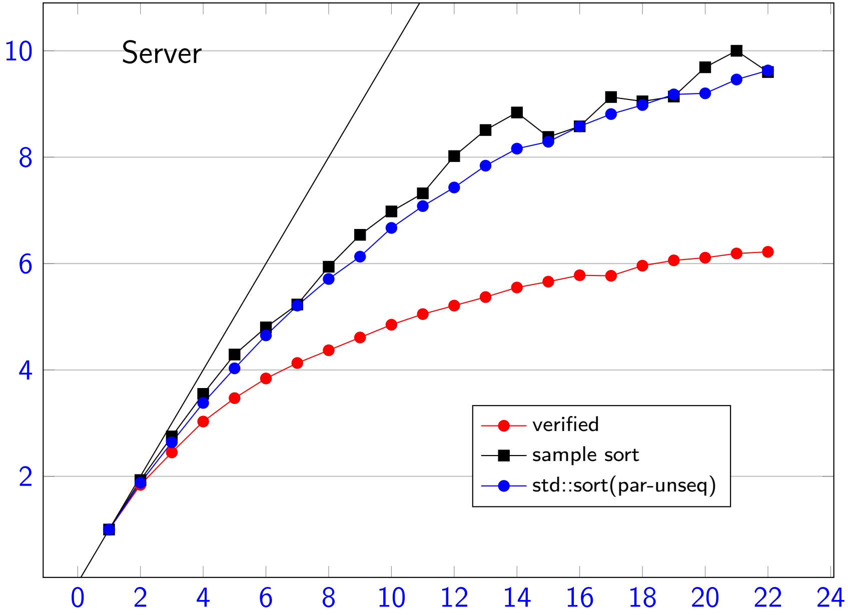
Server (22 core), string
verified
std::sort(par-unseq)
sample sort
800
600
400
200
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

Speedup



Laptop
5
4
3
2
1
0
2
4
6
8
10
12
verified
sample sort
std::sort(par-unseq)

Speedup


Server
verified
sample sort
std::sort(par-unseq)
0
2
4
6
8
10
12
14
16
18
20
22
24
2
4
6
8
10

Conclusion

- Verification of parallel programs
  - stepwise refinement to tackle complexity
  - down to LLVM, small TCB
  - fast verified programs
- Idea: shallow embedding, using access reports
  - backwards compatible with sequential IRF
- Future work
  - state-of-the-art parallel sorting
  - fractional separation logic
  - more concurrency
  - complexity of parallel algorithms
  - GP-GPUs