A linear functional first-order intermediate language for verified compilers

Sigurd Schneider, Sebastian Hack, Gert Smolka

ITP 2015, Nanjing

2015-08-24

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Introduction

Binding vs. assignment

Binding	Assignment

Introduction

Binding vs. assignment

Binding	Assignment

let x=e in s

- *x* is bound in term *s*
- *functional*

Introduction

Binding vs. assignment

Binding	Assignment
let x=e in s	x := e; s

Binding

let x=e in s

- x is bound in term s
- *functional*

Assignment

x := e; s

- x is a register
- *imperative*

Introduction

Binding vs. assignment

Binding	Assignment
let x=e in s	x := e; s

Binding

let x=e in s

- x is bound in term s
- *functional*

Assignment

x := e; s

- x is a register
- *imperative*

SSA-based register assignment
Translation from binding to assignment

Intermediate language IL

A linear first-order functional language with external calls

$s, t ::=$	term
$\mid \text{let } x = e \text{ in } s$	variable binding
$\mid \text{let } x = \alpha \text{ in } s$	external call
$\mid \text{if } e \text{ then } s \text{ else } t$	conditional
$\mid e$	value
$\mid \text{fun } f\ \overline{x} = s \text{ in } t$	function definition
$\mid f\ \overline{e}$	application

Intermediate language IL

A linear first-order functional language with external calls

SAARLAND UNIVERSITY
COMPUTER SCIENCE

$s, t ::=$	term
$\mid \text{let } x = e \text{ in } s$	variable binding
$\mid \text{let } x = \alpha \text{ in } s$	external call
$\mid \text{if } e \text{ then } s \text{ else } t$	conditional
$\mid e$	value
$\mid \text{fun } f\ \overline{x} = s \text{ in } t$	function definition
$\mid f\ \overline{e}$	application

1. **First-order** CFGs
   Functions f, g not first-class

Intermediate language IL

A linear first-order functional language with external calls

$s, t ::=$	term
$\mid \text{let } x = e \text{ in } s$	variable binding
$\mid \text{let } x = \alpha \text{ in } s$	external call
$\mid \text{if } e \text{ then } s \text{ else } t$	conditional
$\mid e$	value
$\mid \text{fun } f\ \overline{x} = s \text{ in } t$	function definition
$\mid f\ \overline{e}$	application

1. **First-order** CFGs
   Functions f, g not first-class
2. **Tail-call only** intra-procedural
   $f\ \overline{e}$ only in tail position

Intermediate language IL

A linear first-order functional language with external calls

$s, t ::=$	term
$\mid \text{let } x = e \text{ in } s$	variable binding
$\mid \text{let } x = \alpha \text{ in } s$	external call
$\mid \text{if } e \text{ then } s \text{ else } t$	conditional
$\mid e$	value
$\mid \text{fun } f\ \overline{x} = s \text{ in } t$	function definition
$\mid f\ \overline{e}$	application

1. **First-order** CFGs
   Functions f, g not first-class
2. **Tail-call only** intra-procedural
   $f\ \overline{e}$ only in tail position
3. **Linear** simpl.
   Restricted sequentialization
   $\text{let } x = e \text{ in } s$ (not: $s; t$)

Intermediate language IL

A linear first-order functional language with external calls

$s, t ::=$	term
$\mid \text{let } x = e \text{ in } s$	variable binding
$\mid \text{let } x = \alpha \text{ in } s$	external call
$\mid \text{if } e \text{ then } s \text{ else } t$	conditional
$\mid e$	value
$\mid \text{fun } f\ \overline{x} = s \text{ in } t$	function definition
$\mid f\ \overline{e}$	application

1. **First-order** CFGs
   Functions f, g not first-class
2. **Tail-call only** intra-procedural
   $f\ \overline{e}$ only in tail position
3. **Linear** simpl.
   Restricted sequentialization
   $\text{let } x = e \text{ in } s$ (not: $s; t$)
4. **External calls** realistic
   $\text{let } x = \alpha \text{ in } s$

Example

A functional and an imperative interpretation

$$F(n, m) := n * (n + 1) * \ldots * m$$

Example

A functional and an imperative interpretation

$$F(n, m) := n * (n + 1) * \ldots * m$$

Functional IL

```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let m = p + 1 in
  f (k,m)
 else
  j
in f (i,n)
```

Example

A functional and an imperative interpretation

$$F(n, m) := n * (n + 1) * \ldots * m$$

Functional IL

```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let m = p + 1 in
  f (k,m)
 else
  j
in f (i,n)
```

Imperative IL/I

```
i := 1;
fun f (j,p) =
 c := p <= m;
 if c then
  k := p * j;
  m := p + 1;
  f (k,m)
 else
  j
in f (i,n)
```

Example

A functional and an imperative interpretation

$$F(n, m) := n * (n + 1) * \ldots * m$$

Functional IL

```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let m = p + 1 in
  f (k,m)
 else
  j
in f (i,n)
```

No closure created: **goto**

Imperative IL/I

```
i := 1;
fun f (j,p) =
 c := p <= m;
 if c then
  k := p * j;
  m := p + 1;
  f (k,m)
 else
  j
in f (i,n)
```

Parameter passing in IL/I is parallel assignment:

$$j, p := k, m$$

Example

A functional and an imperative interpretation

SAARLAND UNIVERSITY
COMPUTER SCIENCE

$$F(n, m) := n * (n + 1) * \ldots * m$$

Functional IL	Imperative IL/I

Functional IL

```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let m = p + 1 in
  f (k,m)
 else
  j
in f (i,n)
```

Imperative IL/I

```
i := 1;
fun f (j,p) =
 c := p <= m;
 if c then
  k := p * j;
  m := p + 1;
  f (k,m)
 else
  j
in f (i,n)
```

Example

A functional and an imperative interpretation

$$F(n, m) := n * (n + 1) * \ldots * m$$

Functional IL	Imperative IL/I

Functional IL

```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let x = p + 1 in
  f (k,x)
 else
  j
in f (i,n)
```

