Translating In and Out of a Functional Intermediate Language

Sigurd Schneider, Sebastian Hack, Gert Smolka

Talk at Inria Rocquencourt

17.03.2014

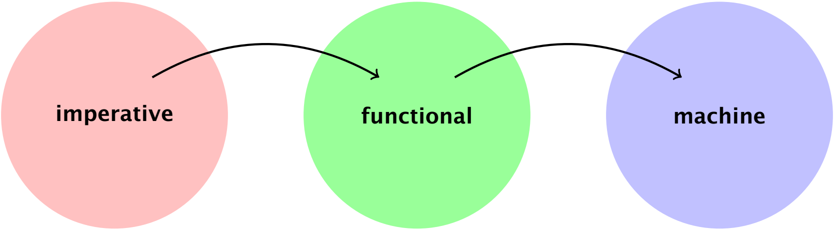
SAARLAND UNIVERSITY
COMPUTER SCIENCE

Overview

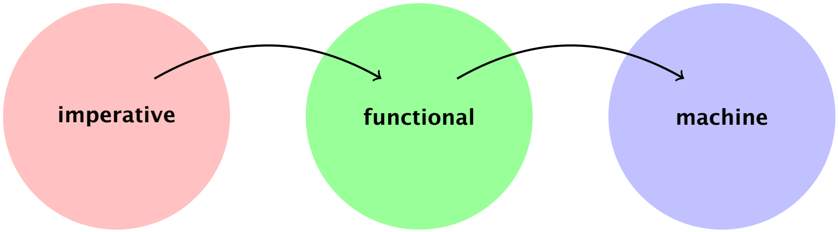
imperative **functional** **machine**

Overview

SAARLAND UNIVERSITY
COMPUTER SCIENCE



Overview



Why? **How?**

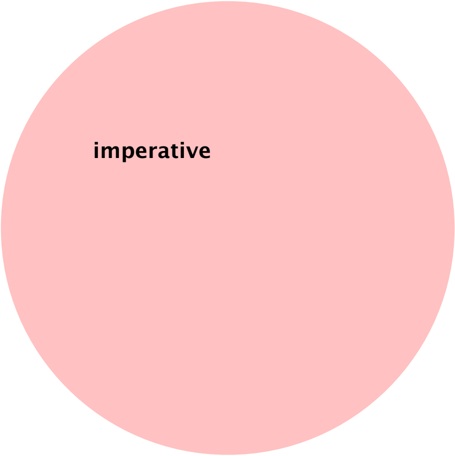
Overview

Why a functional intermediate language

1. Modern IRs are evolving towards "functional style"
   - Static Single Assignment (SSA), Higher-order functions
   - VSDG [Johnson and Mycroft 2003], PEG [Tate et al. 2009], Firm-IR [Braun, Buchwald, and Zwinkau 2011]
2. Compositional program equivalence
   - Syntactic proofs instead of semantic proofs?
   - Induction instead of coinduction?
3. In this talk
   - Translation: Imperative $\bullet\!\!-\!\!\bullet$ Functional
   - Relation to SSA-construction and register assignment
   - Inductive correctness proof of copy propagation

