Translating In and Out of a Functional Intermediate Language

Sigurd Schneider, Sebastian Hack, Gert Smolka

Talk at Inria Rocquencourt

17.03.2014
Overview

imperative  functional  machine
Overview

Why?  

Functional

How?  

Imperative

Machine
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 🜀 Functional
   ▶ Relation to SSA-construction and register assignment
   ▶ Inductive correctness proof of copy propagation
Overview

How to integrate a functional intermediate language

imperative
Overview
How to integrate a functional intermediate language

imperative

functional
Overview

How to integrate a functional intermediate language

Overview
How to integrate a functional intermediate language

Syntax of IL

\[
s, t ::= \text{let } x = e \text{ in } s \\
| \text{if } x \text{ then } s \text{ else } t \\
| x \\
| \text{fun } f \bar{x} = s \text{ in } t \\
| f \bar{x}
\]

- 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

\[
\begin{align*}
\text{Op} & \quad \frac{V \vdash e \downarrow v}{L \mid V \mid \text{let } x = e \text{ in } s} \\
& \quad \rightarrow L \mid V_x^\downarrow \mid s \\
\text{If} & \quad \frac{\text{val2bool}(Vx) = i}{L \mid V \mid \text{if } x \text{ then } s_0 \text{ else } s_1} \\
& \quad \rightarrow L \mid V \mid s_i \\
\text{F-Let} & \quad \frac{L \mid V \mid \text{fun } f \overline{x} = s \text{ in } t}{L, f := (V, \overline{x}, s) \mid V \mid t} \\
\text{I-Let} & \quad \frac{L \mid V \mid \text{fun } f \overline{x} = s \text{ in } t}{L, f := (\overline{x}, s) \mid V \mid t}
\end{align*}
\]
Semantics of IL/I and IL/F

Difference: Application Rule

1. \texttt{let } x = 7 \texttt{ in}
2. \texttt{fun } f () = x \texttt{ in}
3. \texttt{let } x = 5 \texttt{ in } f ()

\[
\begin{align*}
\text{F-App} & : L, f := (V', \bar{x}, s), L' \mid V \mid f \bar{y} \\
& \rightarrow L, f := (V', \bar{x}, s) \mid V'_{V \bar{y}} \mid s
\end{align*}
\]

\[
\begin{align*}
\text{I-App} & : L, f := (\bar{x}, s), L' \mid V \mid f \bar{y} \\
& \rightarrow I L, f := (\bar{x}, s) \mid V_{V \bar{y}} \mid s
\end{align*}
\]

\(V':\) Closure Environment

\(V:\) Primary Environment
IL/I without parameters is close to assembly

```
1 i := 1;
2 \textbf{fun} f() =
3 \hspace{1em} c := n > 0;
4 \hspace{1em} \textbf{if} c \textbf{ then}
5 \hspace{2em} i := n \ast i;
6 \hspace{2em} n := n - 1;
7 \hspace{1em} f()
8 \hspace{1em} \textbf{else}
9 \hspace{2em} i
10 \textbf{in}
11 f()
```

- 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 $\bullet\bullet$ $\lambda$-calculus
IL/I without parameters is close to assembly

\[
i := 1;\ 
\textbf{fun} \ f(\ ) = \begin{array}{l}
  c := n > 0; \\
  \textbf{if} \ c \ \textbf{then} \\
  \quad i := n \times i; \\
  \quad n := n - 1; \\
  \quad f(\ ) \\
  \textbf{else} \\
  \quad i \\
\end{array} \begin{array}{l}
  \textbf{in} \\
  \quad f(\ )
\end{array}
\]

IL/I without parameters $\rightarrow$ Assembly

1. Control flow as recursion (irreducible $\rightarrow$ mutually rec.)
2. Data flow through imperative variables

Closely corresponds to assembly language:

- Scheduling, placement, and registers

Landin (1965): Algol $\rightarrow$ $\lambda$-calculus
Program Equivalence
Deterministic Reduction Systems

- Observational Equivalence $\sigma \diamondsuit \sigma'$ on (semantic) states $\sigma, \sigma'$
  - $\sigma, \sigma'$ terminate with the same observation (error or value)
  - $\sigma, \sigma'$ diverge

- Characterization via bisimulation

\[
\begin{array}{c}
\text{Bisim-Step} \\
\sigma_1 \xrightarrow{+} \sigma_1' \\
\sigma_2 \xrightarrow{+} \sigma_2' \\
\sigma_1 \sim \sigma_2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{Bisim-Conv} \\
\nu \in O \\
\sigma_1 \Downarrow \nu \\
\sigma_2 \Downarrow \nu \\
\sigma_1 \sim \sigma_2 \\
\end{array}
\]

- Characterization is
  - sound $\sim \subseteq \diamondsuit$
  - and complete $\diamondsuit \subseteq \sim$
Program Equivalence
For programs from IL/I and IL/F