Imperative IL/I

```
i := 1;
fun f (j,p) =
 c := p <= m;
 if c then
  k := p * j;
  m := p + 1;
  f (k,m)
 else
  j
in f (i,n)
```

Example

A functional and an imperative interpretation

SAARLAND UNIVERSITY
COMPUTER SCIENCE

$$F(n, m) := n * (n+1) * \ldots * m$$

Functional IL

```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let x = p + 1 in
  f (k,x)
 else
  j
in f (i,n)
```

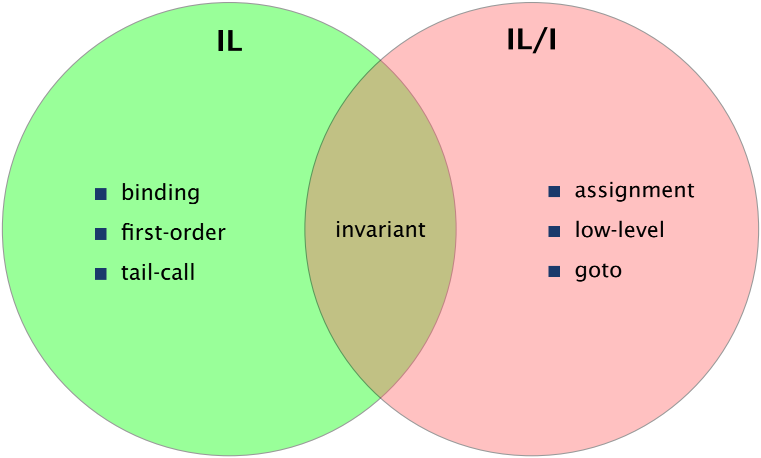
Imperative IL/I

```
i := 1;
fun f (j,p) =
 c := p <= m;
 if c then
  k := p * j;
  x := p + 1;
  f (k,x)
 else
  j
in f (i,n)
```

- When renamed-apart, binding and assignment interchangeable!

Overview

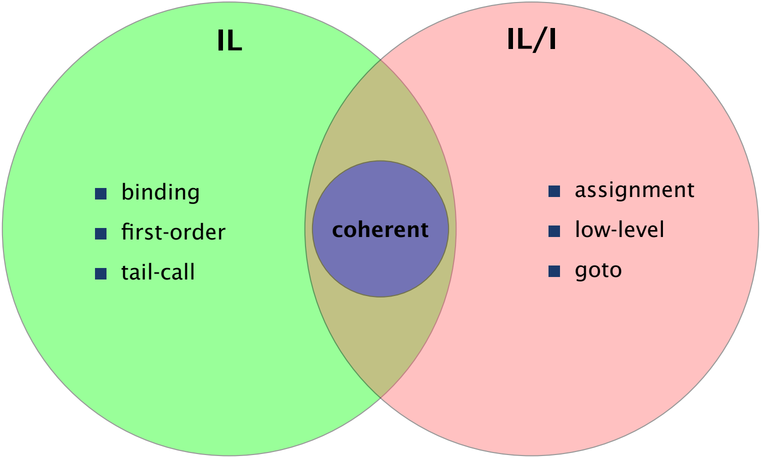
Translating from the functional to the imperative interpretation



Overview

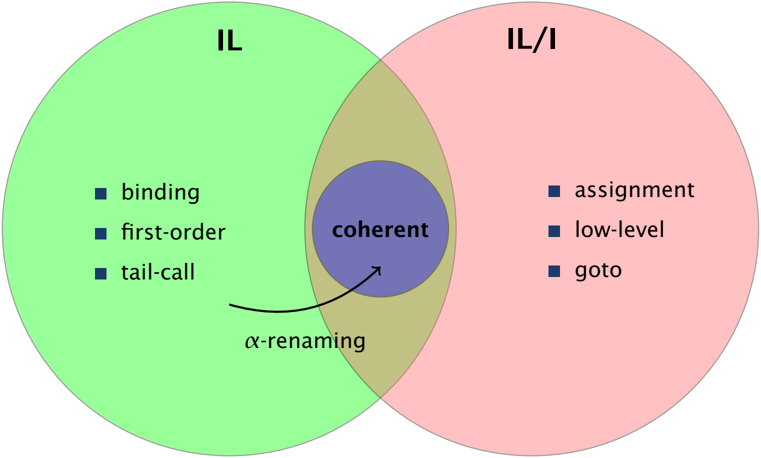
SAARLAND UNIVERSITY
COMPUTER SCIENCE

Translating from the functional to the imperative interpretation



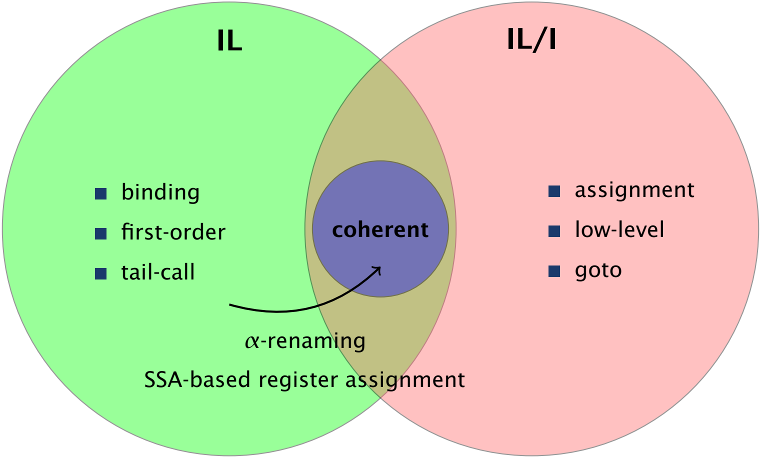
Overview

Translating from the functional to the imperative interpretation



Overview

Translating from the functional to the imperative interpretation



Related work

Static single assignment (SSA)