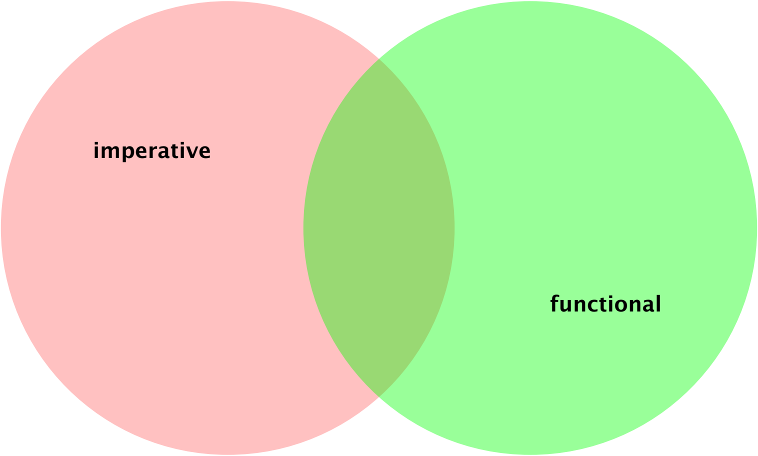
Overview

How to integrate a functional intermediate language



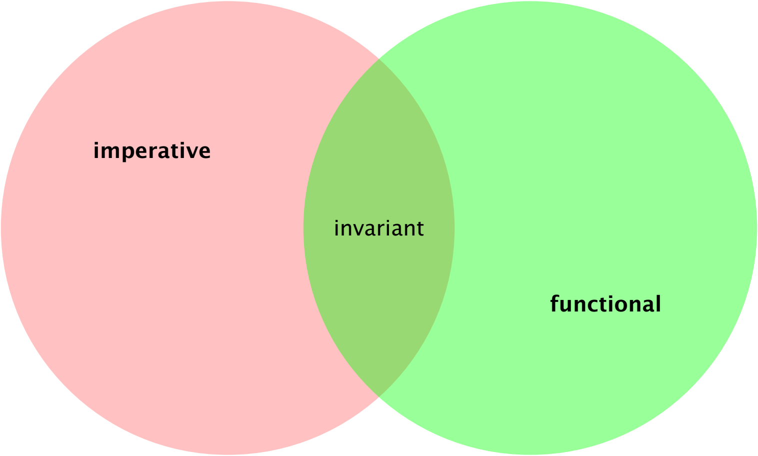
Overview

How to integrate a functional intermediate language



Overview

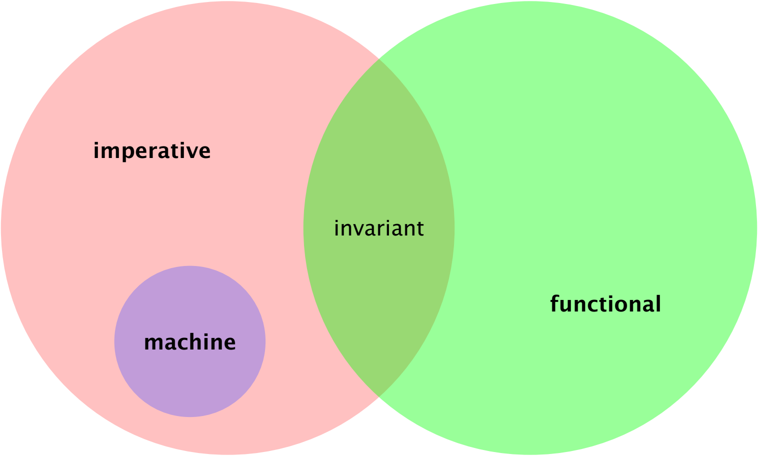
How to integrate a functional intermediate language



1 Proof-carrying code: Beringer, MacKenzie, and Stark (2003)

Overview

How to integrate a functional intermediate language



1. Proof-carrying code: Beringer, MacKenzie, and Stark (2003)
2. Compiler Construction: R. Kelsey and Hudak (1989)

Syntax of IL

$$
\begin{array}{rll}
s, t ::= & \mathit{let}\ x\ =\ e\ \mathit{in}\ s & \text{let (base-types)} \\
\mid & \mathit{if}\ \ x\ \mathit{then}\ s\ \mathit{else}\ t & \text{conditional} \\
\mid & x & \text{value} \\
\mid & \mathit{fun}\ f\ \overline{x}\ =\ s\ \mathit{in}\ t & \text{recursive function} \\
\mid & f\ \overline{x} & \text{application}
\end{array}
$$

- First-order functional language with tail-call restriction
- Fragment of ANF language of Chakravarty, Keller, and Zadarnowski (2003)

Semantics of IL/I and IL/F

Common Part

$$\text{Op}\ \frac{V \vdash e \Downarrow v}{\begin{array}{llll} & L & \mid V & \mid \mathit{let}\ x = e\ \mathit{in}\ s \\ \longrightarrow & L & \mid V^x_v & \mid s \end{array}}$$

$$\text{If}\ \frac{\mathit{val2bool}(Vx) = i}{\begin{array}{llll} & L & \mid V & \mid \mathit{if}\ x\ \mathit{then}\ s_0\ \mathit{else}\ s_1 \\ \longrightarrow & L & \mid V & \mid s_i \end{array}}$$

$$\text{F-Let}\ \frac{}{\begin{array}{llll} & L & \mid V & \mid \mathit{fun}\ f\,\overline{x} = s\ \mathit{in}\ t \\ \longrightarrow & L,\ f := (V, \overline{x}, s) & \mid V & \mid t \end{array}}$$

$$\text{I-Let}\ \frac{}{\begin{array}{llll} & L & \mid V & \mid \mathit{fun}\ f\,\overline{x} = s\ \mathit{in}\ t \\ \longrightarrow & L,\ f := (\overline{x}, s) & \mid V & \mid t \end{array}}$$

Semantics of IL/I and IL/F

Difference: Application Rule


```
let x = 7 in
fun f () = x in
let x = 5 in f ()
```

$$\text{F-App}\ \frac{}{\begin{array}{rlll} & L,\ f := \left(\boxed{V'}, \overline{x}, s\right),\ L' & \mid \boxed{V} & \mid f\ \overline{y} \\ \longrightarrow & L,\ f := \left(\boxed{V'}, \overline{x}, s\right) & \mid \boxed{V'}^{\overline{x}}_{V\overline{y}} & \mid s \end{array}}$$

$$\text{I-App}\ \frac{}{\begin{array}{rlll} & L,\ f := (\overline{x}, s),\ L' & \mid \boxed{V} & \mid f\ \overline{y} \\ \longrightarrow_I & L,\ f := (\overline{x}, s) & \mid \boxed{V}^{\overline{x}}_{V\overline{y}} & \mid s \end{array}}$$

V': Closure Environment

V: Primary Environment

IL/I without parameters is close to assembly

```
i := 1;
fun f () =
  c := n > 0;
  if c then
    i := n * i;
    n := n − 1;
    f ()
  else
    i
in
  f ()
```

- IL/I without parameters ⟷ Assembly
  1. Control flow as recursion (irreducible ⟷ mutually rec.)
  2. Data flow through imperative variables
- Closely corresponds to assembly language:
  - Scheduling, placement, and registers
- Landin (1965): Algol ⟷ λ-calculus

IL/I without parameters is close to assembly

```
i := 1;
fun f () =
  c := n > 0;
  if c then
    i := n * i;
    n := n - 1;
    f ()
  else
    i
in
  f ()
```

```
  i := 1;
f:
  c := n > 0;
  branchz c r
  i := n * i;
  n := n - 1;
  branch f;
r:
  ret i;
```