- Lift $\sim$ to (open) programs: $\sim$

$$s \sim s' : \iff \forall L, V, (L, V, s) \sim (L, V, s')$$

- For IL/F, $\sim$ coincides with contextual equivalence $\simeq$

$$\simeq = \sim$$
Coherence
Sufficient conditions for invariance

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

invariant
Coherence
Sufficient conditions for invariance

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

coherent
Liveness

Judgement

\[ \Lambda \vdash \text{live } s : \Gamma \]

- \( \Lambda \): globals of functions
- \( \Gamma \): set of live variables
- \( s \): program

**Under assumptions** \( \Lambda \) **about the functions appearing in** \( s \), **the behavior of** \( s \) **depends at most on the variables in the set** \( \Gamma \)

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

Rules

\[
\text{Live-Op} \quad \frac{\mathcal{V}(e) \subseteq \Gamma \quad \Gamma' \setminus \{x\} \subseteq \Gamma \quad \Lambda \vdash \text{live } s : \Gamma'}{\Lambda \vdash \text{live let } x = e \text{ in } s : \Gamma}
\]
Liveness

Rules

\[
\text{Live-Op} \quad \frac{\forall (e) \subseteq \Gamma \quad \Gamma' \setminus \{x\} \subseteq \Gamma}{\Lambda \vdash live s : \Gamma'}
\]

\[
\quad \quad \quad \quad \frac{\Lambda' \vdash live \text{ let } x = e \text{ in } s : \Gamma}{\Lambda \vdash live \text{ let } x = e \text{ in } s : \Gamma}
\]

\[
\text{Live-Fun} \quad \frac{\Lambda, f : \Gamma_f \vdash live t : \Gamma'}{\Lambda' \vdash live fun f \bar{x} = s \text{ in } t : \Gamma}
\]

\[
\quad \quad \quad \quad \frac{\Lambda, f : \Gamma_f \vdash live s : \Gamma_f \cup \bar{x}}{\Lambda' \vdash live fun f \bar{x} = s \text{ in } t : \Gamma}
\]

\[
\text{Live-App} \quad \frac{\Gamma_f \subseteq \Gamma \quad \bar{y} \subseteq \Gamma}{\Lambda, f : \Gamma_f, \Lambda' \vdash live f \bar{y} : \Gamma}
\]

\[
\quad \quad \quad \quad \frac{\Gamma_f \subseteq \Gamma \quad \bar{y} \subseteq \Gamma}{\Lambda, f : \Gamma_f, \Lambda' \vdash live f \bar{y} : \Gamma}
\]

\(\Gamma_f\) is called the \textbf{globals} of \(f\)

Rules Live-If, Live-App, Live-Return omitted
Coherence
Main Idea

At every call site, primary and closure environment agree on globals
Coherence
Main Idea

At every call site, primary and closure environment agree on globals

Not invariant

1 \textbf{let} \hspace{0.5em} x = 7 \hspace{0.5em} \textbf{in}
2 \textbf{fun} \hspace{0.5em} f () = x \hspace{0.5em} \textbf{in}
3 \textbf{let} \hspace{0.5em} x = 5 \hspace{0.5em} \textbf{in} \hspace{0.5em} f ()

Coherent

1 \textbf{let} \hspace{0.5em} x = 7 \hspace{0.5em} \textbf{in}
2 \textbf{fun} \hspace{0.5em} f () = x \hspace{0.5em} \textbf{in}
3 \textbf{let} \hspace{0.5em} y = 5 \hspace{0.5em} \textbf{in} \hspace{0.5em} f ()

- Inductively defined judgement $\Lambda \mid \Gamma \vdash coh \, s$
- Function $f$ is \textbf{available} as long as none of its globals $\Gamma_f$ 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 $\Lambda | \Gamma \vdash coh \ s$ and side-conditions about $L$ hold for some $\Lambda$

   **Theorem (Coherence is stable under reduction)**
   $Coh$ is $\rightarrow$-closed

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

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

coherent
Translating to the coherent fragment

Overview

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

coherent
Translating to the coherent fragment

Overview

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

α-renaming
Parameter introduction

coherent
Translating to the coherent fragment

Intuition

Not invariant

1 let x = 7 in
2 fun f () = x in
3 let x = 5 in f ()

Parameter introduction preserves IL/I

1 let x = 7 in
2 fun f x = x in
3 let x = 5 in f x

α-renaming preserves IL/F

1 let x = 7 in
2 fun f () = x in
3 let y = 5 in f ()

Both are simple to compute from liveness information
Translating to the coherent fragment

Relation to well-known compilation phases

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

α-renaming
Parameter introduction

coherent
Translating to the coherent fragment

Relation to well-known compilation phases

IL/I
- imperative
- low-level
- goto

IL/F
- functional
- first-order
- tail-call

coherent

Parameter introduction
SSA-Construction

α-renaming
Register Assignment
Static Single Assignment (SSA)

Motivation

```
1 i := 1;
2 f:
3
4 c := n > 0;
5 branchz c r
6 i := n * i;
7 n := n - 1;
8 branch f;
9 r:
10 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