Related work

IL

```
let i = 1 in
fun f (j,p) =

 let c = p <= m in
 if c then
  let k = p * j in
  let x = p + 1 in
  f (k,x)
 else
  j
in f (i,n)
```

SSA

```
 i := 1;
f:
 j := φ(i,k), p := φ(n,x)
 c := n <= m;
 if c then
  k := p * j;
  x := p + 1;
  goto f
 else
  return i
```

- SSA •—• CPS due to Appel (1998) and Kelsey (1995).
- Chakravarty et al. (2003) reformulates SSA optimization on a functional language in ANF (Sabry et al. 1993).
- IL is a sub-language (up to system calls)

SSA in verified compilers

Related work

1. CompCertSSA: Barthe et al. (2012)
   - Integrates SSA-based optimization passes in CompCert (Leroy (2009))
2. VeLLVM: Zhao et al. (2012)
   - Verifies some SSA-based passes of LLVM

- SSA for optimizations
  - performance of data-flow analyses
- ϕ-functions
  - no functional language
  - underlying semantics uses imperative variables

SSA-based register allocation

Related work

- SSA-based register allocation (Hack et al. (2006))
  - allows phase separation of spilling and register assignment
  - IL version similar to Appel (1992)
  - not considered in verified setting so far:
    out of SSA + non-SSA register allocation
- Blazy et al. (2010) verify non-SSA register allocation algorithm (which must include spilling)
- We only considering register assignment, because SSA-based algorithm allows spilling to be separate phase

Functional and imperative semantics

Related work

- Beringer et al. (2003) use a language with a functional and imperative interpretation for proof-carrying code.
- Grail normal form (GNF) sufficient for functional + imperative semantics to coincide
- Main difference: GNF requires functions to be closure converted, i.e. all variables a function body depends on must be parameters

Contributions

- Coherence
  - relates binding and assignment directly
  - another perspective on SSA and functional programming
- SSA-based register assignment on IL
  - formal correctness proof (using coherence)
  - key property from SSA holds on IL:
    spilling can be considered separately (not possible without SSA)
- Coq development available online:
  `www.ps.uni-saarland.de/~sdschn/publications/lvc15`

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Semantics and program equivalence

Semantics of IL and IL/I

Reduction, events, configurations

- Small-step relation $\xrightarrow{\phi}$
- Decorated with events ϕ

$$\phi ::= \tau \qquad \text{silent event}$$
$$\mid v = \alpha \qquad \text{external event}$$

- Configurations

$$\text{IL: } (F, V, s) \qquad \text{IL/I: } (L, V, s)$$

  - F function env. (with closures)
  - L block env. (no closures)
  - V variable env.
  - s program

Program equivalence

Non-determinism and equivalence

$$\textsc{Extern}\quad \frac{v \in \mathbb{V}}{\begin{array}{ll} & F \mid V \qquad\quad \mid \mathrm{let}\ x = \alpha\ \mathrm{in}\ s \\ \xrightarrow{v=\alpha} & F \mid V[x \mapsto v] \mid s \end{array}}$$

- $\xrightarrow{\phi}$ forms a LTS
- Internally deterministic reduction systems (IDRS)
  - $\sigma \xrightarrow{\phi} \sigma_1 \wedge \sigma \xrightarrow{\tau} \sigma_2 \Rightarrow \phi = \tau$ — τ-deterministic
  - $\sigma \xrightarrow{\phi} \sigma_1 \wedge \sigma \xrightarrow{\phi} \sigma_2 \Rightarrow \sigma_1 = \sigma_2$ — action-deterministic
- Configurations are equivalent ($\simeq$), if they allow the same partial traces

$$\pi ::= \epsilon \mid v \mid \bot \mid v = \alpha, \pi$$

- Sound and complete characterization via (stutter) bisimulation

Liveness

Liveness

Judgment

$$(L, V, s) \stackrel{?}{\simeq} (L, W, s)$$

Liveness

Judgment

SAARLAND UNIVERSITY
COMPUTER SCIENCE

$$V =_X W \Rightarrow (L, V, s) \stackrel{?}{\simeq} (L, W, s)$$

Liveness

Judgment

$$V =_X W \Rightarrow (L, V, s) \overset{?}{\simeq} (L, W, s)$$

$$\boxed{\Lambda \vdash \textbf{live}\; s : X}$$

Λ	live variables of functions
s	program
X	set of live variables

- embedded liveness analysis results as annotations in syntax:

$$\text{fun}\; f\; \overline{x} : X_1 = s_1 \;\text{in}\; s_2$$

- syntactic structure allows for inductive specification
- useful for imperative IL/I
- judgment monotonic in X (larger sets are sound)

Liveness

Properties for IL/I

Theorem (Decidability)

$\Lambda \vdash \textbf{live}\, s : X$ *decidable.*

Theorem (Soundness)

If

1. $\Lambda \vdash \textbf{live}\, s : X$ — *liveness information sound*
2. $L \vDash \Lambda$ — *Λ sound for blocks L*
3. $V =_X W$ — *V, W agree on live set X*

then

$$(L, V, s) \simeq (L, W, s)$$

Coherence

Coherence

Intuition

$$F, f : (W, \overline{x}, s) \mid V \mid f\,\overline{e} \quad\longrightarrow\quad F, f : (W, \overline{x}, s) \mid W[\overline{x} \mapsto \overline{v}] \mid s$$

$$\overset{?}{\simeq}$$

$$F, f : (W, \overline{x}, s) \mid V[\overline{x} \mapsto \overline{v}] \mid s$$

Coherence

Intuition

$$F, f : (W, \overline{x}, s) \mid V \mid f\,\overline{e} \quad \longrightarrow \quad F, f : (W, \overline{x}, s) \mid W[\overline{x} \mapsto \overline{v}] \mid s$$

$$\overset{?}{\simeq}$$

$$F, f : (W, \overline{x}, s) \mid V[\overline{x} \mapsto \overline{v}] \mid s$$

1. If $\boxed{\Lambda \vdash \textbf{live}\; s : X}$ then it suffices if W and V agree on $X \setminus \overline{x}$
2. Call $X \setminus \overline{x}$ **globals of function** f
3. Liveness definition is arranged such that context Λ records globals
4. Define coherence to ensure environments agree on globals at every application