- IL/I without parameters $\bullet\!\!-\!\!\bullet$ Assembly
  1. Control flow as recursion (irreducible $\bullet\!\!-\!\!\bullet$ mutually rec.)
  2. Data flow through imperative variables
- Closely corresponds to assembly language:
  - Scheduling, placement, and registers
- Landin (1965): Algol $\bullet\!\!-\!\!\bullet$ λ-calculus

Program Equivalence

Deterministic Reduction Systems

- Observational Equivalence $\sigma \Updownarrow \sigma'$ on (semantic) states σ, σ'
  - σ, σ' terminate with the same observation (error or value)
  - σ, σ' diverge
- Characterization via bisimulation

$$\text{Bisim-Step}\ \frac{\sigma_1 \longrightarrow^+ \sigma_1' \qquad \sigma_2 \longrightarrow^+ \sigma_2' \qquad \sigma_1' \sim \sigma_2'}{\sigma_1 \sim \sigma_2}$$

$$\text{Bisim-Conv}\ \frac{v \in O \qquad \sigma_1 \Downarrow v \qquad \sigma_2 \Downarrow v}{\sigma_1 \sim \sigma_2}$$

- Characterization is
  - sound $\sim \subseteq \Updownarrow$
  - and complete $\Updownarrow \subseteq \sim$

Program Equivalence

For programs from IL/I and IL/F

- Lift $\sim$ to (open) programs: $\stackrel{\circ}{\sim}$

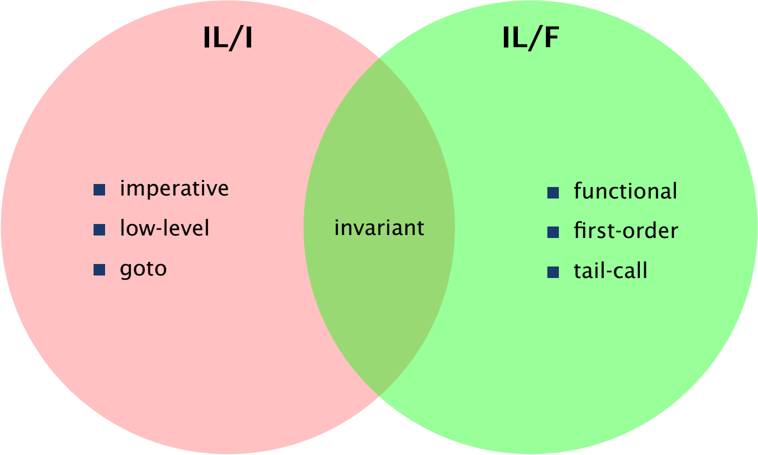
$$s \stackrel{\circ}{\sim} s' \;:\Longleftrightarrow\; \forall L\,V,\; (L, V, s) \sim (L, V, s')$$

- For IL/F, $\stackrel{\circ}{\sim}$ coincides with contextual equivalence $\simeq$

$$\simeq \;=\; \stackrel{\circ}{\sim}$$

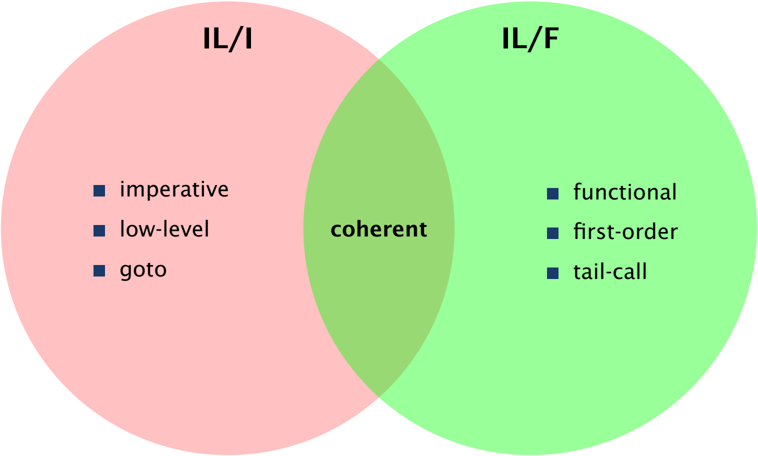
Coherence

Sufficient conditions for invariance



Coherence

Sufficient conditions for invariance



Liveness

Judgement

SAARLAND UNIVERSITY
COMPUTER SCIENCE

$$\boxed{\Lambda \vdash \textbf{\textit{live}}\ s\ :\ \Gamma}$$

Λ	globals of functions
Γ	set of live variables
s	program

Under assumptions Λ about the functions appearing in s,
the behavior of s depends at most on the variables in the set Γ

- inductively defined
- characterizes the post-fixpoints of liveness analysis

Liveness

Rules

$$\text{Live-Op}\ \frac{\mathcal{V}(e) \subseteq \Gamma \qquad \Gamma' \setminus \{x\} \subseteq \Gamma \qquad \Lambda \vdash \textbf{\textit{live}}\ s\ :\ \Gamma'}{\Lambda \vdash \textbf{\textit{live}}\ \mathit{let}\ x\ =\ e\ \mathit{in}\ s\ :\ \Gamma}$$