```
1 i := 1;
2 f:
3
c := n > 0;
4 branchz c r
5 i := n * i;
6 n := n - 1;
7 branch f;
8 r:
9 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

1. $i := 1$
2. $f :$
3. $c := n > 0$
4. branchz $c$ $r$
5. $i := n \times i$
6. $n := n - 1$
7. branch $f$
8. $r :$
9. ret $i$

1. $i := 1$
2. $f :$
3. $j := \phi(i, k)$
4. $m := \phi(n, p)$
5. $c := m > 0$
6. branchz $c$ $r$
7. $k := m \times j$
8. $p := m - 1$
9. branch $f$
10. $r :$
11. 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: $O(|s|)$
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-\( \phi \)-assignment introduces equivalence
  - Holds only in dominated part of the program

Fundamental semantics remains imperative
- \( \phi \) depends on control flow
Static Single Assignment (SSA)

SSA is functional programming

```plaintext
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)
```

  - Function parameters correspond to $\phi$-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
The following two programs only differ in their nesting structure

```
1 fun g () = x in
2 fun f () = g () in
3 let x = 3 in
4 f ()
```

```
1 fun f () =
2 fun g () = x in
3 g () in
4 let x = 3 in
5 f ()
```

Their coherent translations differ in the number of parameters

```
1 fun g x = x in
2 fun f x = g x in
3 let x = 3 in
4 f x
```

```
1 fun f x =
2 fun g () = x in
3 g () in
4 let x = 3 in
5 f x
```

Number of parameters (\(\phi\)-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 minimal number of parameters (\(\phi\)-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

IL/I
- machine
- Parameter introduction
- SSA-Construction

IL/F
- α-renaming
- Register Assignment
- functional
- first-order
- tail-call

coherent
Translating to the coherent fragment II

Relation to well-known compilation phases
Translating to the coherent fragment II

Relation to well-known compilation phases

- Lowering
- Parallel Moves

IL/I

- machine

IL/F

- α-rename
- Register Assignment

coherent

- functional
- first-order
- tail-call
Register Allocation: Example

- Register assignment
  - $\alpha$-renaming $\rho$ to coherent program
  - Decidable local injectivity predicate: $\Lambda \vdash \text{inj}_\rho s$

- Parameter elimination
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

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

```plaintext
1 let i = 1 in
2 fun f (n, i) =
3   if n > 0 then
4     let j = n * i in
5     let m = n - 1 in
6     f (m, j)
7 else
8   i
9 in
10 f (n, i)
```
Register Allocation: Example

- **Register assignment**
  - $\alpha$-renaming $\rho$ to coherent program
  - Decidable local injectivity predicate: $\Lambda \vdash \text{inj}_\rho s$

- **Parameter elimination**
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

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

```plaintext
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

```plaintext
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

**Register assignment**
- \( \alpha \)-renaming \( \rho \) to coherent program
- Decidable local injectivity predicate: \( \Lambda \mid \Gamma \vdash inj_\rho s \)

**Parameter elimination**
- parallel assignment [Rideau, Serpette, and Leroy 2008]

4 variables in use: \( i,j,m,n \)  
2 registers in use: \( i,n \)

```
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)
```

```
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
  - $\alpha$-renaming $\rho$ to coherent program
  - Decidable local injectivity predicate: $\Lambda | \Gamma \vdash inj_{\rho} \ s$

- Parameter elimination
  - parallel assignment [Rideau, Serpette, and Leroy 2008]

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

1 let i = 1 in
2 fun f (n, i) =
3   if n > 0 then
4     let j = n * i in
5     let m = n - 1 in
6     f (m, j)
7 else
8 i
9 in
10 f (n, i)

2 registers in use: i,n

1 i := 1;
2 fun f () =
3   if n > 0 then
4     i := n * i;
5     n := n - 1;
6     f ()
7 else
8 i
9 in
10 f ()
Register Allocation: Correctness Argument

IL/I

- imperative
- low-level
- CFGs

IL/F

- functional
- first-order
- tail-call

parameter elimination

α-equivalence

coherence
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

\[ \text{cp } \theta \left( \text{let } x = y \text{ in } s \right) = \text{cp } \left( \theta_y^x \right) s \]

**Lemma**

\[ \theta s \simeq \text{cp } \theta s \]

We have to show

\[ \theta \left( \text{let } x = y \text{ in } s \right) \simeq \text{cp } \theta \left( \text{let } x = y \text{ in } s \right) \]

Switch to \( \sim \), and exploit closedness under reduction:

\[ L, \ E^z \left( \theta_Y \right), \ \theta^x_z \ s \sim \ L', \ E, \ \text{cp } \left( \theta^x_Y \right) s \]

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

\[ L, \ E, \ \theta^x_{\theta_Y} \ s \sim \ L', \ E, \ \text{cp } \left( \theta^x_{\theta_Y} \right) s \]
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


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


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.