Coherence

Inductive definition

SAARLAND UNIVERSITY COMPUTER SCIENCE

f *available* as long as no global rebound

Coherence

Inductive definition

f available as long as no global rebound

not invariant

```
let x = 7 in
fun f () : {x} = x in
let x = 5 in
f ()
```

Coherence

Inductive definition

f available as long as no global rebound

not invariant

```
1 let x = 7 in
2 fun f () : {x} = x in
3 let x = 5 in
4 f ()
```

f *unavailable* after line 3

Coherence

Inductive definition

f available as long as no global rebound

not invariant	coherent

```
let x = 7 in
fun f () : {x} = x in
let x = 5 in
f ()
```

f *unavailable* after line 3

```
let x = 7 in
fun f () : {x} = x in
let y = 5 in
f ()
```

f *available* in line 4

Coherence

Inductive definition

f available as long as no global rebound

not invariant

```
1 let x = 7 in
2 fun f () : {x} = x in
3 let x = 5 in
4 f ()
```

f *unavailable* after line 3

coherent

```
1 let x = 7 in
2 fun f () : {x} = x in
3 let y = 5 in
4 f ()
```

f *available* in line 4

Coherence judgment $\Lambda \vdash \mathbf{coh}\ s$

- ensures s only applies available functions
- defined relative to liveness information

Coherence

Rules

$\Lambda - \{x\}$ removes definitions from Λ that require x as global

$$\textsc{Coh-Op}\ \frac{\Lambda - \{x\} \vdash \textbf{coh}\ s}{\Lambda \vdash \textbf{coh}\ \text{let}\ x = e\ \text{in}\ s}$$

$$\textsc{Coh-App}\ \frac{\Lambda f \neq \bot}{\Lambda \vdash \textbf{coh}\ f\ \overline{y}}$$

Coherence

Rules

$\Lambda - \{x\}$ removes definitions from Λ that require x as global

$$\textsc{Coh-Op}\ \frac{\Lambda - \{x\} \vdash \textbf{coh}\ s}{\Lambda \vdash \textbf{coh}\ \text{let}\, x = e\, \text{in}\, s}$$

$$\textsc{Coh-App}\ \frac{\Lambda f \neq \bot}{\Lambda \vdash \textbf{coh}\ f\,\overline{y}}$$

$\lfloor \Lambda \rfloor_X$ removes definitions from Λ that require more globals than X

$$\textsc{Coh-Fun}\ \frac{\Lambda; f : X \vdash \textbf{coh}\ t \qquad \lfloor \Lambda; f : X \rfloor_X \vdash \textbf{coh}\ s}{\Lambda \vdash \textbf{coh}\ \text{fun}\, f\,\overline{x} : X = s\, \text{in}\, t}$$

Coherence

Results

We define $strip(V, \overline{x}, s) \;=\; (\overline{x}, s)$ and lift *strip* pointwise to contexts.

Theorem (Coherence implies invariance)

If

1. $\Lambda \vdash \textbf{coh}\ s$ — *s is coherent*
2. $\Lambda \vdash \textbf{coh}\ F$ — *definitions in F are coherent*
3. $\Lambda' \vdash \textbf{live}\ s : X$ *for* $\Lambda \preceq \Lambda'$ — *liveness information is sound*
4. $V =_X W$ — *V, W agree on X*
5. $F, V \vDash \Lambda$ — *closures in F agree with V on globals*

then

$$(F, V, s)_F \simeq (strip\, F, W, s)_I$$

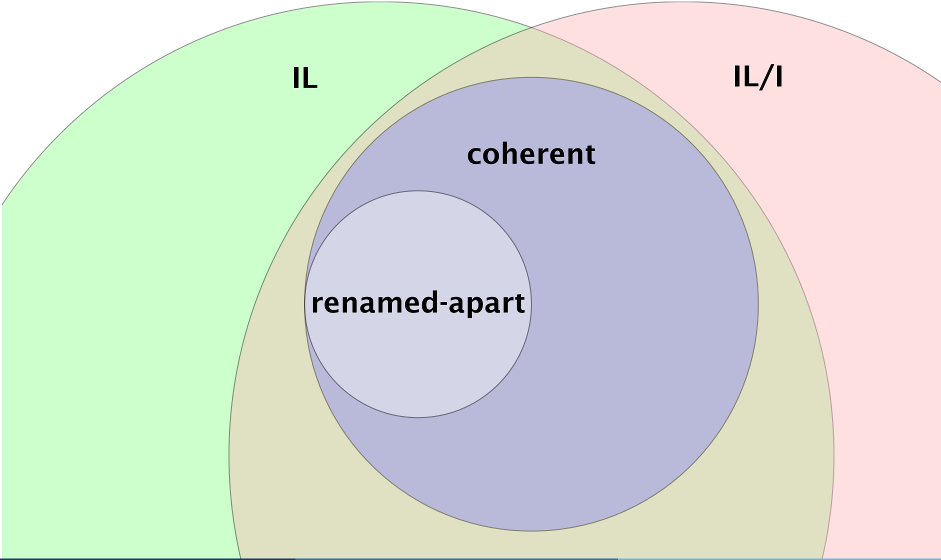
Register assignment

Register assignment

- State-of-the-art SSA-based register assignment algorithm
  - decouples spilling from assignment:
    number of registers bounded by largest live set
  - polynomial-time (coalescing is NP-hard)
  - critically depends on dominance ordering
- Register assignment for functional language IL
  - same properties: register bound, polynomial time
  - straight-forward recursion on syntax
- Correctness argument of assignment phase
  - does not involve dominance
  - via coherence and α-equivalence