Liveness

Rules

$$\text{Live-Op}\ \frac{\mathcal{V}(e) \subseteq \Gamma \qquad \Gamma' \setminus \{x\} \subseteq \Gamma \qquad \Lambda \vdash \textbf{\textit{live}}\ s\ :\ \Gamma'}{\Lambda \vdash \textbf{\textit{live}}\ \mathit{let}\ x\ =\ e\ \mathit{in}\ s\ :\ \Gamma}$$

$$\text{Live-Fun}\ \frac{\begin{array}{ll} \Lambda,\ f : \Gamma_f\ \vdash \textbf{\textit{live}}\ t\ :\ \Gamma' & \Gamma' \subseteq \Gamma \\ \Lambda,\ f : \Gamma_f\ \vdash \textbf{\textit{live}}\ s\ :\ \Gamma_f \cup \overline{x} & \Gamma_f \subseteq \Gamma \setminus \overline{x} \end{array}}{\Lambda \vdash \textbf{\textit{live}}\ \mathit{fun}\ f\ \overline{x}\ =\ s\ \mathit{in}\ t\ :\ \Gamma}$$

$$\text{Live-App}\ \frac{\Gamma_f \subseteq \Gamma \qquad \overline{y} \subseteq \Gamma}{\Lambda,\ f : \Gamma_f,\ \Lambda' \vdash \textbf{\textit{live}}\ f\ \overline{y}\ :\ \Gamma}$$

Γ_f is called the **globals** of f

Rules Live-If, Live-App, Live-Return omitted

Coherence

Main Idea

$\boxed{\text{At every call site, primary and closure environment agree on globals}}$

Coherence

Main Idea

$\boxed{\text{At every call site, primary and closure environment agree on globals}}$

Not invariant

```
let x = 7 in
fun f () = x in
let x = 5 in f ()
```

Coherent

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

- Inductively defined judgement $\boxed{\Lambda \mid \Gamma \vdash \textit{\textbf{coh}}\ s}$
- Function f is **available** as long as none of its globals Γ_f is rebound
- Coherence ensures only available functions can be applied

Coherence

Central Theorems

1. Define the set of coherent states $Coh := \bigcup_\Gamma Coh(\Gamma)$
   - A state (L,V,s) is in $Coh(\Gamma)$
     if $\Lambda \mid \Gamma \vdash \textbf{\textit{coh}}\ s$ and side-conditions about L hold for some Λ

Theorem (Coherence is stable under reduction)

Coh is $\longrightarrow$-closed

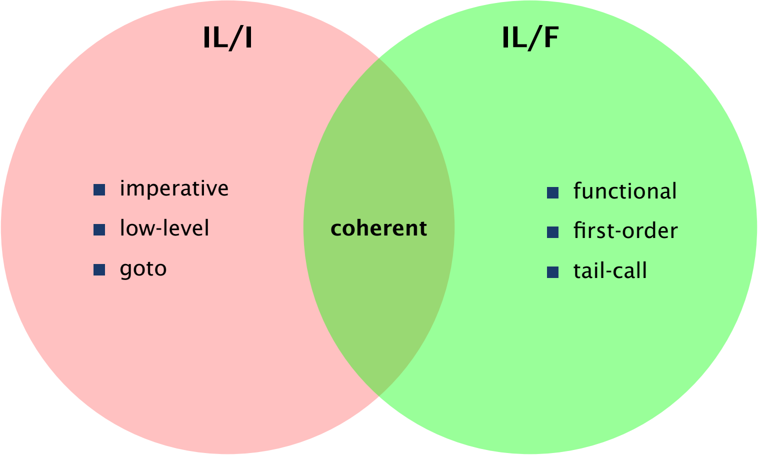
2. Define a bisimulation $\approx_{coh}$
   - Two states are in relation $(L, V, s) \approx_{coh} (strip\, L, V', s)$
     if there is Γ such that $(L, V, s) \in Coh(\Gamma)$ and $V =_\Gamma V'$

Theorem (Coherent programs are invariant)