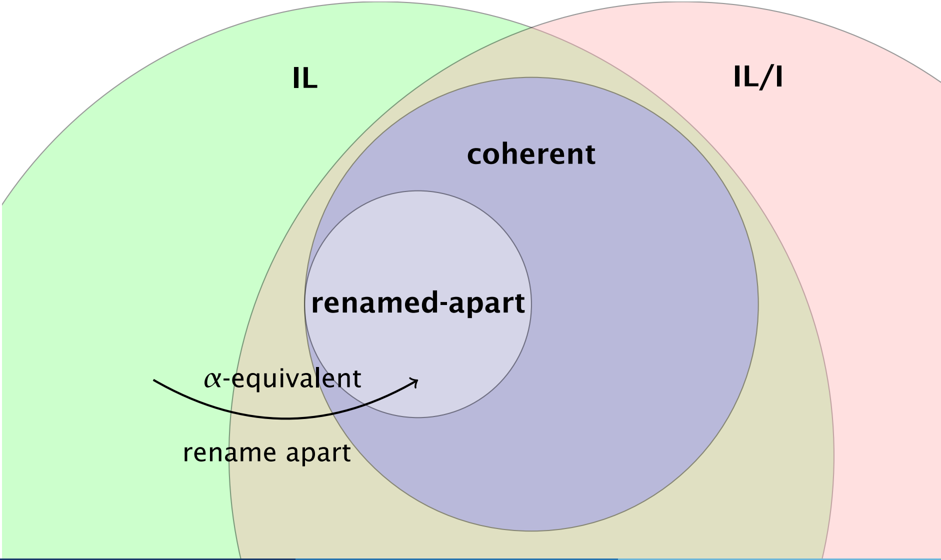
Register assignment

Proof overview



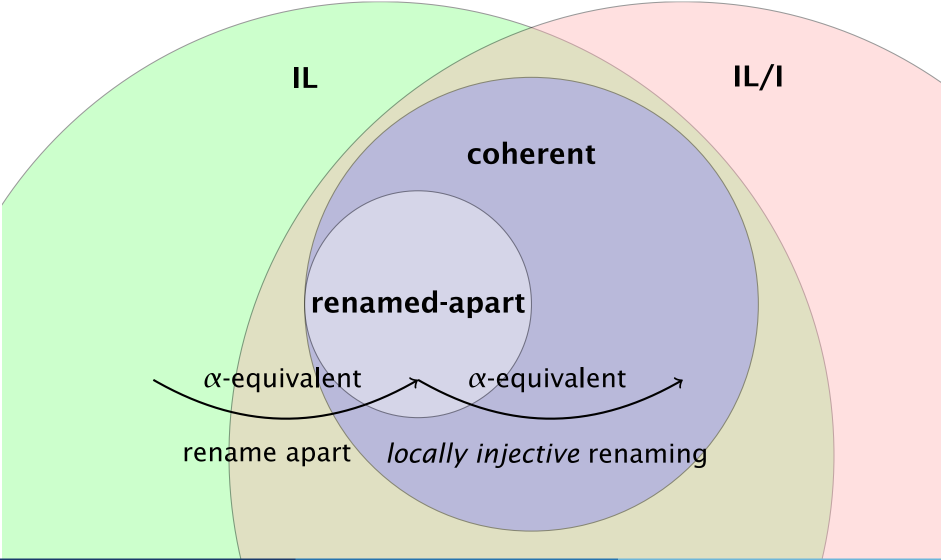
Register assignment

Proof overview



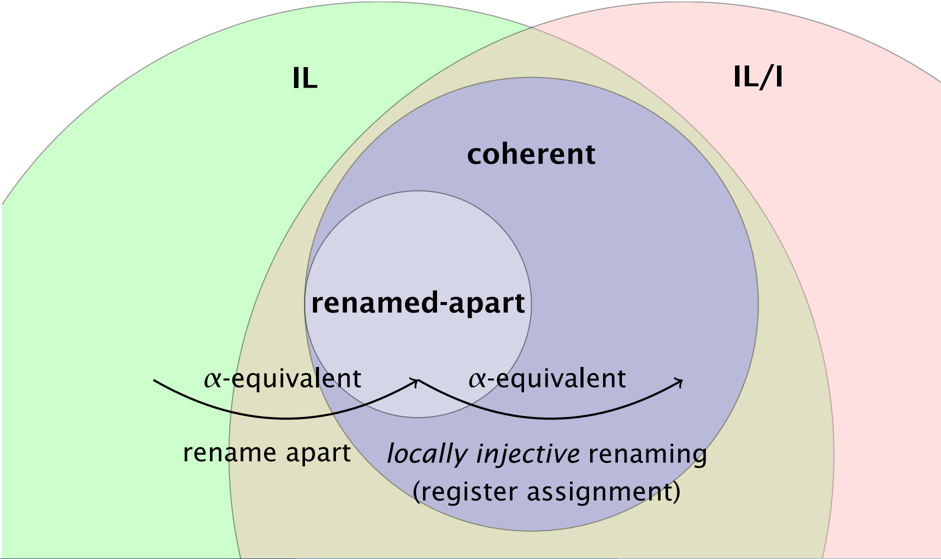
Register assignment

Proof overview



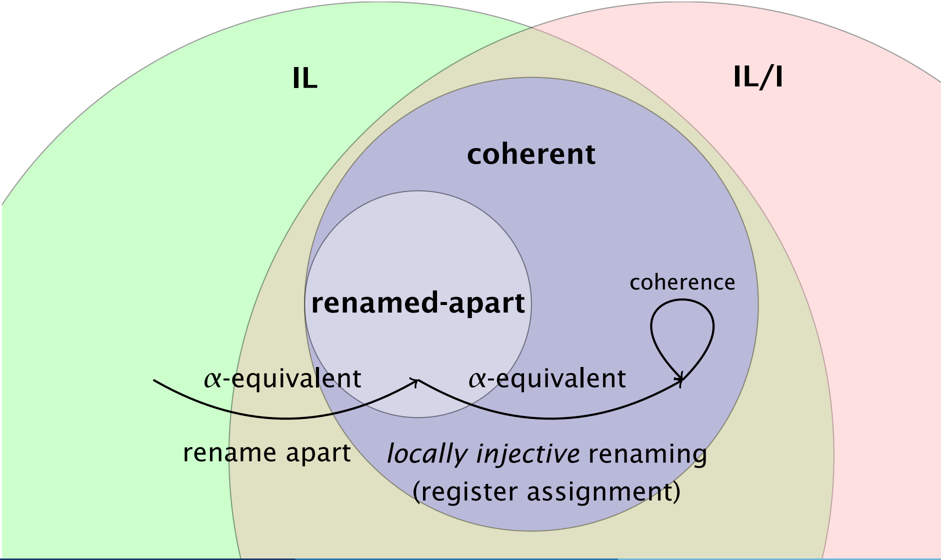
Register assignment

Proof overview



Register assignment

Proof overview



Register assignment

Overview and example


```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let m = p + 1 in
  f (k,m)
 else
  j
in f (i,n)
```

Register assignment

Overview and example


```
let i = 1 in
fun f (j,p) =
 let c = p <= m in
 if c then
  let k = p * j in
  let x = p + 1 in
  f (k,x)
 else
  j
in f (i,n)
```

1. Rename apart
   - Every assignment can be represented as $\rho : \mathcal{V} \to \mathcal{V}$

Register assignment

Overview and example


```
let i = 1 in
fun f (i,n) =
 let c = n <= m in
 if c then
  let i = n * i in
  let n = n + 1 in
  f (i,n)
 else
  i
in f (i,n)
```

1. Rename apart
   - Every assignment can be represented as $\rho : \mathcal{V} \to \mathcal{V}$
2. Rename with *locally injective* ρ
   - A ρs is α-equivalent and coherent
   - B register assignment algorithm yields locally injective renaming

Register assignment

Overview and example

```
let i = 1 in
fun f (i,n) =
 let c = n <= m in
 if c then
  let i = n * i in
  let n = n + 1 in
  f (i,n)
 else
  i
in f (i,n)
```

```
i := 1;
fun f (i,n) =
 c := n <= m;
 if c then
  i := n * i;
  n := n + 1;
  f (i,n)
 else
  i
in f (i,n)
```

1. Rename apart
   - Every assignment can be represented as $\rho : \mathcal{V} \to \mathcal{V}$
2. Rename with *locally injective* ρ

   A ρs is α-equivalent and coherent

   B register assignment algorithm yields locally injective renaming
3. Reinterpret binding as assignment: IL/I

Side Conditions: Liveness and Renamed-Apart

- Call $\boxed{\Lambda \text{ and } s \textit{ suitable}}$ if
  1. s renamed-apart
  2. $\Lambda \vdash \textbf{live}\, s : X$ — liveness sound
     - ⋆ write $[s]$ for X

Side Conditions: Liveness and Renamed-Apart

- Call $\boxed{\Lambda \text{ and } s \textit{ suitable}}$ if

  1. s renamed-apart
  2. $\Lambda \vdash \textbf{live}\, s : X$ — liveness sound
     - ⋆ write $[s]$ for X
  3. $s \subseteq \text{fv}(s)$ — no variable occurs in annotation before it is bound

```
fun f () : {y} = 7 in
let y = 5 in
f ()
```

Side Conditions: Liveness and Renamed-Apart

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Call $\boxed{\Lambda \text{ and } s \textit{ suitable}}$ if
  1. s renamed-apart
  2. $\Lambda \vdash \textbf{live}\, s : X$ — liveness sound
     - write $\lceil s \rceil$ for X
  3. $\lceil s \rceil \subseteq \text{fv}(s)$ — no variable occurs in annotation before it is bound

```
fun f () : {y} = 7 in
let y = 5 in
f ()
```

  4. $\Lambda \subseteq \text{fv}(s)$ — no global from Λ bound in s

```
fun f () : {x} = x in
let y = 5 in
f ()
```

```

let y = 5 in
f ()
```

Side Conditions: Liveness and Renamed-Apart

- Call $\boxed{\Lambda \text{ and } s \textit{ suitable}}$ if
  1. s renamed-apart
  2. $\Lambda \vdash \textbf{live}\, s : X$ — liveness sound
     - ⋆ write $[s]$ for X
  3. $s \subseteq \text{fv}(s)$ — no variable occurs in annotation before it is bound

```
fun f () : {y} = 7 in
let y = 5 in
f ()
```

  4. $\Lambda \subseteq \text{fv}(s)$ — no global from Λ bound in s

```
fun f () : {x} = x in
let y = 5 in
f ()
```

```

let y = 5 in
f ()
```

Local Injectivity

Local Injectivity $\boxed{\rho \vdash \textbf{\textit{inj}}\ s}$ requires $\rho : \mathcal{V} \to \mathcal{V}$ to be injective on every live set X that appears in the liveness derivation.

ρ^{-X}: inverse of ρ on X,

Local Injectivity

Local Injectivity $\boxed{\rho \vdash \textbf{\textit{inj}}\ s}$ requires $\rho : \mathcal{V} \rightarrow \mathcal{V}$ to be injective on every live set X that appears in the liveness derivation.