$\approx_{coh} \subseteq \sim$

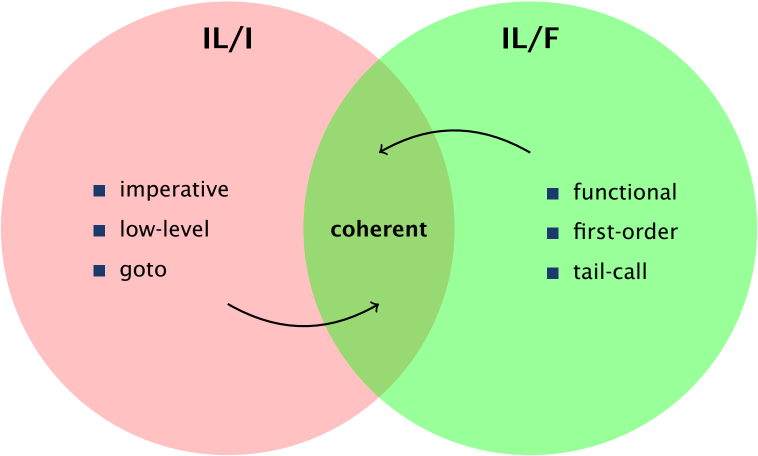
Translating to the coherent fragment

Overview



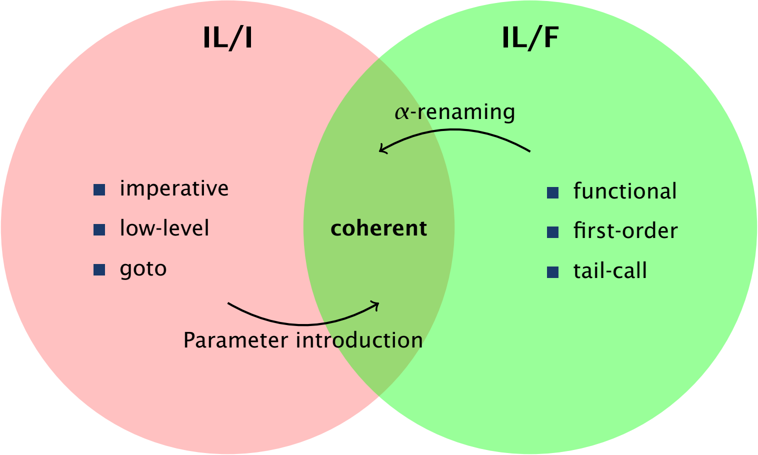
Translating to the coherent fragment

Overview



Translating to the coherent fragment

Overview



Translating to the coherent fragment

Intuition

Not invariant

```
let x = 7 in
fun f () = x in
let x = 5 in f ()
```

Parameter introduction preserves IL/I

```
let x = 7 in
fun f x = x in
let x = 5 in f x
```

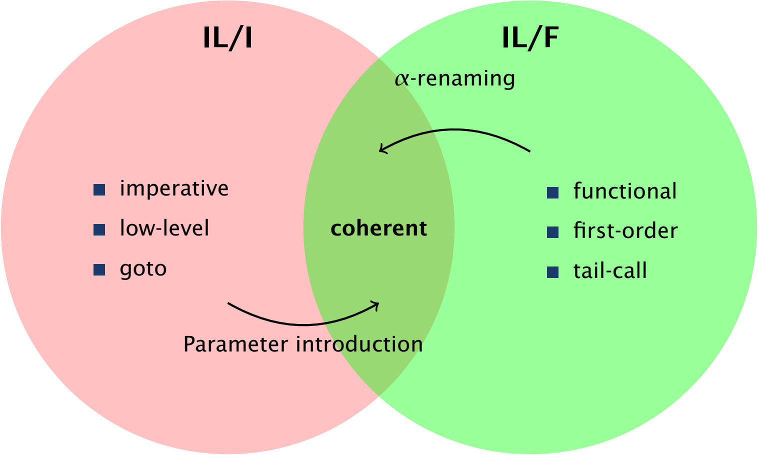
α-renaming preserves IL/F

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

Both are simple to compute from liveness information

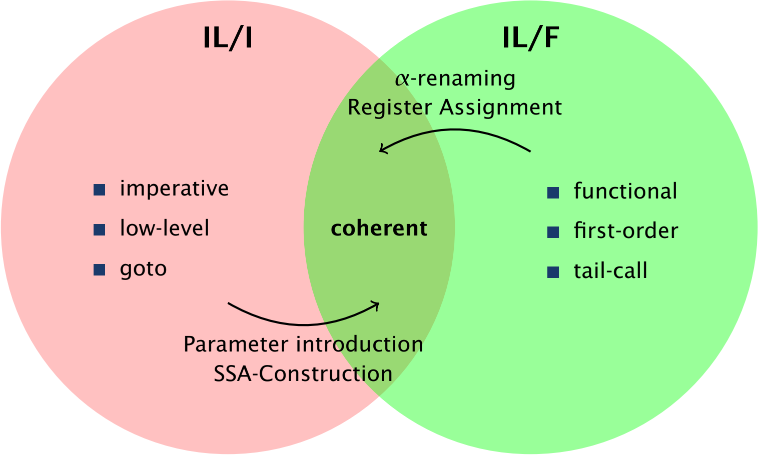
Translating to the coherent fragment

Relation to well-known compilation phases



Translating to the coherent fragment

Relation to well-known compilation phases



Static Single Assignment (SSA)

Motivation

```
  i := 1;
f:


  c := n > 0;
  branchz c r
  i := n * i;
  n := n - 1;
  branch f;
r:
  ret i;
```

- SSA form due to Alpern, Rosen, Wegman, Zadeck (1988)
  1. Each variable assigned at most once
  2. Each variable defined before use
- SSA reduces space requirements of value analysis
  - non-SSA: $O(|s| \cdot |vars(s)|) = O(|s|^2)$
  - SSA: $O(|s|)$

Static Single Assignment (SSA)

Motivation


```
  i := 1;
f:


  c := n > 0;
  branchz c r
  i := n * i;
  n := n - 1;
  branch f;
r:
  ret i;
```

```
i := 1;
f:


  c := n > 0;
  branchz c r
  k := n * i;
  p := n - 1;
  branch f;
r:
  ret i;
```

- SSA form due to Alpern, Rosen, Wegman, Zadeck (1988)
  1. Each variable assigned at most once
  2. Each variable defined before use
- SSA reduces space requirements of value analysis
  - non-SSA: $O(|s| \cdot |vars(s)|) = O(|s|^2)$
  - SSA: $O(|s|)$

Static Single Assignment (SSA)

Motivation

```
  i := 1;
f:
  c := n > 0;
  branchz c r
  i := n * i;
  n := n - 1;
  branch f;
r:
  ret i;
```

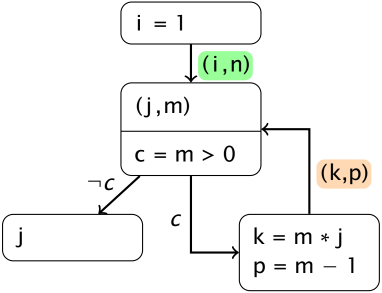
```
i := 1;
f:
  j := φ(i,k)
  m := φ(n,p)
  c := m > 0;
  branchz c r
  k := m * j;
  p := m - 1;
  branch f;
r:
  ret j;
```