Theorem (A)

If

1. Λ *and* s *suitable*
2. s *without unreachable code*
3. $\rho \vdash \textbf{\textit{inj}}\ s$ — ρ *locally injective*

then

1. $\rho\ (\lfloor \Lambda \rfloor_{[s]}) \vdash \textbf{coh}\ (\rho\ s)$ — ρs *coherent*
2. ρ *injective on* $\text{fv}(s) \Longrightarrow \rho, \rho^{-\text{fv}(s)} \vdash \rho\ s \sim_\alpha s$ — ρs *α-equivalent to* s

---

ρ^{-X}: inverse of ρ on X,

Register assignment algorithm

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Definition

- Assume $\mathrm{fresh} : \mathit{set}\,\mathcal{V} \to \mathcal{V}$ such that $\mathrm{fresh}\,X \notin X$ for all finite X.
  - separation of concerns: correctness and code quality

Register assignment algorithm

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Definition

- Assume $\mathrm{fresh} : set\,\mathcal{V} \to \mathcal{V}$ such that $\mathrm{fresh}\,X \notin X$ for all finite X.
  - separation of concerns: correctness and code quality
- $\mathrm{rassign}$ yields renaming $\mathcal{V} \to \mathcal{V}$

Register assignment algorithm

Definition

- Assume fresh : $set\,\mathcal{V} \to \mathcal{V}$ such that $\text{fresh}\,X \notin X$ for all finite X.
  - separation of concerns: correctness and code quality
- rassign yields renaming $\mathcal{V} \to \mathcal{V}$

$$
\begin{array}{lcl}
\text{rassign}\,\rho\,(\text{let}\,x = e\,\text{in}\,s) & = & \text{rassign}\,(\rho[x \mapsto y])\,s \\
\quad \text{where } y = \text{fresh}\,(\rho([s] \setminus \{x\})) & & \\
\text{rassign}\,\rho\,(\text{if}\,e\,\text{then}\,s\,\text{else}\,t) & = & \text{rassign}\,(\text{rassign}\,\rho\,s)\,t \\
\text{rassign}\,\rho\,e & = & \rho \\
\text{rassign}\,\rho\,(f\,\overline{e}) & = & \rho \\
\text{rassign}\,\rho\,(\text{fun}\,f\,\overline{x}\,:\,X' = s\,\text{in}\,t) & = & \text{rassign}\,(\text{rassign}\,(\rho[\overline{x} \mapsto \overline{y}])\,s)\,t \\
\quad \text{where } \overline{y} = \text{freshlist}\,(\rho([s] \setminus \overline{x}))\,|\overline{x}| & &
\end{array}
$$

rassign recurses on program structure,
while SSA algorithm must process statements in dominance order

Register assignment algorithm

Correctness and bound on registers

Theorem (B)

Let Λ *and* s *suitable and* ρ *injective on* $[s]$*. Then:* $\mathrm{rassign}\,\rho\, s \vdash$ ***inj*** s.

Assume variables totally ordered: $x_0 < x_1 < x_2 \ldots$

Theorem (Register Bound)

If

1. Λ *and* s *suitable*
2. $\forall$ *finite sets of variables* Y*: fresh* $Y \in \{x_0, \ldots, x_{|Y|}\}$
3. k *is size of largest set of live variables in* s
4. $\rho(\mathrm{fv}(s)) \subseteq \{x_0, \ldots, x_n\}$.
5. $\rho' = \mathrm{rassign}\,\rho\, s$

Then $\rho'\,(\mathcal{V}_O(s)) \subseteq \{x_0, \ldots, x_{max\{n,k\}}\}$

Coq Development

- This work is part of a very simple verified compiler
- Extraction yields binary that handles the running example
  - Efficient finite set library with type classes (Lescuyer 2012)
  - Cannot assume set extensionality
  - Decision procedures for equivalence on many types
- Development almost completely constructive
  - UIP required for Paco Library (Hur et al. (2013))
- Formal development contains proofs of
  - Backwards translation: IL/I to IL (SSA-construction)
  - Dead code elimination
  - Sparse conditional constant propagation
  - Translation validation for analysis results

Conclusion

- Coherence relates binding to assignment
- Correctness proof of register assignment on IL
  - same advantages as SSA (register bound)
  - correctness via coherence and α-equivalence
  - structural recursion instead of dominance ordering
- Coq development is available online[1]

[1] www.ps.uni-saarland.de/~sdschn/publications/lvc15

Conclusion

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Coherence relates binding to assignment
- Correctness proof of register assignment on IL
  - same advantages as SSA (register bound)
  - correctness via coherence and α-equivalence
  - structural recursion instead of dominance ordering
- Coq development is available online[1]

Thanks! Questions?

[1] www.ps.uni-saarland.de/~sdschn/publications/lvc15

Thank you for your attention! Questions? I

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Appel, A. W. (1992). *Compiling with Continuations*. Cambridge, England: Cambridge University Press.

— (1998). "SSA is Functional Programming". In: *SIGPLAN Notices* 33.4.

Barthe, G. et al. (2012). "A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert". In: *ESOP*.

Beringer, L. et al. (2003). "Grail: a Functional Form for Imperative Mobile Code". In: *ENTCS* 85.1.

Blazy, S. et al. (2010). "Formal Verification of Coalescing Graph-Coloring Register Allocation". In: *ESOP*.

Chakravarty, M. M. T. et al. (2003). "A Functional Perspective on SSA Optimisation Algorithms". In: *ENTCS* 82.2.

Hack, S. et al. (2006). "Register Allocation for Programs in SSA-Form". In: *CC*.

Hur, C. et al. (2013). "The power of parameterization in coinductive proof". In: *POPL*.

Kelsey, R. A. (1995). "A correspondence between continuation passing style and static single assignment form". In: *SIGPLAN Not.* 30 (3).

Leroy, X. (2009). "Formal Verification of a Realistic Compiler". In: *CACM* 52.7.

Thank you for your attention! Questions? II

Lescuyer, S. (2012). *Containers: a typeclass-based library of finite sets/maps*. URL: http://coq.inria.fr/pylons/contribs/view/Containers/v8.4.

Sabry, A. and M. Felleisen (1993). "Reasoning about Programs in Continuation-Passing Style". In: *LSC* 6.3-4.

Zhao, J. et al. (2012). "Formalizing LLVM Intermediate Representation for Verified Program Transformations". In: *POPL*.