- SSA form due to Alpern, Rosen, Wegman, Zadeck (1988)
  1. Each variable assigned at most once
  2. Each variable defined before use
- SSA reduces space requirements of value analysis
  - non-SSA: $O(|s| \cdot |vars(s)|) = O(|s|^2)$
  - SSA: $O(|s|)$

Static Single Assignment (SSA)

Terms and Conditions

```
i := 1;
f:
  j := φ(i,k)
  m := φ(n,p)
  c := n > 0;
  branchz c r
  k := m ∗ j;
  p := m − 1;
  branch f;
r:
  ret j;
```



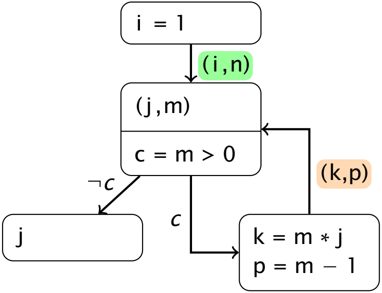
- Where is a variable x defined in an imperative program?
  - In every block that is dominated by the definition of x
  - SSA well-formedness condition involves dominance
- Referential transparency
  - Non-ϕ-assignment introduces equivalence
  - Holds only in dominated part of the program
- Fundamental semantics remains imperative
  - ϕ depends on control flow

Static Single Assignment (SSA)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

SSA is functional programming

```
let i = 1 in
fun f (j,m) =
  let c = m > 0 in
  if c then
    let k = m * j in
    let p = m − 1 in
    f (k,p)
  else
    j
in
  f (i,n)
```



- Correspondence due to Appel (1998) and R. A. Kelsey (1995)
  - Function parameters correspond to ϕ-functions
  - Assignments become let-bindings
- Functional form has several advantages
  - SSA-condition is built in, no additional side conditions
  - Lexical scoping instead of dominance
  - Contextual equivalence instead of referential transparency

Static Single Assignment (SSA)

SSA Construction in the functional setting I

- The following two programs only differ in their nesting structure

```
fun g () = x in
fun f () = g () in
let x = 3 in
f ()
```

```
fun f () =
  fun g () = x in
  g () in
let x = 3 in
f ()
```

- Their coherent translations differ in the number of parameters

```
fun g x = x in
fun f x = g x in
let x = 3 in
f x
```

```
fun f x =
  fun g () = x in
  g () in
let x = 3 in
f x
```

- Number of parameters (ϕ-nodes) depends on nesting structure

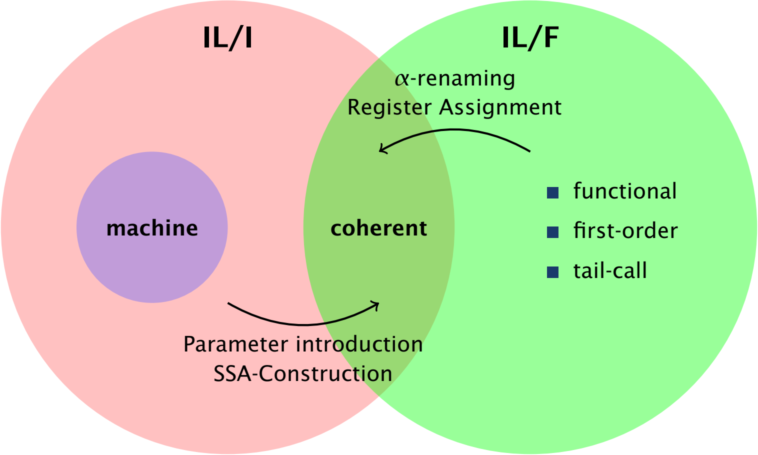
Static Single Assignment (SSA)

SSA Construction in the functional setting II

- SSA construction decomposes into two separate phases
  1. re-nesting block structure
     - requires dominance information
     - dominance hard to verify (Zhao and Zdancewic 2012)
  2. computing additional parameters
     - based on liveness information
     - easy to verify
- If minimial number of parameters (ϕ-nodes) is not required, first step can be omitted
- If source program uses control structures, optimal nesting structure is easy to produce

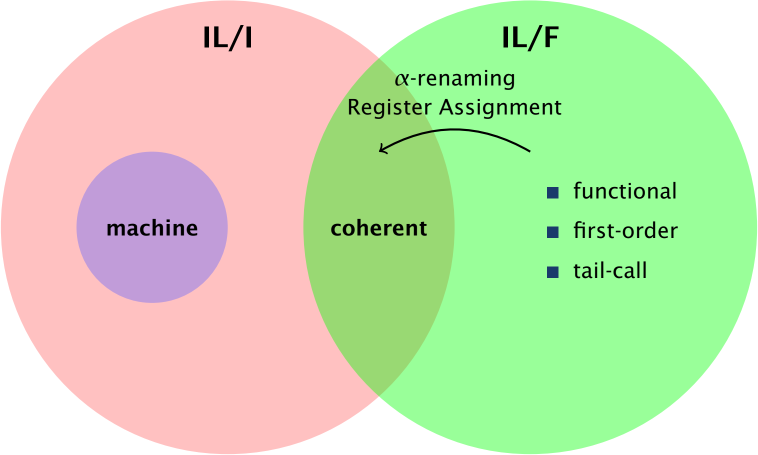
Translating to the coherent fragment II

Relation to well-known compilation phases



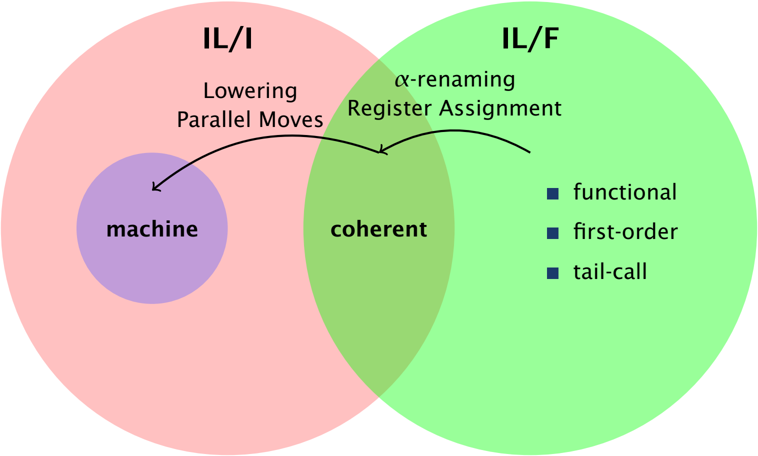
Translating to the coherent fragment II

Relation to well-known compilation phases



Translating to the coherent fragment II

Relation to well-known compilation phases



Register Allocation: Example

- Register assignment
  - α-renaming ρ to coherent program
  - Decidable local injectivity predicate: $\boxed{\Lambda \mid \Gamma \vdash \boldsymbol{inj}_\rho\ s}$
- Parameter elimination
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

4 variables in use: i,j,m,n

```
let i = 1 in
fun f (n,i) =
  if n > 0 then
    let j = n * i in
    let m = n - 1 in
    f (m, j)
  else
    i
in
  f (n,i)
```

Register Allocation: Example

- **Register assignment**
  - α-renaming ρ to coherent program
  - Decidable local injectivity predicate: $\boxed{\Lambda \mid \Gamma \vdash \boldsymbol{inj}_\rho\ s}$
- Parameter elimination
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

4 variables in use: i,j,m,n

```
let i = 1 in
fun f (n,i) =
  if n > 0 then
    let j = n * i in
    let m = n − 1 in
    f (m, j)
  else
    i
in
  f (n,i)
```

2 variables in use: i,n

```
let i = 1 in
fun f (n,i) =
  if n > 0 then
    let i = n * i in
    let n = n − 1 in
    f (n, i)
  else
    i
in
  f (n,i)
```

Register Allocation: Example

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- **Register assignment**
  - α-renaming ρ to coherent program
  - Decidable local injectivity predicate: $\boxed{\Lambda \mid \Gamma \vdash \boldsymbol{inj}_\rho\ s}$
- Parameter elimination
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

4 variables in use: i,j,m,n

```
let i = 1 in
fun f (n,i) =
  if n > 0 then
    let j = n * i in
    let m = n − 1 in
    f (m,j)
  else
    i
in
  f (n,i)
```

2 registers in use: i,n

```
i := 1;
fun f (n,i) =
  if n > 0 then
    i := n * i;
    n := n − 1;
    f (n,i)
  else
    i
in
  f (n,i)
```

Register Allocation: Example

- Register assignment
  - α-renaming ρ to coherent program
  - Decidable local injectivity predicate: $\boxed{\Lambda \mid \Gamma \vdash \boldsymbol{inj}_\rho\ s}$
- **Parameter elimination**
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

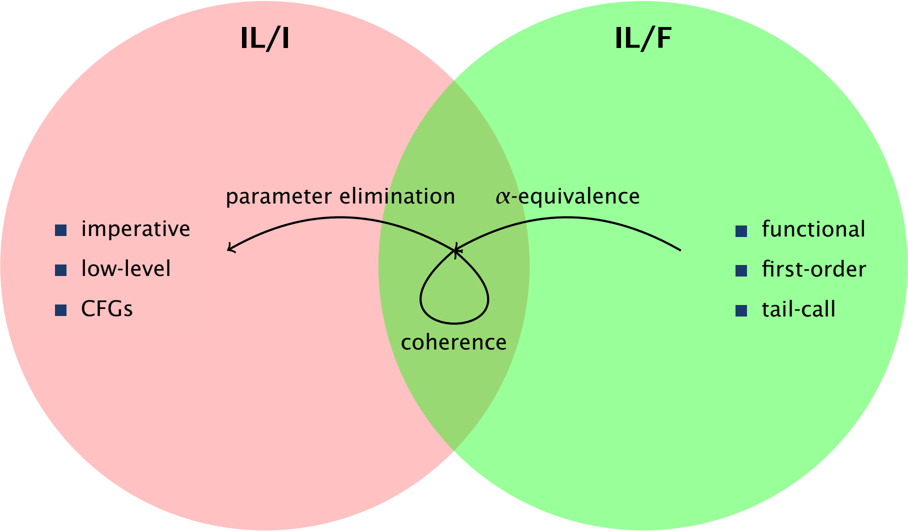
4 variables in use: i,j,m,n

```
let i = 1 in
fun f (n,i) =
  if n > 0 then
    let j = n * i in
    let m = n - 1 in
    f (m,j)
  else
    i
in
  f (n,i)
```

2 registers in use: i,n

```
i := 1;
fun f () =
  if n > 0 then
    i := n * i;
    n := n - 1;
    f ()
  else
    i
in
  f ()
```