Semantics of IL and IL/I

Common rules

$\phi ::= \tau \mid v = \alpha$ events

$\xrightarrow{\phi}$ small step relation

Semantics of IL and IL/I

Common rules

$\phi ::= \tau \mid v = \alpha$ events

$\xrightarrow{\phi}$ small step relation

(F, V, s)

F	function env.
V	variable env.
s	program

Semantics of IL and IL/I

Common rules

$\phi ::= \tau \mid v = \alpha$ events

$\xrightarrow{\phi}$ small step relation

(F, V, s)
- F function env.
- V variable env.
- s program

Op

$$\frac{[\![e]\!]\,V = v}{\begin{array}{ll} & F \mid V \mid \text{let } x = e \text{ in } s \\ \xrightarrow{\tau} & F \mid V[x \mapsto v] \mid s \end{array}}$$

Extern

$$\frac{v \in \mathbb{V}}{\begin{array}{ll} & F \mid V \mid \text{let } x = \alpha \text{ in } s \\ \xrightarrow{v=\alpha} & F \mid V[x \mapsto v] \mid s \end{array}}$$

Cond

$$\frac{[\![e]\!]\,V = v \qquad \beta(v) = i}{\begin{array}{ll} & F \mid V \mid \text{if } e \text{ then } s_0 \text{ else } s_1 \\ \xrightarrow{\tau} & F \mid V \mid s_i \end{array}}$$

Semantics IL and IL/I

Differences

IL

FUN

$$\frac{}{F \mid V \mid \mathrm{fun}\, f\, \overline{x} = s\, \mathrm{in}\, t \xrightarrow{\tau} F; f : (V, \overline{x}, s) \mid V \mid t}$$

APP

$$\frac{[\![\overline{e}]\!]\, V = \overline{v} \qquad Ff = (W, \overline{x}, s)}{F \mid V \mid f\, \overline{e} \xrightarrow{\tau} F^f \mid W[\overline{x} \mapsto \overline{v}] \mid s}$$

Semantics IL and IL/I

Differences

SAARLAND UNIVERSITY
COMPUTER SCIENCE

IL

FUN

$$\frac{}{F \mid V \mid \mathrm{fun}\, f\, \overline{x} = s\, \mathrm{in}\, t \;\xrightarrow{\tau}\; F; f : (\mathrm{V}, \overline{x}, s) \mid V \mid t}$$

APP

$$\frac{[\![\overline{e}]\!]\, V = \overline{v} \qquad Ff = (\mathrm{W}, \overline{x}, s)}{F \mid V \mid f\, \overline{e} \;\xrightarrow{\tau}\; F^f \mid \mathrm{W}[\overline{x} \mapsto \overline{v}] \mid s}$$

IL/I

I-FUN

$$\frac{}{L \mid V \mid \mathrm{fun}\, f\, \overline{x} = s\, \mathrm{in}\, t \;\xrightarrow{\tau}_I\; L; f : (\overline{x}, s) \mid V \mid t}$$

I-APP

$$\frac{[\![\overline{e}]\!]\, V = \overline{v} \qquad Lf = (\overline{x}, s)}{L \mid \mathrm{V} \mid f\, \overline{e} \;\xrightarrow{\tau}_I\; L^f \mid \mathrm{V}[\overline{x} \mapsto \overline{v}] \mid s}$$

Program equivalence

Internally deterministic reduction systems

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Definition

A *reduction system* (RS) is a tuple $(\Sigma, \mathcal{E}, \longrightarrow, \tau, res)$, s.t.

1. $(\Sigma, \mathcal{E}, \longrightarrow)$ is a LTS
2. $\tau \in \mathcal{E}$
3. $res : \Sigma \to \mathbb{V}_\perp$
4. $res\, \sigma = v \Rightarrow \sigma$ $\longrightarrow$-terminal

An *internally deterministic* reduction system (IDRS) additionally satisfies

5. $\sigma \xrightarrow{\phi} \sigma_1 \wedge \sigma \xrightarrow{\phi} \sigma_2 \Rightarrow \sigma_1 = \sigma_2$ action-deterministic
6. $\sigma \xrightarrow{\phi} \sigma_1 \wedge \sigma \xrightarrow{\tau} \sigma_2 \Rightarrow \phi = \tau$ τ-deterministic

Program equivalence

Trace equivalence

$\Pi \ni \pi ::= \epsilon \mid v \mid \perp \mid \phi\pi$	$\phi \neq \tau$	partial trace
$\sigma \triangleright \pi$		σ poduces π

Program equivalence

Trace equivalence

$$\Pi \ni \pi ::= \epsilon \mid v \mid \bot \mid \phi\pi \qquad \phi \neq \tau \qquad \text{partial trace}$$

$$\sigma \rhd \pi \qquad \sigma \text{ poduces } \pi$$

Definition (Trace equivalence)

$\sigma \simeq \sigma' :\Longleftrightarrow \forall \pi, \sigma \rhd \pi \iff \sigma' \rhd \pi$

Program equivalence

Trace equivalence

$\Pi \ni \pi ::= \epsilon \mid v \mid \bot \mid \phi\pi \qquad \phi \neq \tau$ partial trace

$\sigma \rhd \pi$ σ poduces π

Definition (Trace equivalence)

$\sigma \simeq \sigma' :\Longleftrightarrow \ \forall \pi, \sigma \rhd \pi \iff \sigma' \rhd \pi$

$\sigma \sim \sigma'$ bisimilarity

Program equivalence

Trace equivalence

$\Pi \ni \pi ::= \epsilon \mid v \mid \bot \mid \phi\pi \qquad \phi \neq \tau$ partial trace

$\sigma \rhd \pi$ σ poduces π

Definition (Trace equivalence)

$\sigma \simeq \sigma' :\iff \forall \pi, \sigma \rhd \pi \iff \sigma' \rhd \pi$

$\sigma \sim \sigma'$ bisimilarity

Theorem (Soundness and completeness)

Let $(S, \mathcal{E}, \longrightarrow, res, \tau)$ be an IDRS and $\sigma, \sigma' \in S$. Then:

$$\sigma \sim \sigma' \iff \sigma \simeq \sigma'$$

Local Injectivity

The problem with unreachable code

```
fun f () = x in
y
```

```
fun f () = y in
y
```

- $\{x \mapsto y, y \mapsto y\}$ locally injective
- Programs not α-equivalent