Register Allocation: Correctness Argument



Example: Copy Propagation

$$\text{cp}\ \theta\ (let\ x\ =\ y\ in\ s)\ =\ \text{cp}\ (\theta^x_y)\ s$$

- Should be simple to prove: It's a reduction under a context
- Induction can be used, because bisimulation is contextual
- Copy propagation *removes* statements (introducing stutter steps), but this does not matter in the inductive proof
- Contextuality makes trivial cases trivial
- No additional SSA-related side conditions required

Copy Propagation

Correctness Proof

$$\text{cp}\ \theta\ (let\ x\ =\ y\ in\ s)\ =\ \text{cp}\ (\theta_y^x)\ s$$

Lemma

$\theta s\ \simeq\ \text{cp}\, \theta\, s$

We have to show

$$\theta(let\ x\ =\ y\ in\ s)\ \simeq\ \text{cp}\, \theta\, (let\ x\ =\ y\ in\ s)$$

Switch to $\sim$, and exploit closedness under reduction:

$$L,\ E^z_{E(\theta y)},\ \theta^x_z\, s \sim\ L',\ E,\ \text{cp}\, (\theta^x_{\theta y})\, s$$

Since z fresh for s, this is equivalent to the inductive hypothesis:

$$L,\ E,\ \theta^x_{\theta y}\, s \sim\ L',\ E,\ \text{cp}\, (\theta^x_{\theta y})\, s$$

Challenges

- **Compositionality**
  - External events
  - Non-determinism
  - Higher-order
- **Transformations**
  - Compositional methods for verification of structure altering transformations
    - elimination of dead parameters
    - elimination of constant parameters

Conclusion

1. Coherence simplifies translation between imperative and functional programs
2. Allows to transfer correctness arguments from the imperative to the functional side
   - Contextual equivalence is useful
   - SSA is built in
   - Functional language can be used up to register allocation
3. Thesis including admit-free formalization in Coq
   `http://www.ps.uni-saarland.de/~sdschn/master`

Thank you for listening! Questions? I

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Appel, Andrew W. (1998). “SSA is Functional Programming”. In: *SIGPLAN Notices* 4.

Barthe, Gilles, Delphine Demange, and David Pichardie (2012). “A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert”. In: *Programming Languages and Systems - 21st European Symposium on Programming*. Ed. by Helmut Seidl. Lecture Notes in Computer Science. Tallinn, Estonia.

Beringer, Lennart, Kenneth MacKenzie, and Ian Stark (2003). “Grail: a Functional Form for Imperative Mobile Code”. In: *Electr. Notes Theor. Comput. Sci.* 1.

Braun, Matthias, Sebastian Buchwald, and Andreas Zwinkau (2011). *Firm—A Graph-Based Intermediate Representation*. Tech. rep. 35. Karlsruhe Institute of Technology.

Chakravarty, Manuel M. T., Gabriele Keller, and Patryk Zadarnowski (2003). “A Functional Perspective on SSA Optimisation Algorithms”. In: *Electr. Notes Theor. Comput. Sci.* 2.

Click, Cliff and Michael Paleczny (1995). “A Simple Graph-Based Intermediate Representation”. In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. Ed. by Michael D. Ernst. San Francisco, CA, USA.

Johnson, Neil and Alan Mycroft (2003). “Combined Code Motion and Register Allocation Using the Value State Dependence Graph”. In: *Compiler Construction, 12th International Conference. Proceedings*. Ed. by Görel Hedin. Lecture Notes in Computer Science. Warsaw, Poland.

Kelsey, Richard A. (1995). “A Correspondence Between Continuation Passing Style and Static Single Assignment Form”. In: *SIGPLAN Notices* 3. issn: 0362-1340.

Thank you for listening! Questions? II

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Kelsey, Richard and Paul Hudak (1989). "Realistic Compilation by Program Transformation". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*. Austin, Texas, USA.

Landin, Peter J. (1965). "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *Commun. ACM* 2.

Leroy, Xavier (2009). "Formal verification of a realistic compiler". In: *Communications of the ACM* 7.

Rideau, Laurence, Bernard Paul Serpette, and Xavier Leroy (2008). "Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves". In: *Journal of Automated Reasoning* 4.

Tate, Ross et al. (2009). "Equality saturation: a new approach to optimization". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Ed. by Zhong Shao and Benjamin C. Pierce. Savannah, GA, USA.

Zhao, Jianzhou and Steve Zdancewic (2012). "Mechanized Verification of Computing Dominators for Formalizing Compilers". In: *Certified Programs and Proofs - Second International Conference. Proceedings*. Ed. by Chris Hawblitzel and Dale Miller. Lecture Notes in Computer Science. Kyoto, Japan.

Abstract

To use a functional intermediate language for the compilation of an imperative language two translations are of particular importance: Translating from the imperative source language (e.g. C) to the functional language, and translating from the functional intermediate language to an imperative target language (e.g. assembly).

This talk discusses the above translations in the context of the intermediate language IL. IL comes with an imperative and a functional interpretation. IL/I (imperative interpretation) is a register transfer language close to assembly. IL/F (functional interpretation) is a first-order functional language with a tail-call restriction. We devise the decidable notion of coherence to identify programs on which the semantics of IL/I and IL/F coincide. Translations between the interpretations are obtained from equivalence-preserving translations to the coherent fragment. The translation from IL/I to IL/F corresponds to SSA construction. The translation from IL/F to IL/I can be seen as register assignment.

The translations are formalized and proven correct using the proof assistant Coq. Extraction yields translation-validated implementations of SSA construction and register assignment from the above translations.